UNIVERSIDADE ESTADUAL DO NORTE DO PARANÁ

FACULDADES LUIZ MENEGHEL

LILIAN SALVI

UMA ANÁLISE SOBRE AS LINGUAGENS DE MODELAGEM ORIENTADA A ASPECTOS APLICADA EM UM ESTUDO DE CASO

Bandeirantes

2007

LILIAN SALVI

UMA ANÁLISE SOBRE AS LINGUAGENS DE MODELAGEM ORIENTADA A ASPECTOS APLICADA EM UM ESTUDO DE CASO

Trabalho de Conclusão de Curso submetido às Faculdades Luiz Meneghel da Universidade Estadual do Norte do Paraná, como requisito parcial para a obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Prof. Ms. André Luís Andrade Menolli.

Bandeirantes

2007

LILIAN SALVI

UMA ANÁLISE SOBRE AS LINGUAGENS DE MODELAGEM ORIENTADA A ASPECTOS APLICADA EM UM ESTUDO DE CASO

Trabalho de Conclusão de Curso submetido às Faculdades Luiz Meneghel da Universidade Estadual do Norte do Paraná, como requisito parcial para a obtenção do grau de Bacharel em Sistemas de Informação.

COMISSÃO EXAMINADORA

_____

Prof. Ms. André Luís Andrade Menolli
Faculdades Luiz Meneghel

_____

Profª. Ms. Cristiane Yanase H. de Castro
Faculdades Luiz Meneghel

_____

Prof. Ms. Roberto Vedoato
Faculdades Luiz Meneghel

Bandeirantes, __ de_____ de 2007.

Aos meus pais, grandes guerreiros e fonte de inspiração para todas as conquistas de minha vida, a vocês todo o meu amor, reconhecimento e agradecimento por tudo.

AGRADECIMENTOS

Em primeiro lugar agradeço a Deus, pois sem Ele tenho certeza que não teria nem ao menos conseguido começar uma faculdade, sou muito agradecida por tudo que Ele fez e que sempre faz por mim.

Sou eternamente grata aos meus pais, que são o meu espelho, meu chão, minha estrutura, meu porto seguro, que às vezes por mais que eu estivesse ausente estavam presentes em meu coração sempre me dando forças, amor e votos de confiança para que eu pudesse conseguir alcançar este objetivo tão sonhado por nós, amo vocês! Agradeço também as minhas irmãs, Ligian e Larissa, que apesar dos conflitos decorrentes do dia-a-dia nos damos muito bem, obrigado pelo companheirismo, afeto e amor.

Aos meus familiares que deram ajuda, força e muitos conselhos para eu seguir em frente, e que nos momentos mais difíceis se fizeram presentes, agradeço especialmente a Tia Lídia, Tio Sola, Tio Jura, Tia Izabel, e finalmente a grande e maravilhosa Tia Batota, que no momento mais difícil da faculdade esteve presente me apoiando e fazendo com que eu não desacreditasse nessa grande vitória, amo vocês, muito obrigada por tudo.

Agradeço também aos meus amigos, que não são muitos, mas que são suficientes para me fazer completa, estes, que apesar de perdas e da distância nos mantemos muito unidos. Deram-me forças pra alcançar tudo que sempre objetivei, dentre eles estão: Anna Karolina, Isa Brito, Isabella, Nalige, Sheila, Vanessa, Marquinho e Tiagão, e aos primos-amigos, pois além de primos são grandes amigos!! Pra vocês, Alex, Amandinha, Chone, Iza Zanini, Juliana e Suelen Big Night o mesmo agradecimento dedicado aos amigos... Amo muito vocês, muito obrigado por tudo!

Sou muito grata aos mestres, que me “agüentaram” durante estes anos transmitindo o conhecimento necessário com muito profissionalismo e, em alguns casos amizade também, aos que me colocaram obstáculos o meu agradecimento, pois foi através deles que aprendi e cresci, aos que além de tudo foram amigos agradeço em dobro, pois se fizeram eternos para mim!

E finalmente agradeço aos incríveis amigos da faculdade, estes com os quais vivi certamente os melhores anos da minha vida, os inesquecíveis anos de faculdade, dias de festas, farra, jogos (de madrugada e universitário), conversas intermináveis, estudos, trabalhos – sem dia e hora pra começar e pra acabar; enfim, alegrias, tristezas, incertezas, derrotas e vitórias e, por fim, união, afeto, carinho, companheirismo e AMOR! Tenham certeza que já são eternos em meu coração! A: Camila e Renata (Amigas de kit!), Amanda, Carol, Fernanda (Mercedes), Karina e

Mariana (amigas de classe e extraclasse), todo o meu amor e agradecimento por terem sido vocês, pessoas ilustres os meus grandes amigos durante esta fase maravilhosa. Amo vocês!

Agradeço também aos colegas de classe que se fizeram não tão menos importantes, nas aulas intermináveis, nos dias de provas e seminários, na ajuda em trabalhos e em estudos pré-provas, dentre eles – Cido e Pedro. A todos, minha gratidão por tudo!

Ao meu orientador Prof. Ms. André (Menolli), que me auxiliou em tudo, me ensinando, e me agüentando durante os momentos de pré TCC, mesmo naqueles em que estive mais ausente que presente, agradeço pela paciência e dedicação destinada a mim, fico lisonjeada por ter um grande profissional como meu orientador, muito obrigada por tudo!

E ao André de Oliveira, que não mediu esforços para me ajudar com seus conhecimentos na execução do Estudo de caso, muito obrigada por tudo André!

Por fim, agradeço a todos que acreditaram e também aos que não acreditaram nesta minha conquista, pois foi através de vocês que tirei forças de onde eu não tinha para concretizar este grande sonho! Muito obrigada!

“É melhor tentar e falhar, que preocupar-se, e ver a vida passar, é melhor tentar ainda que em vão, que sentar-se fazendo nada até o final.
Eu prefiro na chuva caminhar, que em dias tristes em casa me esconder, prefiro ser feliz, embora louco, que em conformidade viver!”

Martin Luther King

RESUMO

Com o aumento da complexidade dos sistemas, a programação orientada a objetos já não tem sido eficaz em alguns pontos, como por exemplo, a dificuldade em se modularizar corretamente todos os interesses de um sistema. A ineficiência da POO em alguns casos ocasiona em um sistema o espalhamento e entrelaçamento de código. Para a solução deste grande problema foi criada a Programação Orientada a Aspectos – que identifica e modulariza corretamente os interesses, sejam eles requisitos funcionais ou não-funcionais. Porém este paradigma ainda tem grandes desafios, como a determinação de uma linguagem de modelagem, já que ainda não possui uma linguagem padrão. Através da modelagem de software é possível especificar a estrutura e o comportamento do sistema. Então a MOA (Modelagem Orientada a Aspectos) tem grandes dificuldades de encontrar uma linguagem que modele exatamente todas as suas operações com coesão e clareza. Neste trabalho é apresentado um estudo sobre algumas abordagens de linguagens de modelagem orientada a aspectos que estão sendo usadas atualmente. Dentre as apresentadas, são salientadas a UMLaut e a AODM – ambas de extensões UML. A abordagem AODM é aplicada em um estudo de caso de um sistema de reservas de laboratórios online e através de Bluetooth, os estudos são feitos na área do desktop, o qual são modelados aspectos sistêmicos, tais como, tratamento de exceções, auditoria, gerenciamento de transações e aspectos colaborativos, como, por exemplo, restrições arquiteturais.

Palavras Chave: modelagem orientada a aspectos, UMLaut, AODM, UML.

ABSTRACT

With the increase of the complexity of the systems, the object-oriented programming already hasn't been efficient in some points, as for example, the difficulty to modularize correctly all the concerns of a system. The inefficiency of the POO in some cases causes in a system the scattering and interlacement of code. For the solution of this big problem the Aspect-Oriented Programming was created - that it identifies and modularize the concerns correctly, be requisite functionaries or not-functionaries. However this paradigm still has big challenges, as the determination of a modeling language, already that it isn't possess a language standard. Through the software modeling it is possible to specify the structure and the behavior of the system. So the (Aspect-Oriented Modeling) has big difficulties to find a language that shapes all accurately its operations with cohesion and clarity. In this work is presents a study about some approaches of aspect-oriented modeling languages that are being used currently. Amongst the presented ones, the UMLaut and the AODM - both of extensions UML are pointed out. The AODM approach is applied in a study of case of a system of reserves of laboratories online and through Bluetooth, the studies are made in the area of desktop, which are shaped systemic aspects, such as, handler exceptions, auditor, and collaborative aspects management of transactions, as, for example, architectural restrictions.

Keywords: aspect - oriented modeling, UMLaut, AODM, UML.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE SIGLAS

AODM – Aspect-Oriented design model
aSideML - aSide Language Modeling
MOA - Modelagem Orientada a Aspectos
MOO- Modelagem Orientada a Objetos
OA - Orientação a Aspectos
OO - Orientação a Objetos
POA - Programação Orientada a Aspectos
POO - Programação Orientada a Objetos
UAE - UML-based Aspect Engineering
UML - Unified Modeling Language
UMLaut - Unified Modeling Language All purposes Transformer

SUMÁRIO

1. INTRODUÇÃO

No desenvolvimento de um software, existem vários paradigmas de programação que podem ser aplicados, e um dos mais utilizados é o Orientado a Objetos. Sua base são as *classes* e *objetos*, e este obteve um retorno satisfatório diante os desenvolvedores, por ter como objetivos fundamentais, reduzir a complexidade e aumentar confiabilidade no desenvolvimento de software, produzindo assim softwares mais confiáveis, oferecendo proteção aos dados através do encapsulamento. Os principais conceitos utilizados na Programação Orientada a Objetos (POO), além das classes e objetos, são herança e polimorfismo, que também são utilizados na modelagem de sistemas.

A linguagem de modelagem mais usada em POO é a UML – Linguagem Unificada de Modelagem, linguagem gráfica usada para visualização, especificação, construção e documentação de artefatos de sistemas complexos de software (BOOCH; RUMBAUGH; JACOBSON, 2006).

Segundo Chavez e Garcia, 2003, a orientação a objetos modulariza bem alguns tipos de interesses, como, por exemplo, interesses de negócio: geração de extratos, gerenciamento de contas. No entanto, a programação orientada a objetos não é capaz de modularizar outros tipos de interesses, que se espalham por várias outras classes no sistema. Vários tipos de interesses não se adéquam à decomposição tradicional da POO. Cada decomposição funciona bem para alguns requisitos, mas deixa outros completamente espalhados pela hierarquia de classes.

Por conta da incapacidade da Orientação a Objetos (OO) de modularizar os interesses ocasionam os problemas resultantes da descentralização do código, dentre eles estão:

- **Replicação de código:** um mesmo tratamento pode ser necessário em classes diferentes, que não estejam na mesma árvore de heranças;
- **Dificuldade de manutenção:** a cada alteração, ou correção, a implementação dos interesses implicará em uma varredura, que terá que percorrer todos os locais onde há implementação;

- **Redução de capacidade de reutilização de código:** o tratamento usado para implementação de um interesse qualquer em uma classe genérica pode impedir o reuso em uma classe especializada;
- **Aumento da dificuldade de compreensão:** já que a implementação vai ficar maior, será mais difícil entender o programa. WINCK & GOETTEN JUNIOR (2006).

Para a resolução de tais problemas, foi criado outro paradigma, já com grande aceitação entre os profissionais da área, a Programação Orientada a Aspectos (POA), que para muitos é vista como um complemento da programação orientada a objetos, trata-se de um novo paradigma de desenvolvimento de *software* que provê a separação avançada de interesses[1], e tem solução para o encapsulamento de funcionalidades ortogonais (RAINONE, 2005).

Para a programação orientada a aspectos existem algumas linguagens de modelagem a serem usadas, porém o paradigma ainda não possui uma linguagem de modelagem padrão. Por isso, foi realizado um estudo mais aprofundado para averiguar a viabilidade das linguagens de modelagens mais usadas.

Assim, este trabalho traz uma análise das linguagens orientada a aspectos mais utilizadas atualmente, apontando vantagens e desvantagens, através de um estudo de caso. Deste modo será feito uma comparação entre as mais viáveis, através dos resultados encontrados no estudo de caso.

1.1 Objetivos

Gerais

Fazer um estudo das principais linguagens de modelagem orientada a aspectos, comparando-as a fim de analisar suas principais características, pontos fortes e desvantagens no uso delas na modelagem de um sistema

---

[1] Interesses: do inglês *Concerns,* é um requisito, uma propriedade, uma funcionalidade de um sistema (CHAVEZ e GARCIA, 2003). São as características relevantes de uma aplicação.

POA (Programação Orientada a Aspectos).

Específicos

Revisão Bibliográfica elicitando as principais técnicas e linguagens de modelagem orientada a aspecto.

Estudar as principais características das principais linguagens levantadas, e através de estudo bibliográfico definir quais são as mais utilizadas e melhores estruturadas.

Definir um estudo de caso o qual possa ser aplicado a Modelagem Orientada a Aspecto.

