UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA ESTATÍSTICA

Pedro Ghilardi

MODELAGEM DE ASPECTOS POR MÚLTIPLOS PONTOS DE VISTA

Florianópolis

2011

Pedro Ghilardi

MODELAGEM DE ASPECTOS POR MÚLTIPLOS PONTOS DE VISTA

Dissertação submetida ao Programa de Pós-Graduação em Ciências da Computação para a obtenção do Grau de Mestre em Ciências da Computação.
Orientador: Prof. Dr. Ricardo Pereira e Silva

Florianópolis

2011

Catalogação na fonte elaborada pela biblioteca da
Universidade Federal de Santa Catarina

A ficha catalográfica é confeccionada pela Biblioteca Central.

Tamanho: 7cm x 12 cm

Fonte: Times New Roman 9,5

Maiores informações em:
http://www.bu.ufsc.br/design/Catalogacao.html

Pedro Ghilardi

MODELAGEM DE ASPECTOS POR MÚLTIPLOS PONTOS DE VISTA

Esta Dissertação foi julgada aprovada para a obtenção do Título de "Mestre em Ciências da Computação", e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciências da Computação.

Florianópolis, 18 de julho 2011.

Prof. Chefe, Dr. Ronaldo Santos Mello
Coordenador do Curso

Banca Examinadora:

Prof. Dr Ricardo Pereira e Silva
Presidente

Prof. Dr. Ricardo Pereira e Silva
Orientador

AGRADECIMENTOS

Agradeço ao Prof. Ricardo pela sua orientação e compreensão.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

LISTA DE SÍMBOLOS

SUMÁRIO

1 INTRODUÇÃO

Este capítulo contextualiza o tema deste trabalho, apresentando motivações para o uso da programação orientada a aspectos e a necessidade de realizar a modelagem de sistemas orientados a aspectos. Apresentam-se os objetivos, a metodologia da pesquisa e a justificativa para realização do estudo. Ao final, apresentam-se os resultados esperados, as limitações e a organização desta dissertação.

1.1 CONTEXTUALIZAÇÃO

A Programação Orientada a Objetos (POO) é um paradigma de programação amplamente difundido e utilizado para o desenvolvimento de aplicações. Esse paradigma permite implementações em um maior nível de abstração e o reuso de comportamentos (**??**).

O desenvolvimento de um sistema utilizando POO geralmente é dividido em quatro fases: análise, projeto, implementação e testes (PRESSMAN, 2001). Durante as fases de análise e projeto eliciam-se os requisitos e elabora-se a modelagem do sistema. Esta modelagem deve expressar características estruturais e dinâmicas. As características estruturais representam os interrelacionamentos entre os componentes do sistema: as classes em um programa orientado a objetos. A parte dinâmica representa as funcionalidades do sistema e como elas são realizadas em tempo de execução. Ambas características devem ser modeladas em alto (modelos representando o domínio do problema) e baixo nível de abstração (modelos próximos ao nível de código), obtendo assim uma visão do todo e da parte (SILVA, 2000).

Na fase de análise, primeiramente deve-se descobrir qual o problema do cliente e eliciar quais são os requisitos do sistema. Cada requisito pode ser classificado como um interesse do cliente afim de atingir um objetivo final (**??**). Ao final do desenvolvimento, após a modelagem nas fases de análise e projeto e o código nas fases de implementação e testes, todos interesses do cliente devem estar resolvidos, tendo como resultado um sistema completo e funcional. Os interesses podem ser separados em dois tipos (**??**):

- Interesses núcleo: Capturam uma funcionalidade principal e impactam apenas uma parte do sistema.
- Interesses entrecortantes (*Crosscutting concerns*): Capturam uma funcionalidade que impacta uma ou mais partes do sistema.

A POO pode ser utilizada para implementação de ambos tipos de interesses. O problema é que para sistemas complexos, há uma tendência de misturar interesses entrecortantes com interesses núcleo, prejudicando a manutenabilidade e compreensão do código. Conclui-se que a POO tem limitações para implementar extensões para comportamentos que impactam várias partes de um sistema: os interesses entrecortantes (**??**). A Programação Orientada a Aspectos (POA) é uma extensão a POO para permitir uma implementação mais elegante para interesses entrecortantes.