Estudar a fundo duas das principais linguagens levantadas, e aplicá-las na modelagem do estudo de caso proposto.

Comparar os resultados da aplicação das linguagens na modelagem do estudo de caso proposto.

1.2. Justificativas

A programação orientada a aspectos é um paradigma muito novo, criado pela Xerox, em 1997, ainda em fase de pesquisa, pois não possui uma linguagem de modelagem padrão, por isso é necessário que as linguagens que estão sendo utilizadas em Modelagem Orientada a Aspectos (MOA) sejam testadas, para que assim, seja encontrada uma linguagem de modelagem de acordo com os princípios da POA.

1.3. Organização do trabalho

A estrutura do trabalho está distribuída em quatro capítulos, o segundo apresenta os conceitos gerais da Orientação a Objetos e Orientação a Aspectos, e também um estudo sobre a modelagem orientada a aspectos, apontando as abordagens mais usadas atualmente. Já no terceiro capítulo são apresentadas as

linguagens escolhidas para realização do estudo de caso e a realização deste. E, finalmente no quarto capítulo é realizado o estudo de caso, onde são apresentados os resultados e as conclusões do trabalho.

2. REVISÃO DE LITERATURA

Neste capítulo são apresentados os conceitos de Orientação a Objetos e Orientação a Aspectos, e também um estudo sobre a modelagem orientada a aspectos, apontando as abordagens mais usadas atualmente.

2.1. Programação Orientada a Objetos

Muito utilizada por conseguir gerenciar a complexidade crescente dos sistemas sendo construídos nas empresas, a programação orientada a objetos viabiliza o trabalho conjunto de grandes equipes e o cumprimento de metas de qualidade, prazo e orçamento. Aumenta também a produtividade de analistas e programadores pela reutilização de código pronto e depurado escrito em sistemas anteriores ou adquiridos no mercado (LOZANO, 2007).

A programação Orientada a Objetos é baseada em alguns conceitos fundamentais, que são apresentados resumidamente nos tópicos abaixo.

2.1.1 Classes

A classe nada mais é que uma aplicação estruturada em módulos, que agrupam um estado e operações, que possuem características e comportamentos semelhantes. As classes podem ser estendidas e/ou usadas como tipos (cujos elementos são objetos).

Ela é a abstração dos objetos, que implementa as responsabilidades em um sistema (WINCK & GOETTEN JUNIOR, 2006).

São organizadas em hierarquias, especializadas ou generalizadas, podendo conter definição de subclasses. A classe serve de modelo para criação de objetos, por exemplo, as classes estão para os objetos assim como as plantas arquitetônicas estão para as casas (VEDOATO, 2007).

2.1.2. Objetos

Segundo Winck & Goetten Junior (2006; p.18) o conceito básico de orientação a objetos é a decomposição da solução em objetos definidos pelas classes, assim, uma de suas unidades fundamentais é o objeto, que é instância de uma classe.

Ele representa qualquer coisa do mundo real que seja manipulada pelo programa, ou então pode representar blocos de construção do programa (LOZANO, 2007).

Um objeto é capaz de encapsular estado e comportamento da classe, eles são dinâmicos, pois existem somente em tempo de execução do programa.

Quando existe um uso racional de objetos, conforme estabelecido no paradigma de desenvolvimento OO, e sempre obedecendo aos princípios associados à sua definição, aí então, tem-se a “chave” para que bons sistemas sejam desenvolvidos (WINCK & GOETTEN JUNIOR, 2006).

2.1.3. Herança

Uma grande vantagem da programação orientada a objetos é a capacidade de reutilização de *software* através de herança, que, por meio desta, é possível uma subclasse herdar variáveis de instância e métodos da superclasse previamente definidos (DEITEL E DEITEL, 2003).

A herança pode ser simples ou múltipla, a simples deriva apenas de uma superclasse, já a múltipla herda propriedades de mais de uma superclasse.

A subclasse normalmente adiciona seus próprios atributos e métodos, portanto representa um conjunto menor de métodos mais específicos. Ela pode acessar membros *public* e *protected* da superclasse, e também pode usar os membros com sua superclasse, isto é, se estiverem no mesmo pacote; porém não devem acessar os membros *private* , isso ajuda muito na fase de testes, de depuração e na manutenção correta dos sistemas,já que assim as informações ficam ocultados, ou seja, encapsulados. (DEITEL E DEITEL, 2003).

Para que o encapsulamento seja preservado, todas as variáveis de

instância devem ser declaradas *private*, só sendo acessíveis através do método *get* e *set* da classe. Estes métodos acessam o membro *private* de modo que o *get* acessa os dados e o *set* altera os dados.

A criação de subclasses (classes derivadas de uma superclasse) permite o aumento incremental da funcionalidade dos objetos ou da sua especialização. A herança não é seletiva, de modo que todas as propriedades do objeto são objeto (VEDOATO, 2007).

Para criar e inicializar os membros da subclasse é necessário um construtor, que deve chamar primeiro o construtor da superclasse (explícita ou implicitamente), ou seja, os construtores nunca são herdados – eles são específicos para a classe em que são definidos. (DEITEL E DEITEL, 2003).

Um construtor deve ser público, não tendo retorno, e tendo o mesmo nome da classe, se o construtor receber argumentos, eles devem ser especificados nos parênteses após o nome da classe, no comando *new* e aceita sobrecargas também (LOZANO, 2007).

A superclasse possui um relacionamento hierárquico com suas subclasses, essas hierarquias são utilizadas para averiguar a complexidade do sistema, sendo também diretas ou indiretas em relação a superclassse, é direta quando a subclasse estende (*extends*) à superclasse, e indireta quando a classe filha é herdada de níveis acima na hierarquia de classes.

Segundo Deitel & Deitel (2003, p.501), durante todo o processo do desenvolvimento de um *software* orientado a objetos, o desenvolvedor procura aspectos comuns entre as classes e os divide para formar as superclasses. As subclasses são então personalizadas além dos recursos herdados da superclasse.

2.1.4. Polimorfismo

Através do polimorfismo é possível projetar e implementar sistemas que ficam mais facilmente extensíveis, já que uma mesma mensagem é enviada a objetos de classes distintas (DEITEL E DEITEL, 2003).

Existem dois tipos de polimorfismo, o dinâmico e o estático, o primeiro

redefine os métodos com parâmetros idênticos, usa somente uma hierarquia e não tem tipos de retorno diferentes, o segundo sofre uma sobrecarga dos métodos com parâmetros diferentes, podendo ser número, tipo ou ordem, tendo vários tipos de retorno, e é feito em uma mesma classe ou na mesma hierarquia. (LOZANO, 2007).

Em programação orientada a objetos a combinação de herança mais polimorfismo resultam em uma ligação dinâmica que nada mais é que o processo de ligar a mensagem ao método em tempo de execução.

Um grande problema da herança é que uma subclasse pode herdar métodos que ela não precise ou não deveria ter. Para isto não acontecer o desenvolvedor da classe deve assegurar que as propriedades fornecidas por uma classe são apropriadas para subclasses futuras. Mesmo assim, é comum que a subclasse exija que o método execute uma tarefa de maneira que seja especifica para a subclasse. Quando isso acontece, o método da subclasse pode ser redefinido na subclasse por uma implementação apropriada.

2.2. Programação orientada a aspectos

Os paradigmas atuais não atendem às necessidades para a implementação dos requisitos de um sistema completo sem que parte dos conceitos desses paradigmas sejam quebrados (WINCK & GOETTEN JUNIOR, 2006). Para tentar resolver esses problemas, foi proposto o desenvolvimento orientado a aspectos, que consegue fazer a separação avançada de interesses, desde o nível da implementação até outros estágios do processo de desenvolvimento, incluindo especificação de requisitos, análise e projeto.

A programação orientada a aspectos foi criada no ano de 1997, em Palo Alto, nos laboratórios da Xerox, por Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Cideira Lopes, Jean-Marc Loengtier e John Irwin; segundo Winck e Goetten Junior (2006, p. 42), o objetivo da época era construir uma abordagem que fosse um conjunto não necessariamente homogêneo, que permitisse à linguagem de programação, composta por linguagem central e varias linguagens específicas de domínio, expressar de forma correta as características sistêmicas (também chamadas de ortogonais ou transversais) do comportamento do

sistema.

Esta nova abordagem recebeu o nome de Meta-Programação. Caracteriza-se pelo compilador, pois os compiladores destinados a POA não geram produto final (programa compilável, executável ou interpretável), mas sim um novo código. Nesta compilação são acrescidos novos elementos no código, para dar suporte nas novas abstrações. Ainda assim, o código resultante deve ser novamente compilado, para aí sim geram o produto final.

Mais uma característica da meta-programação usada em POA é a reflexão computacional, onde parte do código gerado é destinado a alterar características do programa (WINCK & GOETTEN JUNIOR, 2006).

A POA permite que a implementação de um sistema seja separada em requisitos funcionais e não-funcionais, disponibilizando assim, a abstração de aspectos para a decomposição de interesses sistêmicos.

Os princípios da POA são separar o código referente ao negócio do sistema dos interesses transversais, de forma bem definida e centralizada.

De acordo com Chavez & Garcia (2003) os interesses são modularizados por meio de diferentes abstrações, providas por linguagens, métodos e ferramentas, como mostra o exemplo da Figura 1.

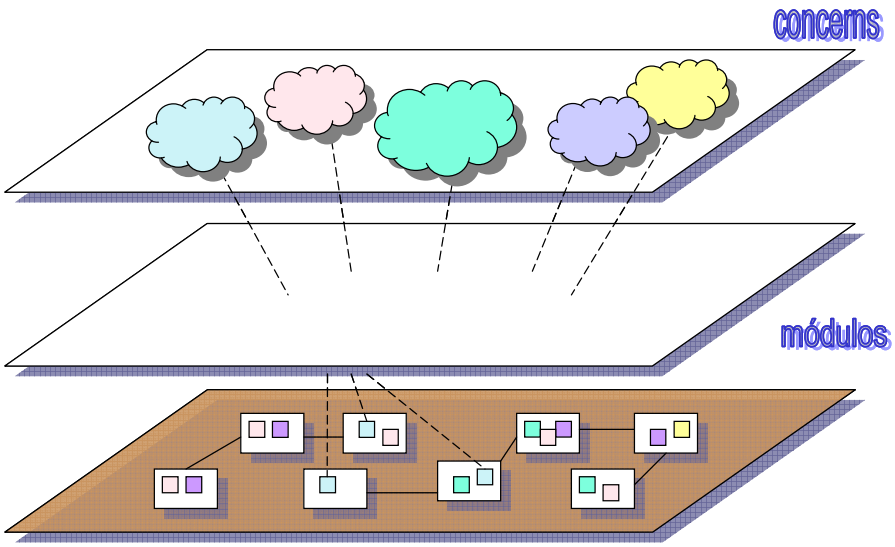

Figura 1 – Separação de interesses com POA. (FONTE: CHAVEZ & GARCIA, 2003)

A POA propõe:

- Identificar e separar interesses;
- Encapsular interesses a módulos coesos e pouco acoplados em vários estágios (requisitos, análise, projeto e implementação);

- Manter rastreabilidade.

Na Figura 1, cada nuvem representa um interesse sistêmico implementado no sistema, por estarem bem separados e definidos, os componentes podem ser melhor reutilizados e a sua manutenção e legibilidade torna-se mais agradável.

2.2.1. Interesses Transversais

De acordo com o dicionário Collins Gem, *concern* significa interesse e *crosscut* significa entrecortar, cortar através, portanto, *crosscutting concern* significa Interesse Transversal.

Na Figura 2, é apresentado o esquema de uma implementação orientada a objetos. Neste exemplo, apesar dos problemas que o paradigma OO pode apresentar, ele satisfaz grande parte dos objetivos para os quais este paradigma foi criado, mas os códigos em vermelho, que representam algum interesse transversal, se espalham por várias classes, fazendo assim com que muitas classes tenham funcionalidades para os quais não foram projetadas. A maioria das propostas tenta solucionar estes problemas sem perder de vista os benefícios da OO.

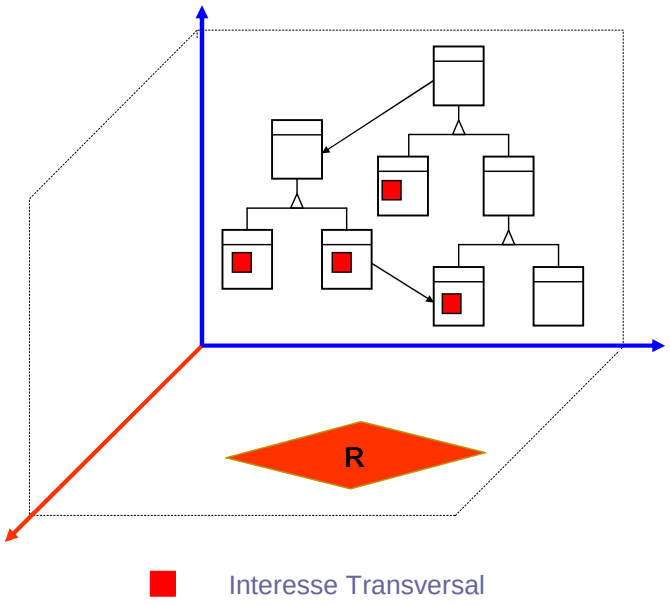

Figura 2 – Situação Problema em OO (FONTE: CHAVEZ & GARCIA, 2003).