O objetivo da POA é a modularização dos interesses entrecortantes para que os mesmos fiquem separados dos módulos que implementam os interesses núcleo da aplicação (**??**). Os interesses devem ser separados em módulos em todas as fases do desenvolvimento: análise, projeto, implementação e testes. Assim, obtém-se um mapeamento direto de um interesse, facilitando a manutenção e compreensão de um sistema desenvolvido utilizando aspectos. A separação de interesses é uma excelente prática para a construção de sistemas complexos, pois quanto maior o número de interesses, maior a complexidade para implementá-los. Com a separação de interesses, cada módulo representa um interesse, que pode ser implementado separadamente, diminuindo a complexidade de implementação (**??**).

1.2 DEFINIÇÃO DO PROBLEMA

Segundo (SILVA, 2000), na modelagem de programas orientados a objetos podem-se identificar quatro pontos de vista essenciais: estrutural e comportamental de sistema e estrutural e comportamental de classe. Um sistema modelado pelos quatro pontos de vista permite a compreensão e manutenção do sistema e contribui para a geração de código em qualquer linguagem de programação, sendo considerada assim uma modelagem completa (SILVA, 2007).

Em relação a modelagem de programas orientados a aspectos, deve-se modelar a estrutura e a dinâmica do sistema. Algumas propostas já foram elaboradas para modelar aspectos: (**??**), (**??**), (**??**), (**??**), (**??**), (**??**) e (**??**). No entanto, não existe nenhuma abordagem para modelagem de aspectos que represente adequadamente as partes estrutural e dinâmica de um sistema, permitindo visualizar o efeito dos aspectos na modelagem. Além disso, é importante que a modelagem permita representar as características inerentes à POA, como a possibilidade de capturar múltiplos pontos de execução de um programa com apenas uma expressão regular. A maior parte das propostas se limitam a modelar apenas parte destas características.

1.3 OBJETIVOS

1.3.1 Objetivo Geral

Propor uma metodologia para modelar aspectos nas fases de análise e projeto com a segunda versão da UML. Esta modelagem deve representar a estrutura e a dinâmica de um sistema, automatizar parte do processo de modelagem, permitindo a alternância de visões da dinâmica do sistema (com e sem explicitação dos aspectos) e, com isso, facilitar a compreensão e manutenção dos modelos.

1.3.2 Objetivos Específicos

- Modelar as características inerentes a aspectos com a segunda versão da Unified Modeling Language (UML) (UNIFIED...,) .
- Permitir a alternância de visões do comportamento dinâmico de um sistema orientado a aspectos, visualizando o efeito de modelos de aspectos em modelos base.
- Representar a estrutura e a dinâmica de sistemas orientados a aspectos, fornecendo subsídios para a composição de modelos, geração de código e facilitando a compreensão e manutenção do sistema.
- Automatizar partes do processo de modelagem realizando a composição de modelos de aspectos com modelos base.

1.4 HIPÓTESE DE PESQUISA

É possível modelar todas as características de programas orientados a aspectos com a segunda versão de UML, bem como é possível manipular esta modelagem em um procedimento algorítmico capaz de explicitar e ocultar dinamicamente os aspectos da modelagem.

1.5 JUSTIFICATIVA

Tratando-se de POO, uma modelagem pode ser considerada completa se modelar os quatro pontos de vista fundamentais (SILVA, 2000). Em relação a programas orientados a aspectos, definem-se quatro requisitos principais para se obter uma modelagem completa:

1. Representação das características inerentes à programas orientados a aspectos como *wildcards*, pontos de corte, avisos e introduções.
2. Representação da estrutura e da dinâmica do sistema a fim de contribuir para geração de código, composição de modelos e com o objetivo de facilitar a compreensão e manutenção.
3. Estar dentro dos padrões de modelagem: A modelagem deve utilizar apenas elementos presentes na segunda versão da UML, ou estender a linguagem através de estereótipos, valores rotulados e restrições definidas em um Perfil UML.

Sabendo que a modelagem estrutural e dinâmica de um sistema facilita a compreensão e manutenção e também contribui para a geração de código e composição de modelos, conclui-se que este trabalho justifica-se pelo uso de artefatos que representem a estrutura e a dinâmica de um sistema, automatizando parte do processo de modelagem. A compreensão do fluxo de execução de programas orientados a aspectos é difícil, assim, justifica-se a implementação de um visualizador do efeito de aspectos em modelos base. Sabe-se também que a segunda versão da UML é um padrão e é amplamente utilizada na modelagem de aplicações, sendo assim, justifica-se que os modelos propostos estejam de acordo com a especificação proposta por esta linguagem. Em relação a aspectos, a modelagem de *wildcards* é fundamental para possibilitar a captura dos pontos do sistema que serão modificados de maneira praticável. Logo, justifica-se a preocupação em modelar todas as características inerentes a POA.

1.6 MÉTODO DE PESQUISA

A realização deste trabalho é composta pelos seguintes passos:

1. Análise de trabalhos correlatos, definindo parâmetros de comparação entre as diferentes propostas e estabelecendo a configuração ideal de parâmetros necessários para se ter uma modelagem completa para sistemas orientados a aspectos.
2. Elaborar a proposta de solução, satisfazendo a configuração ideal previamente definida.
3. Implementar o suporte para modelagem proposta no ambiente SEA (SILVA, 2000), gerando um produto deste trabalho, que é o suporte a modelagem de programas orientados a aspectos no ambiente SEA.
4. Realizar um caso de uso de exemplo com a proposta de modelagem, realizando a composição de aspectos em nível de modelo e utilizando o visualizador dinâmico.
5. Comparar a modelagem proposta com os trabalhos correlatos, explicitando os diferentes requisitos cumpridos por cada proposta.

1.7 CLASSIFICAÇÃO DA PESQUISA

A classificação desta pesquisa foi realizada baseando-se nas informações sobre metodologia de pesquisa de (SILVA, 2001). Em relação a natureza, classifica-se esta pesquisa como **aplicada**, pois ela tem como objetivo gerar conhecimentos para aplicação prática em cima de problemas específicos na área de modelagem de sistemas orientados a aspectos. A abordagem do problema utilizada nesta dissertação é **qualitativa**, com o pesquisador realizando a análise dos dados indutivamente, onde o processo e seu significado são os focos principais da abordagem. Em relação aos objetivos, classifica-se esta pesquisa como **exploratória**, realizando a análise dos trabalhos correlatos e identificando quais parâmetros são implementados por cada proposta para modelagem de sistemas orientados a aspectos. Finalmente, em relação aos procedimentos técnicos utilizados, classifica-se esta pesquisa como **pesquisa bibliográfica**, pois foi elaborada a partir de material já publicado sobre a modelagem de sistemas orientados a aspectos. Esta pesquisa também tem um caráter **experimental** com a implementação do protótipo para modelagem de aspectos no ambiente SEA.

1.8 RESULTADOS ESPERADOS

Com a conclusão desse trabalho tem-se como resultados esperados:

- Definição dos parâmetros ideais para uma proposta de modelagem para sistemas orientados a aspectos.
- Uma ferramenta que possibilite modelar um sistema orientado a aspectos, representando as características inerentes à POA, permitindo realizar a composição de aspectos e visualizar o efeito dos mesmos em modelos base.

- Comparação entre este trabalho e outros propostos na literatura, de acordo com os parâmetros ideiais para modelagem de aspectos.

1.9 LIMITAÇÕES DESTA DISSERTAÇÃO

Este trabalho foca na linguagem AspectJ, linguagem padrão para o desenvolvimento de programas orientados a aspectos em Java. Esta é a linguagem mais madura para o paradigma de POA, sendo a primeira linguagem originada a partir do trabalho pioneiro sobre POA (**??**). A ferramenta proposta neste trabalho não permitirá a geração de código na linguagem AspectJ, apenas a composição em nível de modelo.