A modularização de requisitos transversais é obtida através de 2D. Dessa forma, requisitos transversais, que aparecem espalhados nas classes, podem ser movidos para um único ponto no espaço.

Analisando a Figura 2, uma solução apontada pela POA, é isolar os requisitos transversais e fazer um novo tipo de módulo (R). Compor, combinar nos locais desejados e reutilizar, combinando em outros locais.

Dessa forma, fazendo as modificações necessárias à solução em POA é apresentada na Figura 3, o qual os requisitos que estavam espalhados por diversas classes ficam isolados em R – nos Interesses Transversais.

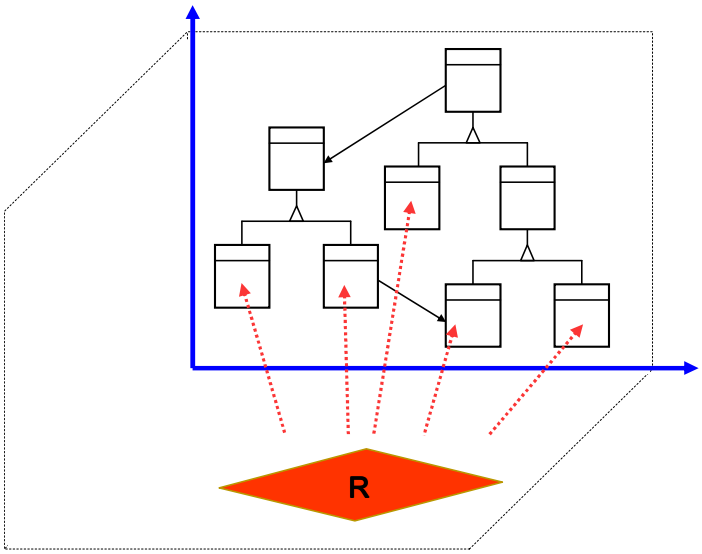

Figura 3 – Solução do problema com POA (FONTE: CHAVEZ & GARCIA, 2003).

Com a modularização dos interesses transversais feita na solução encontrada na figura acima, há uma nova abstração de dados, o aspecto.

Os benefícios encontrados com a utilização da POA são a facilidade de compreensão, manutenção e reutilização WINCK & GOETTEN JUNIOR (2006).

2.2.2. Composição de um sistema orientado a aspectos

Piveta (2001) *apud* Winck & Goetten Junior (2006, p. 45), afirma que um sistema que utiliza a programação orientada a aspectos é composto pelos seguintes componentes:

- **Linguagem de componentes:** deve permitir ao programador implemente as funcionalidades básicas de um sistema, não prevêem nada a respeito do que deve ser implementado na linguagem de aspectos. Exemplos: Java, C++, C#, PHP.
- **Linguagem de aspectos:** deve suportar a implementação das propriedades desejadas de forma clara e concisa, fornecendo construções necessárias para que o desenvolvedor crie estruturas que descrevam o comportamento dos aspectos e definam em que situações eles ocorrem.
- **Combinador de aspectos:** o combinador de aspectos (*aspect weaver*) tem a função de os programas escritos em linguagens de componentes com os escritos em linguagem de aspectos, como por exemplo, a linguagem Java como linguagem de componentes e o *AspectJ* como linguagem de aspectos.
- **Programas escritos em linguagem de componentes:** os componentes podem ser considerados as unidades funcionais do sistema. Num sistema de controle de banco, os componentes são as contas, clientes, funcionários. E sendo programação orientada a aspectos, os componentes são abstrações providas pela linguagem, que permite a implementação da funcionalidade do sistema.

2.2.3. Conceitos básicos de orientação a aspectos

A POA possui quatro conceitos básicos, abordados a seguir.

2.2.3.1. Join Points (Pontos de junção)

Pontos de junção são locais de execução do programa, como por exemplo, uma chamada de método, ou execução de método. É aplicado em métodos (chamadas ou execução), construtor (chamadas ou execução de métodos), execução de tratamento de exceções e atributos (consulta, modifica valores) (CHAVEZ & GARCIA, 2003).

De acordo com Winck & Goetten Junior (2006; p. 47), a partir dos pontos de junção, são criadas regras que dão origem aos pontos de atuação.

Todos os *join points* possuem um contexto associado, por exemplo, a chamada de método possui um método chamador – o objeto alvo, os argumentos do método disponível no contexto. A Tabela 1 demonstra alguns pontos de junção e seus contextos.

Tabela 1 – Exemplos de postos de junção e seus contextos (FONTE: CHAVEZ & GARCIA, 2003).

Ponto de junção	**Objeto "corrente"** "this"	**Objeto "alvo"** "target"	**Argumentos** "args"
Method Call	executing object	target object	method args
Method Execution	executing object	executing object	method args
Constructor Call	executing object	-----	constructor args
Constructor Execution	executing object	executing object	constructor args
Field reference	executing object	target object	------
Field assignment	executing object	target object	assigned value
Handler execution	executing object	executing object	caught exception

2.2.3.2. Pointcuts (Pontos de atuação)

Os *pointcuts* são responsáveis por selecionar os *join points*, devem detectar em quais *join points* o aspecto deve interceptar.

Têm como objetivo, a criação de regras gerais para definição de eventos que serão os pontos de junção, os pontos de atuação devem apresentar dados da execução relativo a cada ponto de junção, que serão utilizados pela rotina disparada pela ocorrência do ponto de junção mapeado no ponto de atuação (WINCK & GOETTEN JUNIOR, 2006).

Após a identificação do interesse é necessário agrupá-lo em um ponto de atuação, onde a combinação entre o ponto de junção e o ponto de atuação torna clara a ocorrência de interesses sistêmicos, já que este, estará, por muitas classes no sistema, e estas classes assim, implementarão mais de um interesse.

2.2.3.3. Advice (Adendo)

Segundo Chavez e Garcia (2003), *Advice* é um trecho de código a ser executado quando um *pointcut* for atingido.

Os *advices* são compostos pelos pontos de atuação, que apenas selecionam os *join points*, para implementar o código é necessário o *advice*, que será executado quando o ponto de atuação definir as regras do ponto de junção.

Existem três tipos de *advices:*

- *Before*: executado antes de o join point ser executado. Exemplo:

```
before(): move() {
System.out.println("Uma linha será movida.");
}
```

Quadro 1 - Exemplo de before (FONTE: CHAVEZ & GARCIA, 2003).

- *After*: executado depois do join point; exemplo:

```
after() returning: move() {
        System.out.println("Uma linha foi movida.");
}
```

Quadro 2 - Exemplo de after (FONTE: CHAVEZ & GARCIA, 2003).

- *Around*: neste caso o adendo escolhe quando vai executar, podendo ser antes ou depois;

```
around(): move() {
        System.out.println("permiteMovimento = " + enableMove);
        if (enableMove) { proceed(); }
        System.out.println("around advice concluido");
}
```

Quadro 3 - Exemplo de around (FONTE: CHAVEZ & GARCIA, 2003).

2.2.3.4. *Aspects* – Aspectos

Os aspectos encapsulam os *join points*, os *point cuts*, os *advices*, em uma só unidade, são definidos de forma semelhante à classe. Propriedades como sincronização, interação entre componente, distribuição e persistência, ficam espalhados por diversos componentes do sistema, os aspectos agrupam esses componentes em uma unidade no sistema.

Na programação orientada a aspectos, os interesses são agrupados em aspectos, evitando que o código fique espalhado e emaranhado, como acontece se, por exemplo, em um sistema orientado a objetos, conforme mostra a Figura 4.

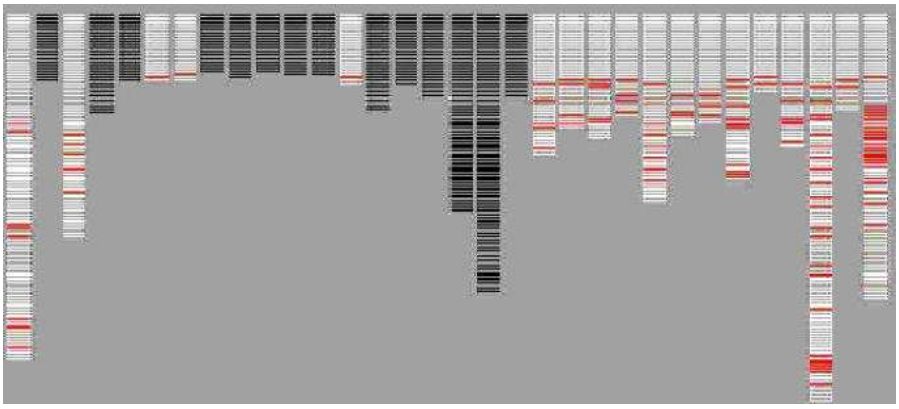

Figura 4 – Código espalhado pelo programa. (FONTE: KICZALES, 2001 *apud* MONTEIRO, 2004).

Deste modo o aspecto não pode existir isoladamente para implementação dos interesses do sistema (funcionais ou sistêmicos), os objetos continuam existindo, e neles são implementados os interesses funcionais, já nos aspectos são implementados os interesses sistêmicos, como por exemplo, auditoria (log), tratamento de exceções, persistência, distribuição, dentre outros. Pode-se definir então, que a unidade principal da POA é o par aspecto-objeto (WINCK & GOETTEN JUNIOR, 2006).

A finalidade da POA não é substituir as técnicas orientadas a objetos, mas complementá-las para possibilitar a separação dos interesses multidimensionais em uma unidade - o aspecto. O aspecto permite que mudanças, quando necessárias, sejam definidas no código-fonte de um sistema de forma menos invasiva (seja em sua estrutura ou em seu comportamento). Assim, é possível alcançar a modularização, beneficiando tanto o desenvolvimento como a manutenção de um

sistema (MONTEIRO, 2004). A Figura 5 demonstra bem isso.

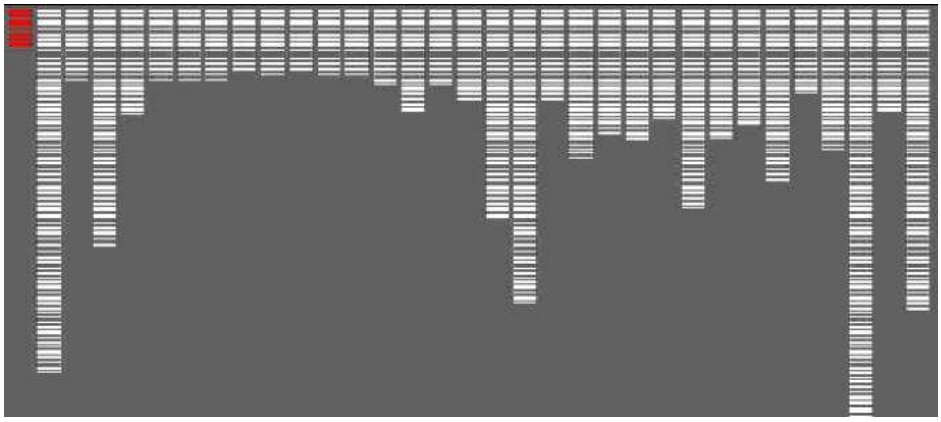

Figura 5 - Representação de um interesse multidimensional concentrado num aspecto (FONTE: KICZALES, 2001 *apud* MONTEIRO, 2004).

2.3. Modelagem orientada a aspectos

A modelagem possui uma importância muito grande durante o desenvolvimento do programa, já que é através dela que fica possível e muito mais clara a visualização dos comportamentos do programa.

Para tornar viável e fácil o entendimento de eventos do mundo real em todas as suas áreas é feita uma abstração onde somente os aspectos relevantes são considerados, assim detalhes que antes eram ignorados anteriormente, agora podem ser vistos e solucionados com mais minuciosidade e cuidado (MONTEIRO, 2007).

Segundo (BOOCH et al, 2000 *apud* Monteiro, 2004) o êxito do desenvolvimento do software, está, em grande parte na contribuição que a modelagem traz, já que esta garante uma comunicação unificada entre os desenvolvedores e evita discrepâncias de informações ou ambigüidades.

Uma modelagem traz bons resultados se alguns itens forem seguidos:

- A escolha do modelo a ser criado, influencia na maneira de como o problema será tratado e como uma solução é definida;
- Os modelos podem ser divulgados em níveis diferentes de precisão;
- Os melhores modelos relacionam-se à realidade, onde o modelo que for tratado isoladamente pode não ter resultado satisfatório, pois

qualquer sistema será melhor analisado por meio de um conjunto de modelos quase independentes(MONTEIRO, 2004);

A maneira de como os modelos para a modelagem são escolhidos, onde que realidade representará e qual nível de abstração irá corresponder a tal realidade, do mesmo jeito que os modelos devem ser definidos cautelosamente, para que se aproximem ao máximo da realidade e que alcancem as soluções já esperadas.