1.10 ORGANIZAÇÃO DESTA DISSERTAÇÃO

Esta dissertação está organizada da seguinte maneira: o Capítulo 2 apresenta a fundamentação teórica com conceitos de programação orientada a aspectos e da linguagem AspectJ, conceitos de análise e projetos de sistemas e modelagem e meta-modelagem com a segunda versão da UML. O Capítulo 3 apresenta uma análise acerca dos trabalhos correlatos. O Capítulo 4 apresenta apresenta a metodologia de modelagem proposta. O Capítulo 5 apresenta os resultados obtidos e faz uma comparação entre a metodologia proposta e outras metodologias existentes. Finalizando, O Capítulo 6 apresenta as conclusões deste trabalho.

2 CONTEXTUALIZAÇÃO

O objetivo deste capítulo é fornecer subsídios para a compreensão dos assuntos tratados nesta dissertação. Primeiramente, contextualiza-se o paradigma de Programação por Aspectos (POA), apresentando a motivação para o uso de aspectos e as construções específicas da linguagem AspectJ, que devem ser representáveis na modelagem. A segunda parte discorre sobre análise e projeto de sistemas utilizando a segunda versão da UML. Apresentam-se alguns conceitos sobre análise e projeto e os diagramas da UML. O tópico final fala sobre meta-modelagem, que permite estender um modelo para representar elementos de um domínio específico.

2.1 PROGRAMAÇÃO POR ASPECTOS

Esta seção discorre sobre a separação de um sistema em um conjunto de interesses e os problemas decorrentes da implementação de interesses com os paradigmas tradicionais. Devido a estes problemas surge a necessidade de um novo paradigma para realizar a separação de interesses: o paradigma de Programação Orientada a Aspectos (POA). A seção é finalizada com a apresentação da principal linguagem para implementação de aspectos em Java: AspectJ.

2.1.1 Abstração de Interesses

Ao iniciar a análise e projeto de um sistema, existe um conjunto de requisitos que devem ser implementados para satisfazer as necessidades do cliente. Cada requisito pode ser considerado um interesse que deve ser realizado para finalizar o sistema como um todo. Assim, um sistema é composto por um conjunto de interesses. Em um sistema para gerenciamento de hotel é possível identificá-los como: reserva de quartos, *check-in*, *check-out*, plano de fidelização, controle de acesso, garantia de *performance*, etc (**??**). Segundo (**??**), um sistema contém dois tipos de interesses:

- **Interesses núcleo**: Capturam uma funcionalidade principal e impactam apenas uma parte do sistema. São implementados em um único módulo.
- **Interesses entrecortantes** (*Crosscutting concerns*): Capturam uma funcionalidade que impactará diversas partes do sistema. São implementados em diferentes módulos.

No sistema de gerenciamento de hotel é possível classificar reserva de quartos e plano de fidelização como interesses núcleo. Controle de acesso e garantia de *performance* impactam diversas partes do sistema, por isso, são classificados como interesses entrecortantes. Controle de acesso deve verificar quais usuários podem acessar quais partes do sistema e, por isso, deve ser executado em todos os interesses núcleo. Garantia de *performance* impacta todos os outros interesses, pois para garantir *peformance* todo o sistema deve ser implementado com cuidados específicos.

Utilizando os paradigmas convencionais para implementação de interesses, como a Programação Orientada a Objetos (POO), o módulo para implementar o interesse de reserva de quartos seria composto por código referente a reserva, e também a garantia de *performance* e controle de acesso. A figura 1 mostra os interesses presentes no módulo de reserva de quartos. Observa-se a implementação de vários interesses no mesmo módulo. Segundo (**??**), essa situação é conhecida como **emaranhamento de código** (*code tangling*).

Outro sintoma de implementação deselegante de interesses é o **espalhamento de código** (*code scattering*), quando o código referente a um mesmo interesse está disposto em diferentes módulos (**??**). O espalhamento de código pode acontecer em duas situações:

- **Duplicação de código**: Diferentes módulos núcleo executam código referente a um mesmo interesse entrecortante.