A próxima seção irá apresentar algumas propostas de linguagens de modelagem orientada a aspectos, já que ainda o paradigma não possui uma linguagem específica. Serão apresentadas as linguagens *aSide Language Modeling (asideML)* , *UML-based Aspect Engineering* **(**UAE) linguagens de *early aspect* – fazem parte da fase de análise e as linguagens *Unified Modeling Language All purposes Transformer* (UMLaut) e *oriented - aspect design model* (AODM) que são linguagens utilizadas para a fase de projeto.

2.3.1. asideML

A *asideML* é uma linguagem desenvolvida para especificar e comunicar *designs* orientado a aspectos. Oferece notação, semântica e regras com o principal objetivo de tratar abordar a modelagem conceitual de um sistema em termos de aspectos e [2]*crosscutting* se baseando no modelo de objetos de UML.

De acordo com Chavez, 2004, o termo *aside* (*to or toward the side*), significa elementos que estão do lado dos outros. Também pode ser interpretado como *a side* (um lado), no sentido de um *aspecto*.

2.3.1.1. Objetivos

A *asideML* tem por objetivos, os seguintes itens a seguir:

- Fornecer uma linguagem visual que possa desenvolver e se comunicar com modelos de aspectos;

- Fornecer semântica e notação para ter, assim suporte aos novos requisitos introduzidos pelos aspectos;
- Oferecer suporte às especificações independentes de linguagem que podem ser mapeadas para linguagens de programação orientadas a aspectos particulares;
- Seguir padrões de indústria.

2.3.1.2. Artefatos

A *asideML* proporciona elementos de apresentação para a maioria dos elementos de modelagem, com estes, organizados em diagramas gráficos é possível fornecer várias visões do sistema a ser desenvolvido. A Tabela 2 mostra os modelos de *asideML* e os diagramas que vão exibi-los, e também os novos elementos de modelagem e as perspectivas relevantes para cada diagrama.

Tabela 2 – Dimensões da modelagem orientada a aspectos com *asideML*.(FONTE: Chavez, 2004; p. 138).

Modelo	Diagramas	Perspectivas	Elementos
estrutural	diagrama de aspectos		aspecto, interface transversal, característica transversal
	diagrama de classe estendido	centrado em aspecto	aspecto, *crosscutting*
		centrado na base	interface transversal, *order*
comportamental	diagrama de sequência estendido	ponto de combinação	ponto de combinação dinâmico
	diagrama de colaboração aspectual		instância de aspecto, colaboração aspectual
	diagrama de sequência		instância de aspecto, interação aspectual
processo de combinação	diagrama de classes combinadas		classe combinada
	diagrama de colaboração combinada		colaboração combinada
	diagrama de sequência combinada		interação combinada

[2] *Crosscutting* – é um relacionamento entre um aspecto parametrizado e elemento base.

2.3.1.3. Diagramas

Segundo Chavez, 2004, p. 139, alguns diagramas fornecidos pela linguagem *asideML* são novos e outros complementam a UML a fim de apresentar os elementos crosscutting e seus relacionamentos para os elementos base, a seguir serão apresentados os diagramas relativo a linguagem de modelagem *asideML* :

- **Diagrama de aspecto**

O diagrama de aspecto apresenta descrição completa de um aspecto. A descrição incorpora as interfaces transversais, características locais e relacionamentos de herança.

- **Diagramas de colaboração aspectuais**

É possível a visualização de cada característica transversal comportamental.

Este diagrama oferece apresentação gráfica de uma colaboração aspectual (tipo de colaboração que modela a realização de uma operação transversal definida dentro do aspecto). O diagrama de colaboração aspectual dá suporte à visão de interação que envolve instâncias de aspectos e elementos base.

- **Diagramas de seqüências aspectuais**

Oferece apresentação gráfica de um conjunto de mensagens organizadas em seqüências temporais, que podem ser invocações e operações de aspectos, sob a perspectiva de aspectos. Tem suporte a visão de interação (envolve instâncias de aspectos e elementos base).

- **Diagramas de processo de combinação**

Oferece apresentação gráfica para um grupo de elementos combinados (são elementos que alinhados enfatizam as melhorias proporcionadas pelos elementos crosscutting).

- **Diagramas de classes estendidos**

Oferece apresentação gráfica da visão de projeto estático de um sistema, onde classes e aspectos residem como cidadãos de primeira classe. Cada aspecto pode ser visto separadamente em detalhe em um diagrama de classes correspondente (CHAVEZ, 2004).

2.3.1. 4. Considerações de asideML

Levando-se em conta a rápida evolução da POA, a modelagem orientada a aspectos enfrenta problemas, que pede ainda mais pesquisa sobre o assunto. Assim, como outras linguagens de modelagem aqui não citadas, a *asideML* ainda não conseguiu se habituar muito bem à POA, bem pela rápida evolução em que se encontra a OA; enquanto tem um desempenho muito bom em alguns aspectos, falha em outros.

2.3.2 A notação UAE

Esta seção apresenta a notação UAE (UML – *based Aspect Engeneering*) que português seria UML baseada em engenharia de aspectos, para modelagem de sistemas orientados a aspectos por meio da extensão da UML.

Para utilizar a UAE em programação orientada a aspectos é utilizado uma ferramenta que dê suporte a esta notação, a ferramenta de modelagem [3] MVCASE, que irá servir de apoio no projeto orientado a aspectos.

A UAE utiliza a aplicação AspectJ como padrão , e na Figura 6 é mostrado um novo diagrama sendo criado com a utilização da ferramenta MVCASE.

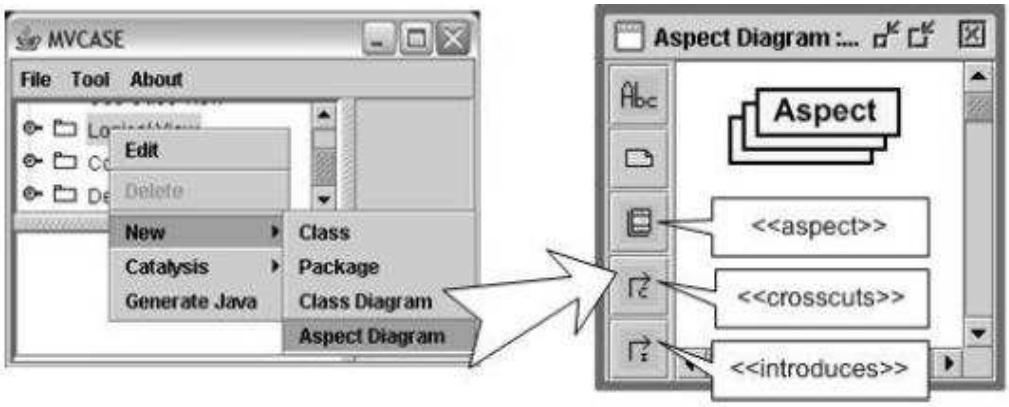

Figura 6 – Extensão de MVCASE para o Projeto Orientado a Aspectos (FONTE: GARCIA, 2004).

[3] Disponível para download em: http: //www.recope.dc.ufscar.br/mvcase/

A principal construção em *AspectJ* é o aspecto. Um aspecto define uma funcionalidade específica que pode afetar diferentes partes de um sistema, como, por exemplo, persistência em banco de dados. Além disso, os aspectos podem alterar a estrutura dinâmica de um sistema por meio dos pontos de combinação (*join points*) ou, ainda, alterar a estrutura estática por meio da declaração de introduções (*introductions*).

Um aspecto normalmente define um ponto de atuação (*pointcut*) que identifica os pontos de combinação e adiciona comportamentos a eles por meio de sugestões (*advices*) (GARCIA, 2004).

A notação UAE busca solucionar problemas como poluição visual do modelo e códigos que, dependendo da utilização podem ter significados diferentes, como é o caso do (**!**), oferecendo uma notação simples, onde a representação de aspectos é diferenciada em relação à de classes, com pouca poluição visual e voltada a aplicação *AspectJ*(PAWLAK et al., 2002 Apud GARCIA, 2004).

2.3.3. UMLaut

A *Unified Modeling Language All purposes Transformer* – UMLaut é uma ferramenta para manipulação de modelos UML – incluindo diagramas de seqüência (CARTAXO, 2006).

Do ponto de vista do *Aspect Oriented Software Development*(ASOD), a UMLaut pode ser vista como um framework que oferece um kit de combinadores específicos para cada aplicação para construção de modelos de design detalhados a partir de modelos UML orientados a aspectos e de alto nível(CHAVEZ,2004; p119) – (funcionais, dinâmicos, e aspectos estáticos anotados com ocorrência no padrão de projeto, estereótipos e valores rotulados) detalhados adequadamente para modelos de design tanto para implementação, simulação ou validação. A UMLaut é desenvolvida com o Projeto Triskell e assim que estiver completa será distribuída gratuitamente. A UMLaut é o único projeto relatado pela Iniciativa de Componentes Confiáveis (IRISA Org. 2005).

Os estereótipos são usados para criar um subtipo de um determinado tipo de elemento de modelo, por exemplo, para especificar que instâncias de uma classe devem ser persistentes. As ocorrências de padrão de projeto são usadas para especificar que um determinado padrão de projeto deve ser aplicado em um determinado lugar no modelo. Os valores rotulados fornecem ao combinador outras informações para orientar o processo combinação (CHAVEZ, 2004).

O framework lida com o processo de combinação de designs utilizando UMLaut o procedimento de combinação é implementado como um processo de combinação de modelo: cada etapa do processo de combinação é uma etapa transformacional aplicada ao modelo UML, em que o resultado não também é um modelo de UML.

A abordagem UMLaut da suporte a combinação, no entanto não suporta todos os requisitos de UML.

2.3.4. AODM

A linguagem de modelagem AODM (*oriented - aspect design model*) elaborada por STEIN, 2002, introduz notações não padronizadas pela UML para representar alguns conceitos da POA (PENCZEK, 2006). É definido como extensão UML e prevê mecanismos de combinação de modelos para a geração de diagramas UML tradicionais. Seu objetivo é levar a orientação a aspectos para a fase de projeto, reproduzindo também o processo de combinação de *AspectJ* na UML(SOUZA, 2007).

Em AODM utiliza-se como padrão a ferramenta *AspectJ* fazendo uma comparação minunciosa entre *AspectJ* e UML. Nesta comparação constatou que *links*, na especificação UML, são adequados para representar os pontos aonde os aspectos atuarão (*join points* do *AspectJ*). Tanto *pointcuts* e *advices* podem ser representados como operações estereotipadas, devido à grande semelhança estrutural destas construções em relação às operações padrões da UML.

Aspectos são semelhantes a classes, portanto são representados como classes estereotipadas. Segundo Caldas & Rodrigues (2006); p.38, os aspectos só são diferentes das classes em seus mecanismos de herança, pois os aspectos filhos herdam todas as características do aspecto pai, porém somente as operações abstratas de Java e os *pointcuts* podem ser cancelados.

Os atributos funcionam normalmente, já que um aspecto em *AspectJ* tem a possibilidade de suportar atributos (STEINMACHER; GIMENES; LIMA, 2007). Os aspectos também podem acoplar os mesmos relacionamentos de associação e da generalização como as classes (CALDAS & RODRIGUES, 2006). Os relacionamentos entre classes e aspectos são representados por um novo estereótipo para relacionamento chamado *crosscut*, a qual estará dizendo que o aspecto está tomando conta do lugar onde ele está relacionado.

2.3.5.1. Diagramas

Segundo Stein apud Chavez, 2004 p. 122, nos diagramas, os estereótipos AODM definidos são interpretados da mesma maneira que em suas metaclasses correspondentes. Alinhados com o nome do estereótipo acima do nome do elemento.

Os aspectos são representados por um retângulo, tendo nome, atributos e operações. *Pointcuts* e *advices* são representados como *strigs* que especificam suas assinaturas, aparecem no compartimento de operações. O comportamento *advice* é descrito usando diagramas de seqüência UML.

STEIN utiliza um exemplo de um framework de um padrão de objeto *Observer* , que segundo Chavez, 2004; p.120, realiza um rótulo colorido(*color label*), ficando de observador, e quando o botão (*button*) – que exerce papel de sujeito, é clicado o padrão *observer* muda a cor. O quadro abaixo apresenta um aspecto abstrato, **SubjectObserverProtocol** o qual implementa o padrão *Observer* para **Button** e **ColorLabel** .

```
aspect SubjectObserverProtocolImpl
     extends SubjectObserverProtocol {

   /* concrete pointcut declaration */
   pointcut stateChanges(Subject s):
       target(s) && call(void Subject.click());

   /* introduction # Subject: Button # */
   declare parents: Button implements Subject;

   public Object Button.getData() { return this; }

   /* introduction # Observer: ColorLabel # */
   declare parents: ColorLabel implements Observer;

   public void ColorLabel.update() { colorCycle(); }
}
```

Quadro 4 – Implementação Aspecto Abstrato (FONTE: CHAVEZ 2004; p 121).