  Exemplo: Interesse para registro de informações (*logging*) de um sistema; cada módulo núcleo contém chamadas a métodos da API de *logging* para registrar informações.

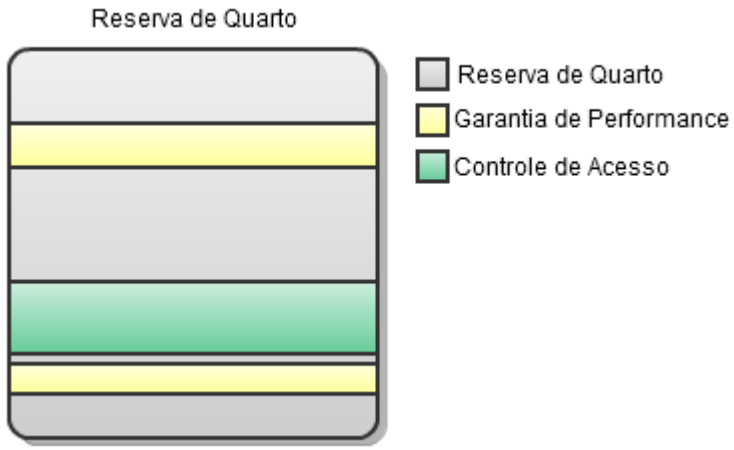

Figura 1 – Separação de interesses do módulo para Reserva de Quarto

- **Complementação de código**: Diferentes módulos executam código complementar para implementar um interesse entrecortante.

  Exemplo: Interesse para autorização em um sistema; um módulo implementa o gerenciamento da sessão, outro módulo implementa o gerenciamento de permissões e controle de acesso e outro realiza a autenticação de usuários; cada módulo implementa uma parte do interesse de autorização.

O emaranhamento e espalhamento de código dificultam a manutenção de um sistema, pois modificar um interesse impacta em modificar diferentes módulos. Além disso, é complicado realizar um rastreamento entre interesses e módulos, pois o código de um interesse está disposto em mais de um módulo. Outro problema é a dificuldade de reuso de módulos devido a dependência de um módulo com vários interesses (**??**).

2.1.2 A necessidade de aspectos

Os problemas destacados na sessão 2.1.1 indicam a necessidade de separação de interesses em diferentes módulos. A Programação Orientada a Aspectos (POA) (**??**) é um paradigma de programação para representar elegantemente os interesses de um sistema que impactam diversos módulos. Uma representação elegante de um interesse o separa em um único módulo e permite o reuso entre diferentes aplicações. O objetivo da POA é a modularização dos interesses entrecortantes para que os mesmos fiquem separados dos módulos que implementam os interesses núcleo da aplicação (**??**).

É importante observar que, boa parte dos interesses de uma aplicação são interesses núcleo e podem ser implementados elegantemente com a POO. Por isso, a POA não pretende substituir a Programação Orientada a Objetos (POO), mas sim complementá-la com uma melhor representação para os interesses entrecortantes.

2.1.3 Metodologia de desenvolvimento

Segundo (**??**), para implementar um sistema utilizando aspectos geralmente executam-se três fases:

1. **Decomposição de Aspectos**: Identificação de quais requisitos são interesses núcleo e quais são interesses entrecortantes.

2. **Implementação de Interesses**: Implementação de cada interesse separadamente.

3. **Composição de Aspectos** (*Weaving*): É a implementação de um aspecto para cada interesse entrecortante. O aspecto define o comportamento que será executado em determinados pontos de

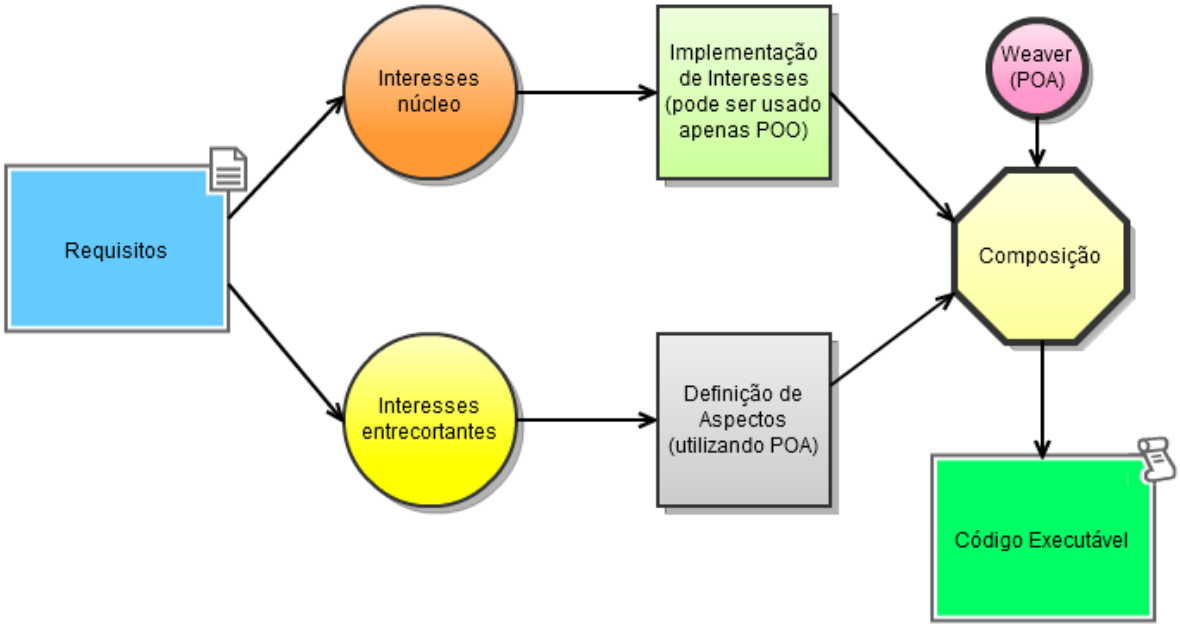

Figura 2 – Fluxo de desenvolvimento de uma aplicação com aspectos

execução de um ou mais interesses. Cada interesse entrecortante está contido em um único módulo: o aspecto. Após a implementação dos aspectos, se inicia o processo de composição *weaving* que insere o comportamento dos interesses entrecortantes nos interesses núcleo (nos pontos definidos nos aspectos). A figura 2 mostra o fluxo de desenvolvimento de uma aplicação orientada a aspectos.

2.1.4 A linguagem AspectJ

A linguagem AspectJ (ASPECTJ, 2012) é uma extensão a linguagem Java para se trabalhar com aspectos. Qualquer programa implementado em Java pode ser estendido utilizando AspectJ. A linguagem provê mecanismos para representar interesses entrecortantes e permite a composição dos mesmos com os interesses núcleo de um sistema. A linguagem oferece construções de extensão dinâmicas e estáticas. As extensões dinâmicas permitem que um novo comportamento seja executado antes, durante ou depois de um determinado ponto de execução do sistema. As extensões estáticas permitem adicionar novos elementos na estrutura das classes, por exemplo, a inserção de um novo método ou atributo.

2.1.4.1 Construções Dinâmicas

Um dos principais conceitos dinâmicos que devem ser compreendidos na linguagem é o de **ponto de junção**. Um ponto de junção é um determinado ponto na execução de um programa. É importante observar que, um ponto de junção não é uma construção sintática de AspectJ, mas sim um conceito.

A figura 3 descreve o fluxo de execução de um programa através da troca de mensagens entre diferentes objetos. Nesta troca de mensagens existem diversos pontos de junção que podem ser capturados pela linguagem AspectJ. A seguir, será destacado cada ponto de junção nesse fluxo de execução. No início da troca de mensagens, é chamado o método **umMetodo()** do **objetoA**. A chamada de um método em AspectJ é representada pelo ponto de junção *call*. Após a chamada do método, o código do mesmo será executado. A execução de um método é representada pelo ponto de junção *execution*. A duração de execução do método **umMetodo()** pode ser visualizada na linha de vida em azul. No início da execução de **umMetodo()**, realiza-se a chamada a **outroMetodo()**. A linha de vida em verde representa a duração da execução desse método. Dentro de **outroMetodo()** o método **metodoInterno()** é chamado.