É implementado também um aspecto concreto de nome **SubjectObserverProtocolImpl** que implementa o padrão de projeto *Observer*. O qual é apresentado pelo Quadro 5.

```
abstract aspect SubjectObserverProtocol {

   abstract pointcut stateChanges(Subject s); // pointcut

   after(Subject s): stateChanges(s) {  // after advice
      for (int i = 0; i < s.getObservers().size(); i++) {
        ((Observer)s.getObservers().elementAt(i)).update();}
   }

   // inter-type declarations - Subject role
   private Vector Subject.observers = new Vector();
   public  void   Subject.addObserver(Observer obs) {
     observers.addElement(obs);
     obs.setSubject(this);
   }
   public  void   Subject.removeObserver(Observer obs) {
     observers.removeElement(obs);
     obs.setSubject(null);
   }
   public Vector Subject.getObservers() {return observers;}

   // inter-type declarations - Observer role
   private Subject Observer.subject = null;
   public void Observer.setSubject(Subject s) {subject = s;}
   public Subject Observer.getSubject() { return subject; }
}
```

Quadro 5 – Implementação Aspecto concreto SubjectObserverProtocolImpl (FONTE: CHAVEZ, 2004; p.121).

A Figura 7 apresenta o código do padrão de projeto *Observer* implementado através da modelagem orientada a aspectos proposta por STEIN, a AODM. Onde o aspecto abstrato **SubjectObserverProtocol** define um *pointcut* abstrato **stateChanges**, um *after advice* **advice_id01**, e duas introduções **Subject** e **Observer**. Além do aspecto, suas outras propriedades especificam que o aspecto deve ser instanciado uma vez para o ambiente global (uma vez por Java Virtual Machine). Especificação somente útil em implementações de AspectJ. O aspecto **SubjectObserverProtocol** é estendido por um aspecto concreto chamado **SubjectObserverProtocolImpl**, que atribui o *pointcut* **stateChanges** a um conjunto concreto de *join points* . Além disso, o aspecto concreto contém duas outras introduções chamadas *Button* e *ColorLabel* (CHAVEZ, 2004).

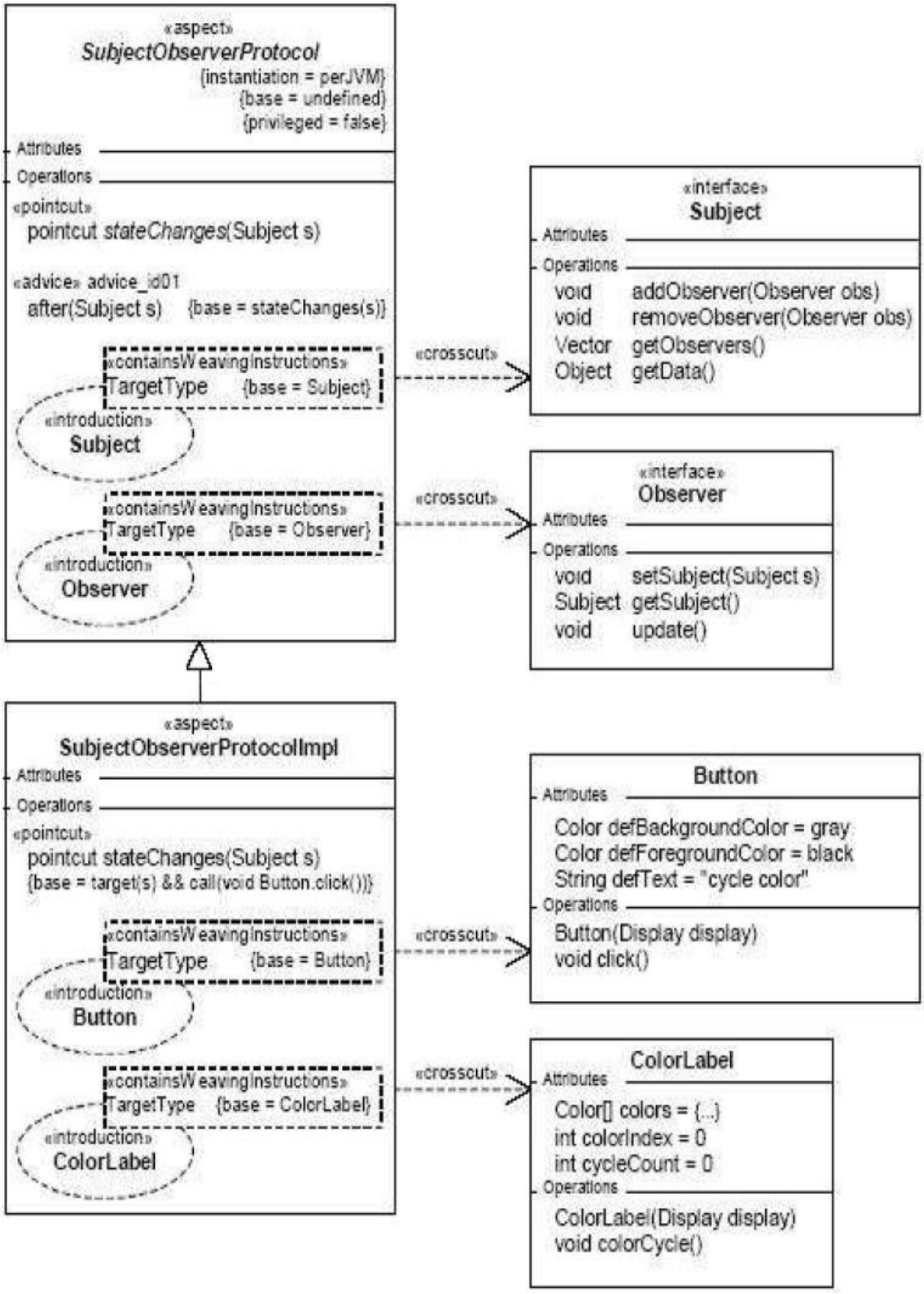

Figura 7 – Representação AODM (FONTE: CHAVEZ, 2004 p.123)

Em AODM os *advices* são representados por colaboração, como mostra a Figura 8, apontando a especificação *after advice* o qual está contido no aspecto **SubjectObserverProtocol.**

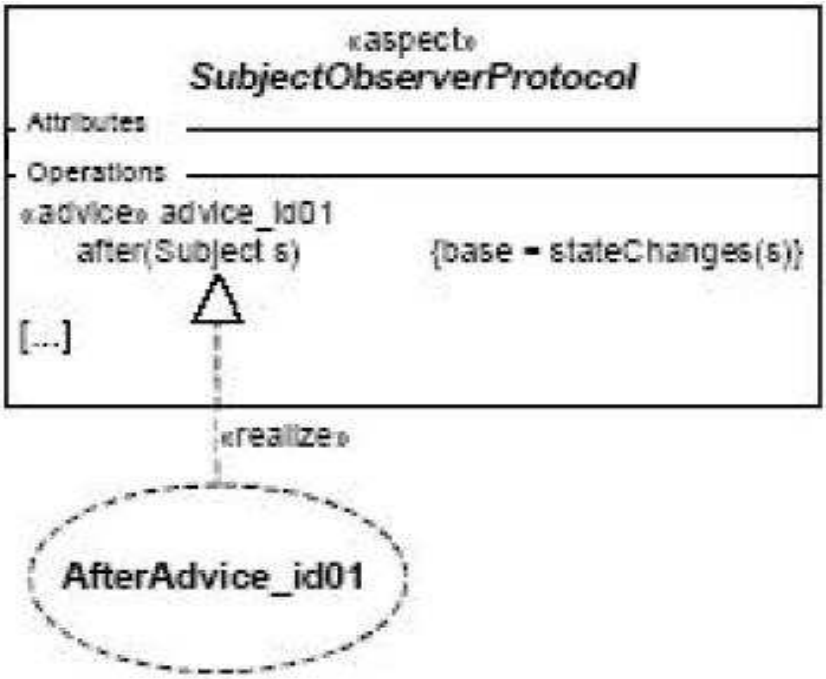

Figura 8 – Ilustração de *Advice* com AODM (FONTE: CHAVEZ, 2004; p. 124).

A colaboração realizada pelo *after advice* é mostrada pela Figura 9.

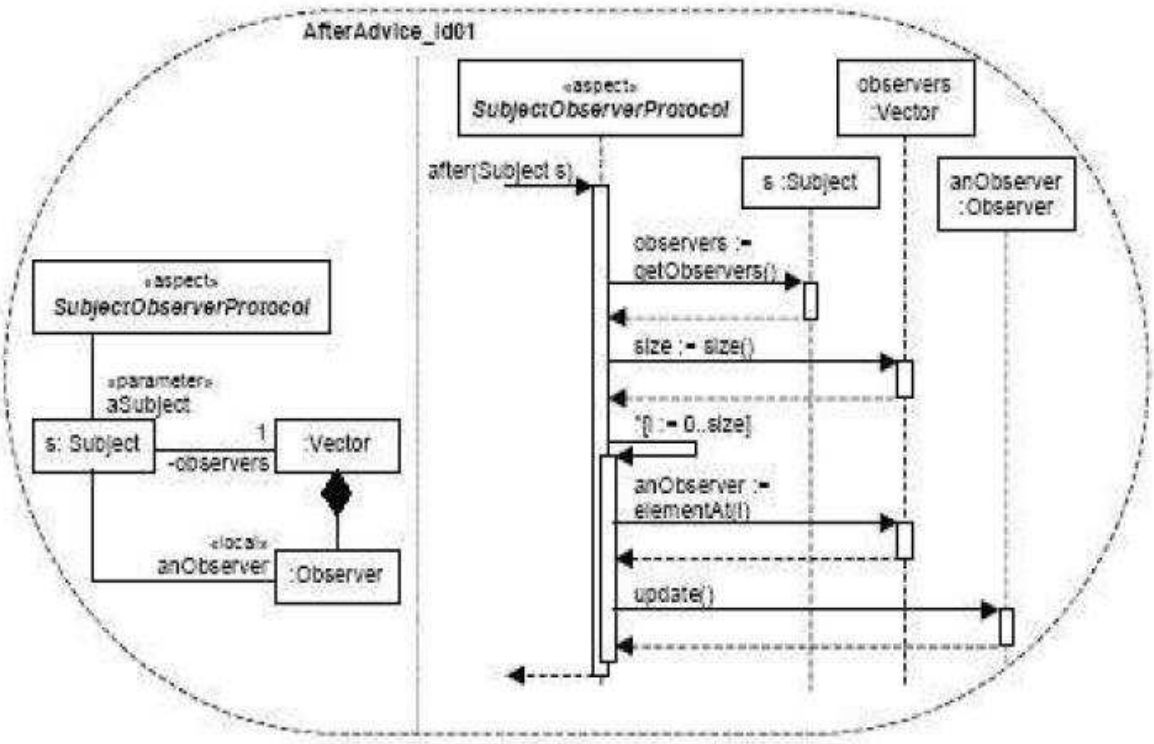

Figura 9 – AODM, Colaboração para *after advice* (FONTE CHAVEZ, 2004; p.125).

A AODM é uma abordagem de grande importância e bem estabelecida para o começo da padronização de sistemas de software baseados em aspectos, pois possibilita que as vantagens da modularização de sistemas sejam trazidas para a fase de projeto (CHAVEZ, 2004, apud Stein, 2002).

3. ANÁLISE DE LINGUAGENS

Para análise das linguagens de modelagens foram utilizadas as seguintes abordagens: *aSideML*, a Notação UAE, que são linguagens de *early aspects* – que são a linguagem referente a fase de análise , e para a realização do estudo de caso foram utilizadas as linguagens *Unified Modeling Language All purposes Transformer* (*UMLaut)* e *Aspect-Oriented design model* (AODM), que são linguagens de fase de projeto.

Durante a pesquisa foram identificadas algumas falhas nas abordagens acima citada, então foram escolhidas duas linguagens como temas de pesquisa mais a fundo, a AODM e a *UMLaut*, que se apresentaram mais eficazes que as outras.

Apesar de ser muito usada durante os anos de 2004 e 2005, a *aSideML* não apresenta alguns requisitos de extrema importância na orientação a aspectos, como a inexistência de herança entre aspectos, segundo a própria autora, Chavez (2004) , é oferecida uma proposta, que seria desenvolvida somente em trabalhos futuros, mas no entanto ainda não foi desenvolvida. A proposta traria os *subaspectos* herdando as propriedades dos *superaspectos*, assim como na herança de classes. A ausência da herança foi e é justificada por nenhuma abordagem de implementação, exceto *AspectJ*, oferecer tal suporte (GARCIA et al., 2007).

A *aSideML* também não oferece declaração explícita de conjuntos de junção e não representa todos os conceitos de POA(CHAVEZ, 2004).

Já na Notação UAE, as dificuldades foram além de tudo, no material, pois só foi encontrada uma referência em relação a esta abordagem, portanto, fica muito difícil fazer uma análise mais a fundo de suas propriedades de linguagem, mas no pouco encontrado, foi diagnosticado que esta notação não oferece técnicas que

suportam o *weaving* (combinação em tempo de compilação), a ferramenta utilizada é o MVCASE – o qual possui extensão a UML conforme Garcia(2004), por fim, não foi possível dar continuidade a pesquisa desta abordagem por insuficiência de material suficiente para um trabalho de pesquisa científica.

A tabela abaixo faz uma comparação entre a abordagem AODM e a UMLaut.

Tabela 3 – Comparação entre AODM e UMLaut.(FONTE: Chavez, 2004; p. 133).

	AODM	**UMLaut**
Aspectos são de primeira classe	Aspectos são classes estereotipadas	Não
Interações explícitas entre aspectos	Não	Não
Extensões sensíveis ao contexto	Sim	Não
Semântica para *crosscutting*	AspectJ	Espalhada em regras de transformação
Quantificação	Sim	Não
Visão do sistema combinado	Modelos em UML	Modelos em UML
Dependência de linguagem	AspectJ	Customizável
Ferramenta	Não	Ferramenta de Transformação
Extensão de UML	Light-Weight	Não

4. ESTUDO DE CASO

Para a realização do estudo de caso foi escolhido um sistema de reserva de laboratórios de informática, que tem acesso via *Bluetooth* e através da internet.

A reengenharia com a utilização da AODM será realizada no desktop. Atuarão nas partes de dados e negócio, onde os aspectos são necessários.

No estudo de caso só será possível visualizar aspectos, *join points, poincuts,* e *crosscuts*, pois o sistema só foi implementado em OO, ficando, assim ocultos algumas operações do paradigma, tais como, *targets* e *adivices*. A figura 10 mostra o diagrama de classes no qual é possível ver aonde os aspectos atuarão.

Como já pôde ser visto durante este trabalho, os aspectos são utilizados para resolver vários problemas que com só a orientação a objetos não era possível. Estes aspectos são organizados nas seguintes categorias apontadas a seguir: *aspectos sistêmicos*, *aspectos colaborativos*, *aspectos subjetivos* e *aspectos evolutivos*. Cada tipo de aspecto traz desafios diferentes para serem solucionados pela MOA, neste trabalho serão tratados somente os aspectos sistêmicos e colaborativos, que são de maior importância:

- **Aspectos Sistêmicos –** são usados para tratar da modularização de interesses de requisitos não-funcionais, tais como persistência, tratamento de erros, mecanismos de auditoria, entre outros (CHAVEZ, 2004). Estes afetam várias classes de forma homogênea, por isso é necessário mecanismos eficientes de quantificação, envolvendo o uso e expressões regulares com o objetivo de facilitar a especificação de pontos de combinação de interesse.
- **Aspectos colaborativos** - Usados para cuidar da necessidade de localizar a interação do componente. Os aspectos colaborativos se parecem muito com as colaborações UML, portanto devem ter interfaces bem claras. Atuam em tempo de compilação.
- **Aspectos subjetivos** - Podem ser usados para cuidar da necessidade de visões segregadas para objetos, introduzindo novos atributos e métodos em um Elemento Base ou restringindo o uso de outros.

- **Aspectos Evolutivos** - Podem ser usados para resolver a necessidade de evolução das classes, ao modificar sua Superclasse, implementar uma interface ou introduzir novos atributos e métodos (CHAVEZ, 2004).

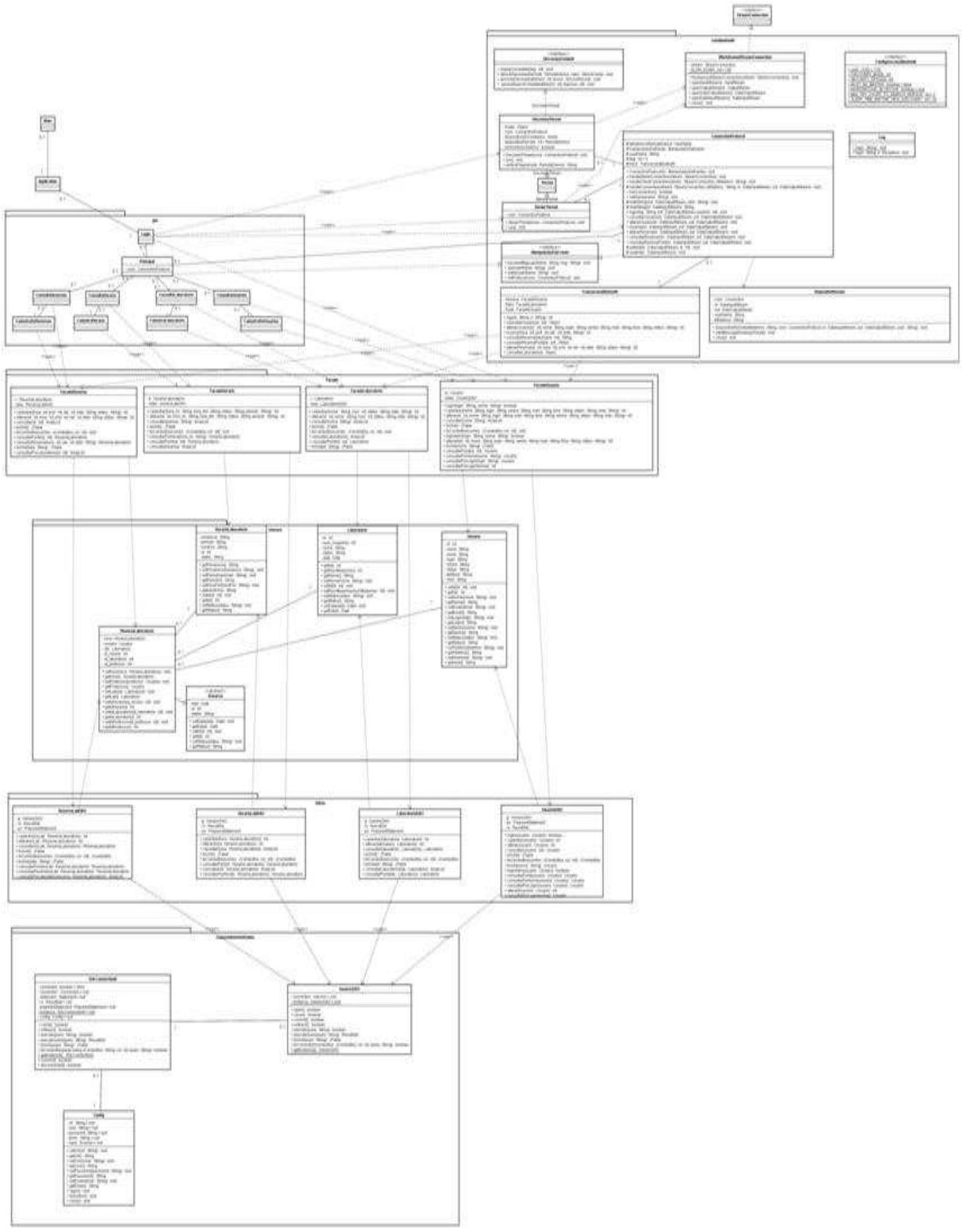

Figura 10 – Diagrama de Classes Sistema de Reserva de Laboratórios – Desktop (FONTE: OLIVEIRA, 2007)

Os aspectos, tanto os sistêmicos, quanto os colaborativos vão atuar nas áreas de auditoria, gerenciamento de transações, tratamento de exceções e restrições arquiteturais, todos apresentados na próxima seção.

4.1. Auditoria

A implementação de sistemas orientado a objetos é feito em entidades únicas, como no caso de *log* de dados, em que a implementação é feita somente em uma classe, sendo referenciada onde há a necessidade da realização de *log*, e como a maioria dos métodos tem necessidade que alguns dados sejam registrados em *log*, as chamadas de método ficam espalhadas por todo o código.

Com a OA os interesses de auditoria podem ser implementados independente da regra de negócio. Dessa forma, o código estaria em um único local centralizado e bem definido no sistema, sendo retirado das classes.

O diagrama abaixo mostra a auditoria do sistema feita em AODM, que age principalmente no pacote do *facade*, que encapsula o acesso a dados e fornece uma simples *interface* de acesso a essas funcionalidades aos componentes de apresentação; onde todo os interesses transversais estão encapsulados em um aspecto, AuditorAspect.

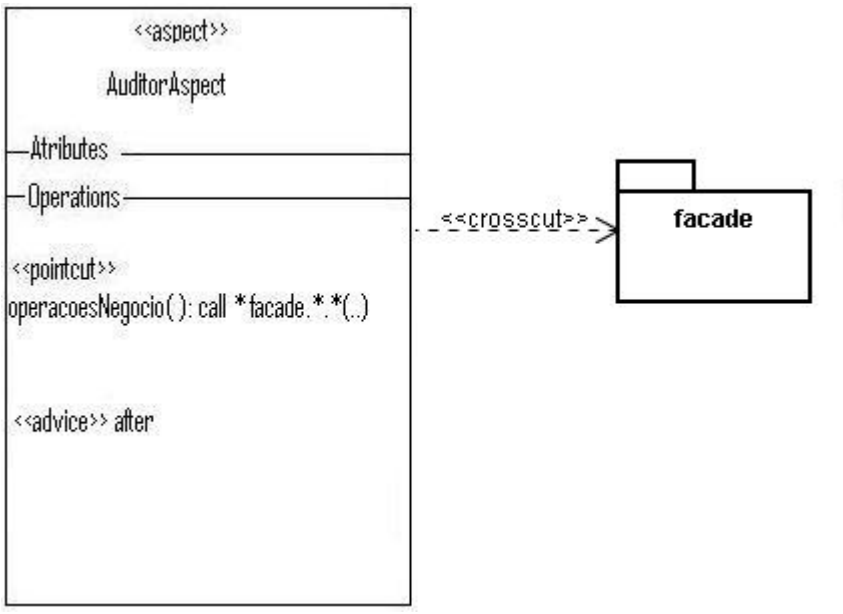

Figura 11 – Diagrama de auditoria em AODM.

4.2. Gerenciamento de transações

Para garantir que os dados armazenados de um sistema sejam persistentes, é necessário o uso das propriedades ACID – atomicidade das operações, consistência dos dados, isolamento quando se executa operações e durabilidade em todos os casos, mesmo em situações que o sistema falhar (ELMASRI & NAVATHE 1994 apud OLIVERIA; MENOLLI; COELHO, 2007).

Os interesses transversais relativo ao gerenciamento de transações do sistema de reservas de laboratórios pertencem ao pacote DAO, aonde o aspecto atuará. O processo de *refactoring* removeu esse código e implementou o interesse de gerenciamento de transações em um aspecto, como mostra o quadro abaixo.

```
01 package transacoes;
02
03 import conexao.SQLConnection;
04 import conexao.GenericDAO;
05
06 public aspect BDTransactionAspect {
07
08    private static SQLConnection c=SQLConnection.getInstancia();
09    pointcut transacoes(): call( * SQLConnection.execute*(..) ) ||
10    call(public * GenericDAO.to.*(..))|| call(
11    java.sql.PreparedStatementSQLConnection.
12    setPreparedStatement(String));
13
14    before(): transacoes(){
15       c.getConnection();
16    }
17
18    after() returning: transacoes(){
19       c.commit();
20    }
21
22    after() throwing(): transacoes(){
23       c.rollback();
24       c.close();
25    }
26 }
```

Quadro 6 – Implementação Aspecto gerenciamento de transações (FONTE: OLIVERIA; MENOLLI; COELHO, 2007) .

A implementação orientada a aspectos desse interesse declara um atributo, que é usado para obter uma instância válida do mecanismo persistente utilizado. Isso é necessário para invocar os serviços transacionais suportados pelo mecanismo persistente. Além disso, essa implementação também define o *pointcut* transações, que é utilizado para identificar os métodos em que a execução é limitada por uma

transação. E por fim, o aspecto mostrado anteriormente implementa três *advices* para iniciar, terminar com sucesso e abortar transações. O primeiro é um *advice* do tipo *before*, que inicia uma transação pouco antes da execução de qualquer método transacional.

O segundo *advice*, confirma a transação e é executado quando a execução de alguma operação transacional retornou com sucesso. O último *advice* executa se qualquer problema ocorrer durante a execução de qualquer método transacional, executando o método roolback, que retorna a transação ao estado original, mantendo o banco de dados em um estado consistente e liberando recursos do mecanismo persistente da aplicação (OLIVERIA; MENOLLI; COELHO, 2007) .

Para a modelagem de tal código de gerenciamento de transações foi feita a modelagem com do aspecto ManegementTransactionAspect, em AODM, como é ilustrado na próxima figura, onde o aspecto atua sobre os *join points* do pacote DAO.