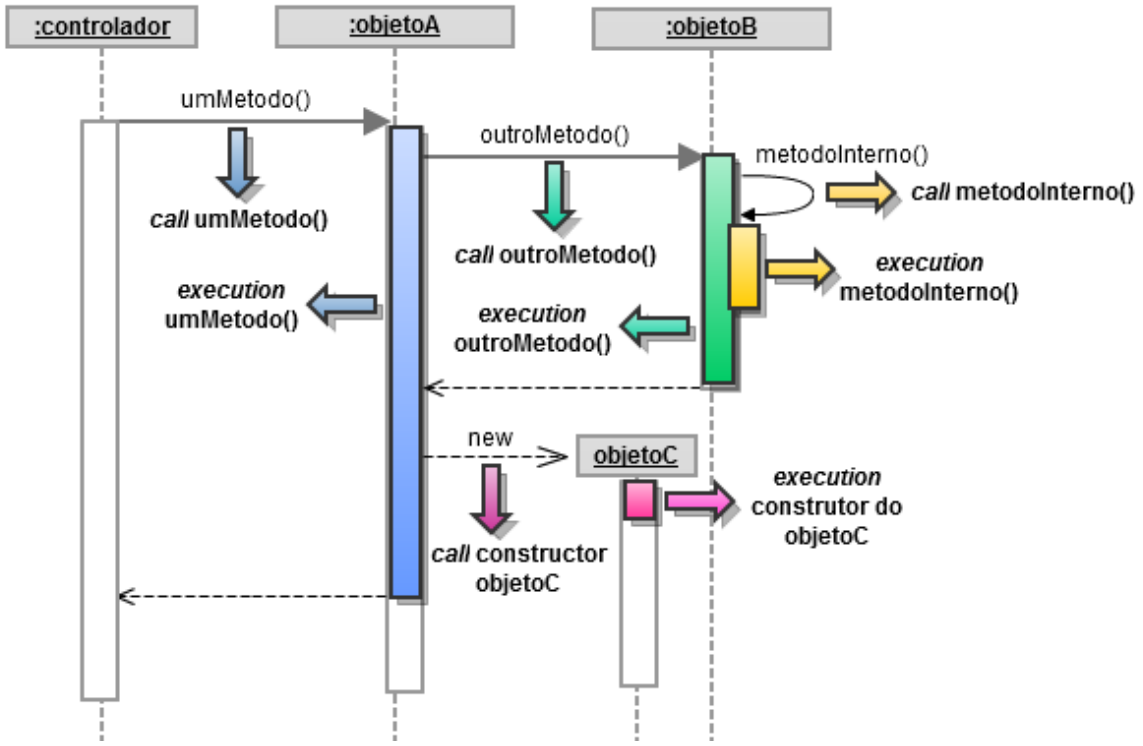

Figura 3 – Identificação de pontos de junção.

```
public static void main(String[] args) {
    ObjetoA objetoA = new ObjetoA();
    objetoA.umMetodo();
}
```
ponto de junção call()

```
public void umMetodo(){
    objetoB.outroMetodo();
    new ObjetoC();
}
```
ponto de junção execution()

Figura 4 – Diferença entre os pontos de junção de chamada (call) e execução (execution).

A linha de vida em amarelo representa a duração de sua execução. Finalmente, instancia-se o **objetoC** através da chamada *new*. A chamada de um construtor também é um ponto de junção *call* em AspectJ. A execução do mesmo pode ser capturada com o ponto de junção *execution*. O tempo de execução da instanciação desse construtor pode ser visualizado na linha de vida em rosa. Os pontos de junção *call* e *execution* são utilizados para capturar chamada e execução de métodos e construtores.

É importante observar a diferença entre entre os pontos de junção de **chamada** (*call*) e **execução** (*execution*). Um ponto de junção de chamada não está no código do método ou construtor sendo chamado, mas sim no código de quem está chamando o método ou construtor em questão. Observe na figura 4 que a chamada (*call*) ao método **umMetodo()** é realizada dentro do código *main* da aplicação. Já o ponto de junção de execução de um método ou construtor é disparado no corpo do método ou construtor em questão. Observa-se na figura que a execução (*execution*) do método **umMetodo()** refere-se à execução do código do próprio método.