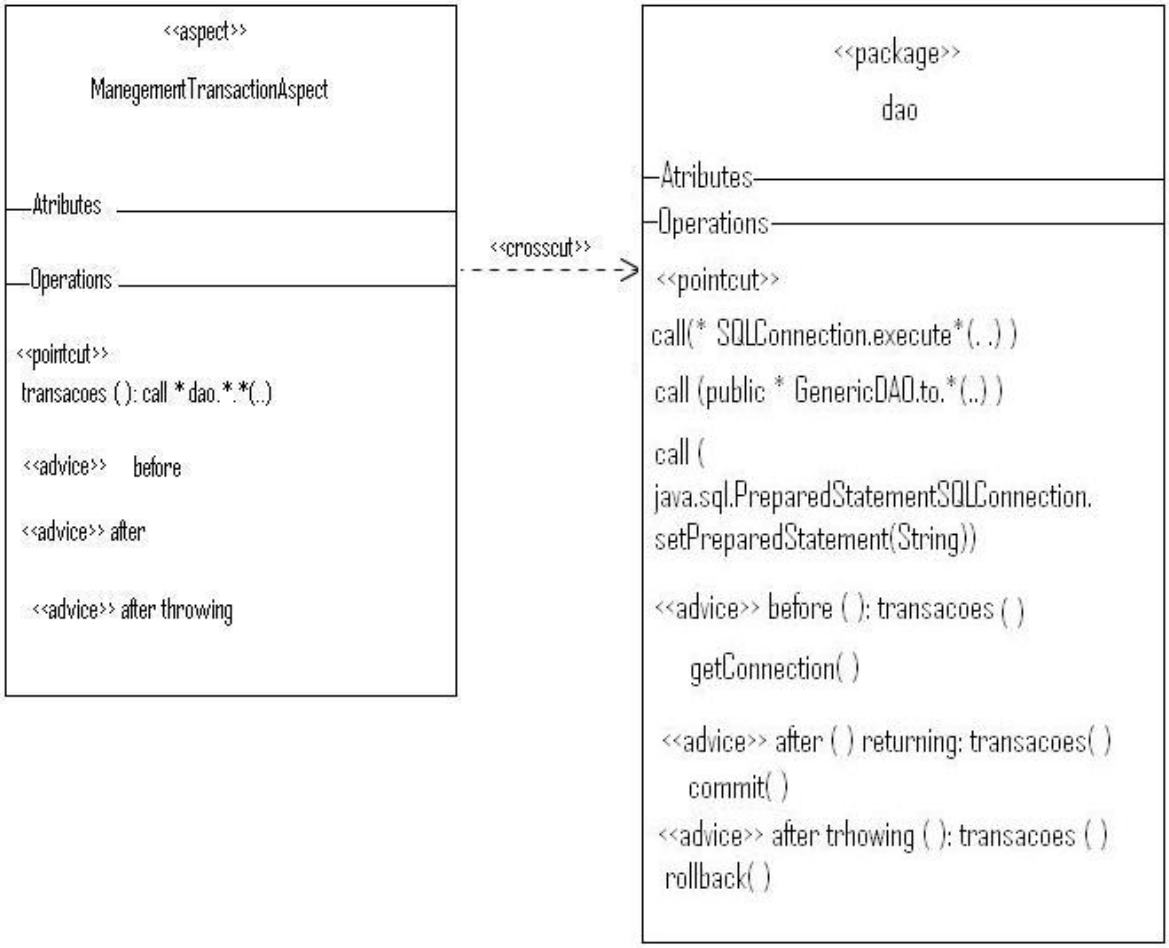

Figura 12 – Gerenciamento de Transações do estudo de caso com AODM.

4.3. Tratamento de exceção

O tratamento de exceção é responsável pela manipulação de erros encontrados durante a execução do sistema. Esses erros podem ocorrer em situações de manipulação de algum recurso do sistema, como acesso a banco de dados ou outro mecanismo de armazenamento, restrições de regras de negócio, como acesso a valores fora do limite de um *array*, divisão por zero e parâmetros inválidos (DEITHEL & DEITHEL, 2005 apud OLIVERIA; MENOLLI; COELHO, 2007). Normalmente o código de manipulação de exceções fica espalhado por toda a aplicação, prejudicando legibilidade e modularidade.

Segundo Oliveira; Menolli; Coelho, 2007; p. 39, alguns *advices* elevam exceções que não são manipuladas por eles. Para tanto foram implementados aspectos que manipulam estas exceções. No estudo de caso estes aspectos manipulam exceções não checadas, desde que elas sejam elevadas pelos *advices* que implementam os interesses de persistência da aplicação.

Existem dois tipos de *advices* que são usados pra o tratamento de exceções, *after throwing* e *around*. O *advice after throwing* permite definir um tratamento(*handler*) para tratar a exceção, mas não impede a propagação da exceção, permite também a conversão da exceção capturada em uma outra. Já o *advice around* permite o completo tratamento da exceção.

No sistema do estudo de caso foi implementado um aspecto PersistenceHandlingException, como pode ser visto no quadro a seguir. O aspecto declara um *pointcut*, para identificar *join points* em que exceções devem ser manipuladas. Também há neste aspecto um bloco de declaração de uma construção *soft*, que determina o tipo de exceção e os *join points* nos quais essas exceções serão manipuladas. O aspecto também define um *advice after throwing*, que é executado quando uma exceção é lançada em específicos *join points.*

```
01 package excecoes;
02
03 import java.sql.*;
04 import conexao.*;
05
06 public aspect PersistenceHandlingExceptionAspect {
07
08    pointcut operacoes(): execution(* SQLConnection.*(..))
09    && !withincode(boolean SQLConnection.getConnection(..));
10
11
12    declare soft:SQLException: operacoes();
13
14    after() throwing(SQLException e) throws SQLException:
15    operacoes() {
16          throw e=new SQLException(e);
17    }
18 }
```

Quadro 7 - Implementação do aspecto PersistenceHandlingExceptionAspect (FONTE: OLIVERIA; MENOLLI; COELHO, 2007) .

Na MOO (Modelagem Orientada a Objetos), os blocos de tratamento de exceção estão espalhados por toda a aplicação, prejudicando a legibilidade e manutenibilidade da aplicação, uma vez que isso provoca um entrelaçamento de código de tratamento de exceção em toda a aplicação, além da repetição de vários trechos de código que implementam esses por vários módulos do sistema.

Com a finalidade de resolver esse problema, pode-se modularizar esse interesse utilizando aspectos, como é ilustrado na figura a seguir (LADDAD, 2003 apud OLIVEIRA; MENOLLI; COELHO 2007).

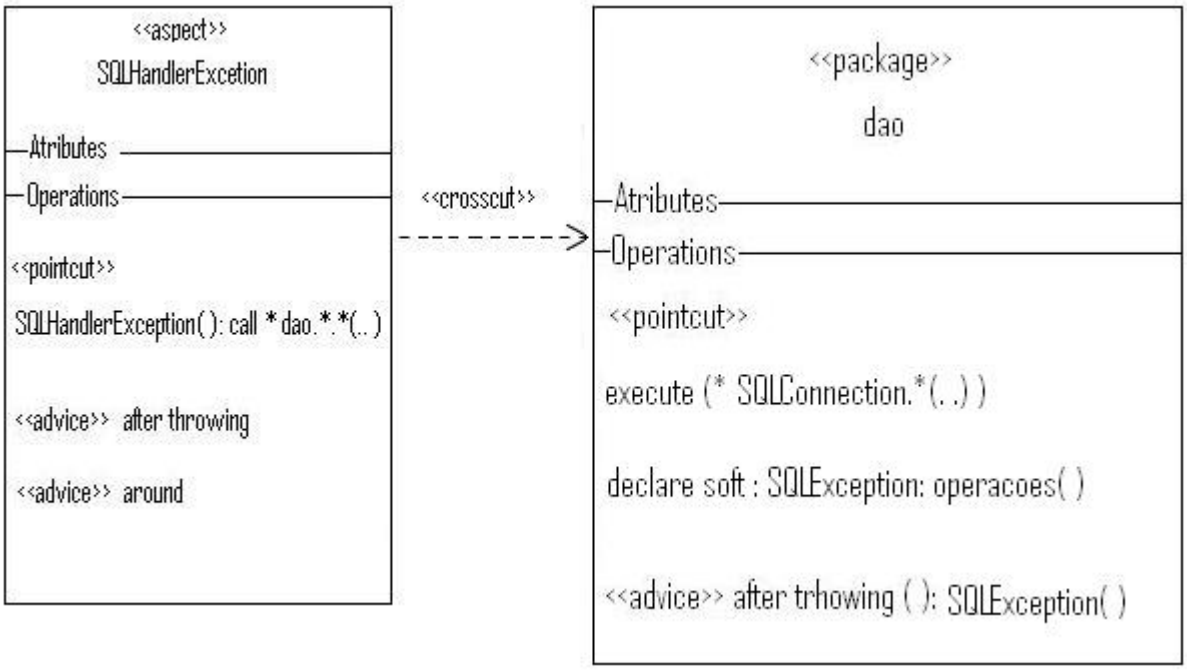

Figura 13 – Tratamento de Exceção do estudo de caso com AODM.

Modelando com OA os interesses transversais do pacote DAO são todos encapsulados dentro do próprio aspecto, deixando assim o sistema mais limpo e legível. O tratamento de exceção também foi realizado no pacote de conexaoBD, onde foi obtido os mesmos resultados – todas as exceções ficam encapsuladas dentro do aspecto.

4.4. Restrições arquiteturais

As restrições arquiteturais têm como objetivo garantir que regras de projeto ou políticas e implementação serão seguidas pelo sistema.

A arquitetura da aplicação impõe algumas restrições de acesso entre as camadas existentes, que são: *gui*, *facade*, *Daos*, *beans* e *conexaoBD*. Onde os componentes da camada *gui* só podem acessar os componentes da camada *facade* e *beans*, já a camada *facade* pode acessar somente os componentes das camadas *beans* e *Daos*, e os componentes da camada *Daos*, tem permissão para acessar os componentes da camada *beans* e conexaoBD. A verificação de regras e políticas em AspectJ é feita em tempo de compilação utilizando as declarações *declare warning* e *declare error,* no exemplo será mostrada essa declaração(OLIVERIA; MENOLLI;

COELHO, 2007) .

Para fazer estas regras em OO será necessário a construção de rotinas extensas e complexas, que consomem tempo e são mais tendentes a falhas, para tanto este interesse transversal identificado durante o processo de *refacturing* do sistema, foi implementado de forma simples e que possa ser reutilizável, através dos aspectos RestricaoArquiteturaAspect e RegrasArquiteturaAspect. Com aponta o quadro oito e nove, descritos a seguir.

```
01 package arquitetura;
02
03 public aspect RestricaoArquiteturaAspect {
04
05    pointcut camadaGUI():within( gui..*);
06    pointcut camadaConexao(): within(conexao..*);
07    pointcut camadaDao(): within(dao..*);
08    pointcut camadaNegocio():within(beans..*);
09    pointcut camadaFacade(): within(facade..*);
10
11    pointcut chamadaCamadaGUI(): call(* gui..*(..))
12      && !camadaGUI();
13
14    pointcut chamadaCamadaConexao():call(* conexao..*(..))
15      && !camadaConexao();
16
17    pointcut chamadaCamadaDao(): call(* dao..*(..))
18      && !camadaDao();
19
20    pointcut chamadaCamadaFacade(): call(* facade..*(..))
21      && !camadaFacade();
22
23    pointcut chamadaCamadaNegocio(): call(* beans..*(..))
24      && !camadaNegocio();
25  }
```

Quadro 8 - Implementação do aspecto RestriçãoArquiteturaAspect (FONTE: OLIVERIA; MENOLLI; COELHO, 2007) .

Onde o aspecto RestricaoArquiteturaAspect, define *pointcuts* para capturar *join points* que partem de cada camada do sistema de reservas. Essas chamadas são capturadas pela construção *within*, que se encontra dentro da definição de cada *pointcut*. Este aspecto também define *pointcuts* auxiliares que capturam chamadas a *join points* que não partem da camada de origem deste *join point*, ou seja, cada um desses *pointcuts* captura *join points* que partem de camadas externas a origem do *join point* capturado. O *pointcut* chamadaCamadaGUI, é um desses *pointcuts* auxiliares que captura chamadas a *join points* de componentes da camada gui, que partem de componentes não pertencentes a esta camada.

O próximo aspecto, RegrasArquiteturaAspect, apresentado no Quadro 7, é responsável por implementar as regras de restrições arquiteturais estabelecidas pela arquitetura da aplicação. Segundo Oliveria, Menolli e Coelho (2007, p.38), foram implementadas apenas duas regras arquiteturais. A primeira regra emite um *warning* ao desenvolvedor toda vez que se tenta acessar métodos de componentes da camada gui de outras camadas da aplicação. Já a segunda regra estabelece um *warning* quando os componentes da camada gui tentam acessar métodos presentes nos componentes das camadas daos e conexaoBD (OLIVERIA; MENOLLI; COELHO, 2007).

```
01 package arquitetura;
02
03 public aspect RegrasArquiteturaAspect {
04
05    declare warning: RestricaoArquitetura.chamadaCamadaGUI()
06      && ! withincode(* Application..*(..)):
07      "Chamadas a comoponentes da camada GUI podem ser feitas
08      somente pela própria camada GUI e pelo componente
09      Application";
10
11    declare warning: (RestricaoArquitetura.chamadaCamadaConexao()
12      || RestricaoArquitetura.chamadaCamadaDao())
13      && RestricaoArquitetura.camadaGUI():
14      "Componentes da camada GUI podem acessar somente componentes
15      dos pacotes facade e beans";
16 }
```

Quadro 9 – Implementação do aspecto RegrasArquiteturaAspect

A partir das informações retiradas do estudo de caso, foram modeladas em AODM as restrições arquiteturais impostas pelo sistema (Figura 11).

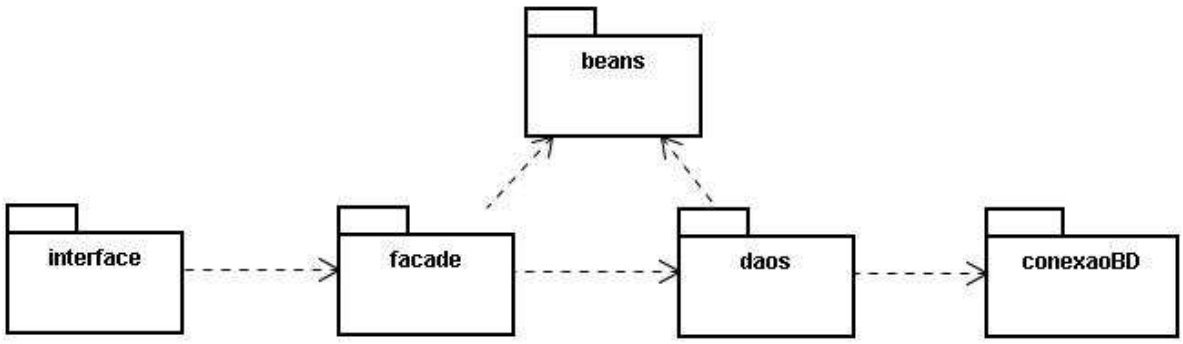

Figura 14 – Restrições Arquiteturais do Estudo de Caso.

5. MATERIAIS E MÉTODOS

O método utilizado neste trabalho é predominante do tipo bibliográfico, isso porque, é na maior parte, baseado em livros, artigos, pesquisas científicas, *sites* sobre o assunto abordado, anais de congressos e cursos já lecionados. Com base nestes materiais, o trabalho contém um estudo sobre modelagem orientada a aspectos, focando algumas linguagens MOA, mostrando a característica específica de cada uma, suas qualidades e defeitos.

Diante disto, foi realizado um estudo de caso foi tomado como base um sistema de reservas de laboratórios *online* e via *Bluetooth* no qual a MOA foi aplicada na parte do *desktop*.

As ferramentas de modelagem usadas foram *Jude Community* 5.0.2 , e *paint* 5.1 onde não foi possível mostrar todos os da modelagem em AODM, e *Eclipse Plataform* 3.3.0 juntamente com o *plugin Eclipse AspectJ Development Tools* 1.5.4 como ferramentas de programação.

Para a conclusão do trabalho são apontadas as características das linguagens abordadas, identificando as linguagens de modelagens orientada a aspectos mais viáveis atualmente, e com a aplicação de uma linguagem no estudo de caso é possível salientar suas vantagens e desvantagens.

6. CONCLUSÕES E TRABALHOS FUTUROS

Com o intuito de encontrar uma linguagem de modelagem padrão o paradigma de orientação a aspectos, várias abordagens foram criadas, mas nenhuma foi estabelecida como padrão, as propostas estão sendo estudadas. Dentre todas, quatro linguagens foram aqui abordadas.

Neste trabalho foram enfatizadas as linguagens *UMLaut* e *AODM* - com a qual foi realizado o estudo de caso. Ambas são de extensão UML; a UMLaut é um framework que usa um kit de combinadores específicos para cada aplicação para construção de modelos de design detalhados a partir de modelos UML orientados a aspectos e de alto nível(CHAVEZ,2004; p119), porém ainda é necessário ter cautela na utilização desta já que suas fontes são poucas, e ainda, apesar de ser realizada com estereótipos, não suporta todos os requisitos de UML, e tendo sua própria ferramenta é necessário suporte para esta, o qual através deste trabalho não foi possível encontrar.

AODM é muito flexível isso por ser utilizada através de estereótipos, estes usados para representar aspectos de *AspectJ* devem ser usados cautelosamente, para não fazer desta extensão um uso comum entre desenvolvedores e assim quebrando a busca por ferramentas e linguagens padrão.

A AODM aplicada no sistema de reservas de laboratório descreve uma reengenharia do diagrama de classes *Desktop*, o qual foi possível implementar separadamente os interesses transversais que intercortam todo o diagrama.

Como trabalhos futuros é proposta uma evolução de como modelar um aspecto, para ter o mesmo poder semântico da modelagem OO, seria um exemplo de aspecto estático, atuando em tempo de compilação onde se tem uma herança em OO e o que está sendo proposto é uma herança entre classes e aspectos, onde classe A e B herdam as propriedades do aspecto H.

REFERÊNCIAS

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML Guia Do Usuário.** 2ª Rio de Janeiro: Campus/elsevier, 2006.

CALDAS, Carlos Eduardo Costeira; RODRIGUES, Cíntia Keli Cabral. **Análise e Projeto Orientado a Aspectos em Sistemas Corporativos: Conceituação, Formulação e Estudo de Caso.** 2006. 81 f. Trabalho de Conclusão de Curso (Graduação) - Curso de Ciência da Computação, Departamento de Informática, Universidade Federal do Pará, Belém, 2006. Disponível em: <www.labes.ufpa.br/.../Tcc/action.do;jsessionid=7BA20BE6531504721BA53C48C74C962C?act=download&id=46>. Acesso em: 01 nov. 2007.

CARTAXO, Emanuela Gadelha. **Geração de Casos de Teste Funcional para Aplicações em Celulares.** 2006. 159 f. Dissertação (Mestre) - Curso de Ciência da Computação, Universidade Federal de Campina Grande, Campina Grande, 2006. Disponível em: <http://copin.ufcg.edu.br/twikipublic/pub/COPIN/DissertacoesMestrado/Dissertacao-EmanuelaGCartaxo.pdf>. Acesso em: 20 set. 2007.

CHAVEZ, Christina; GARCIA Alessandro; Kulesza, Uirá. **Programação Orientada a Aspectos.** Anais do XVII Simpósio Brasileiro de Engenharia de Software, Manaus: Universidade Federal do Amazonas, 2003.

CHAVEZ, Christina Von Flach Garcia. **Um Enfoque Baseado em Modelos para o Design Orientado a Aspectos.** 2004. 298 f. Tese de Doutorado (Doutorado) - Departamento de Ciência da Computação, Puc - Rio - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2004.

DEITEL, H. M.; DEITEL, P. J.. **Java Como Programar:** Projeto orientado a objetos com a UML e Padrões de Projeto. 4ª. Ed. Porto Alegre: Bookman, 2003.

GARCIA, Alessandro et al. (Org.). **WASP'04 – 1º Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos:** Relatório Final. Disponível em: <http://twiki.im.ufba.br/bin/view/WASP04/WebHome>. Acesso em: 10 set. 2007.

GARCIA, Vinícius Cardoso. **Uma ferramenta CASE para o Desenvolvimento de Software Orientado a Aspectos.** 2004. 6 f. Artigo Científico (Graduação) - Departamento de Computação, Universidade Federal de Pernambuco, Recife, 2004.

IRISA (Org.). **UMLAUT: Unified Modeling Language All pUrposes Transformer.** Disponível em: <http://www.irisa.fr/UMLAUT/>. Acesso em: 01 out. 2007.

LOZANO, Fernando. **Programação Orientada a Objetos com Java.** Sun Java Certified Programmer IBM Visual Age Certified Associate Developer. Disponível em: <www.lozano.eti.br>. Acesso em: 13 abr. 2007.

MONTEIRO, Elaine da Silva. **Um Estudo Sobre Modelagem Orientada a Aspectos Baseada em AspectJ e UML.** 2004. 62 f. Monografia (Graduação) - Curso de Sistemas de Informação, Ulbra - Centro Universitário Luterano de Palmas, Palmas, 2004.

OLIVEIRA, André Luiz de. **Uma Aplicação Utilizando as Tecnologias Bluetooth e J2ME.** 2007. 136 f. Trabalho de Conclusão de Curso – Departamento de Informática, Faculdades Luiz Meneghel, Universidade Estadual do Norte do Paraná, Bandeirantes, 2007.

OLIVERIA, André Luiz de; MENOLLI, André Luis Andrade; COELHO, Ricardo Gonçalves. Material de Auxílio ao Mini Curso de AspectJ. **Programação Orientada a Aspectos Utilizando a Linguagem AspectJ.** Bandeirantes: Falm, 2007. 60 p.

PENCZEK, Leonardo. AFR: **Uma Abordagem para a Sistematização do Reúso de Frameworks Orientados a Aspectos.** 2006. 139 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, Pontifícia Universidade

Católica do Rio Grande do Sul - Faculdade de Informática, Porto Alegre, 2006.

RAINONE, Flávia. **Programação Orientada a Aspectos Dinâmica.** 2005. 24 f. Plano de Monografia (Graduação) - Curso de Ciência de Computação, Departamento de Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2005.

SOUZA, Rodrigo Rocha Gomes e. **REAJ: uma ferramenta para engenharia reversa de código AspectJ.** 2007. 59 f. Monografia (Graduação) – Departamento de Ciência da Computação, Instituto de Matemática, Universidade Federal da Bahia, Salvador, 2007.

STEINMACHER, Igor Fábio; GIMENES, Itana Maria De Souza; LIMA, José Valdeni De. **Uma Extensão da UML para Modelagem Orientada a Aspectos Baseada em AspectJ.** Disponível em: <http://www.inf.ufrgs.br/~ifsteinmacher/pdf/public/AOP-Fitem.pdf>. Acesso em: 25 out. 2007.

VEDOATO, ROBERTO, 2007. **Paradigma de Orientação a Objetos (OO).** Material de apoio da disciplina. Curso de Sistemas de Informação, UENP, FALM, Bandeirantes, 2007.

WHITLAM, John et al. (Comp.). **COLLINS GEM DICIONÁRIO:** INGLÊS - PORTUGUÊS/ PORTUGUÊS - INGLÊS. 2ª Glasgow: Disal, 1997.

WINCK, Diogo Vinícius; GOETTEN JUNIOR, Vicente. **AspectJ: Programação Orientada a Aspectos em Java.** 1ª São Paulo: Novatec, 2006. 229 p.

APÊNDICE

Diagrama de restrições arquiteturais em AODM.

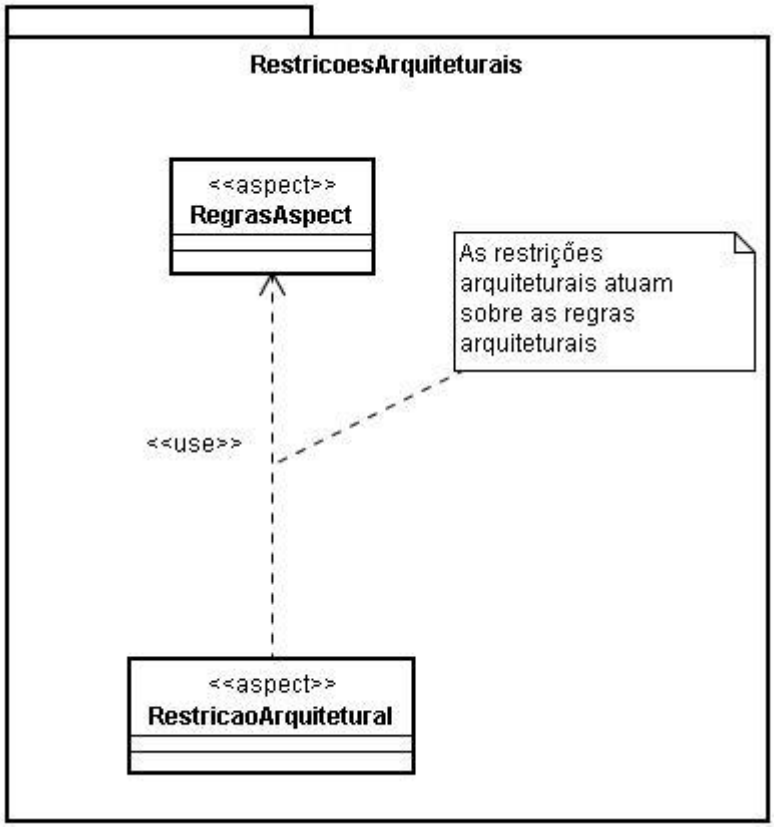

Diagrama de classes do estudo de caso com AODM completo.