Os pontos de junção que capturam a chamada e execução de métodos e construtores são os mais utilizados. No entanto, o modelo de pontos de junção de AspectJ permite capturar também o tratamento de exceções, acesso e modificação de atributos (*get* e *set*), inicialização e pré-inicialização de objetos, contexto de uma execução e execução de avisos. Estes pontos de junção, embora menos utilizados, também devem ser representáveis em uma proposta para modelagem de programas orientados a aspectos.

Os pontos de junção em AspectJ definem quais são os pontos da execução de um programa possíveis de serem capturados. A linguagem deve disponibilizar alguma construção sintática para selecionar pontos

```java
public class Testing {

    public static void main(String[] args) {
        ObjetoA objetoA = new ObjetoA();
        objetoA.umMetodo();
        objetoA.doisMetodo();
        objetoA.tresMetodo();
    }

}
```

Figura 5 – Exemplo de código em Java.

de junção. Para isso, AspectJ disponibiliza os **pontos de corte** (*pointcuts*). Um ponto de corte permite selecionar um conjunto de pontos de junção.

Existem dois tipos de ponto de corte:

- **Com nome**: Tem um nome e pode ser referenciado dentro de um aspecto.
- **Anônimo**: Não tem um nome e não pode ser referenciado dentro de um aspecto. Geralmente é definido dentro de um ponto de corte nomeado.

O código da figura 5 mostra o exemplo de um simples código em Java. É possível capturar pontos específicos da execução deste código com pontos de corte. O ponto de corte **exemploDePontoDeCorte()** foi definido para capturar chamadas ao método **umMetodo()** de objetos do tipo **ObjetoA**. Este ponto de corte é composto por dois pontos de corte anônimos e pode ser visualizado na figura 6. O primeiro ponto de corte anônimo seleciona um ponto de junção, capturando as chamadas ao método **umMetodo()** de objetos do tipo **ObjetoA**. O segundo ponto de corte anônimo seleciona vários pontos de junção, pois captura qualquer chamada a membros (atributos, métodos, construtores, etc) de objetos do tipo **ObjetoA**. É importante observar que, o segundo ponto de corte anônimo seleciona vários pontos de junção em uma única definição. Entre os dois pontos de corte anônimos encontra-se o operador binário **&&**. Este operador especifica que o ponto de corte **exemploDePontoDeCorte()** somente será satisfeito se **os dois pontos de corte anônimos forem satisfeitos**. Assim, o ponto de corte **exemploDePontoDeCorte()** seleciona apenas um ponto de junção: execução da chamada ao método **umMetodo()** com o método membro de um objeto do tipo **ObjetoA**. A captura de um único método pode ser visualizada no código na parte inferior da figura 6 (método selecionado está destacado com uma flecha laranja).

Para possibilitar a captura de vários pontos de junção em um mesmo ponto de corte de maneira praticável, AspectJ disponibiliza os **wildcards**. Um *wildcard* é semelhante a uma expressão regular. As seguintes notações estão disponíveis na definição de *wildcards*:

- * representa qualquer número de caracteres, exceto pontos.
- .. representa um ou mais caracteres, incluindo qualquer número de pontos.
- + representa uma subclasse ou sub-interface de um dado tipo.

Utilizando *wildcards* é possível modificar o ponto de corte especificado na figura 6 para que capture chamadas para qualquer método pertencente a objetos do tipo **ObjetoA**. Para isso, modifica-se o primeiro ponto de corte anônimo, substituindo **umMetodo()** pela notação *****, capturando agora todos os métodos do ObjetoA, independente do nome. Além disso, adiciona-se a notação **..**, capturando métodos com qualquer número de parâmetros. A figura 7 mostra o ponto de corte redefinido. O novo ponto de corte captura a chamada de qualquer método, com qualquer número de parâmetros e qualquer tipo de retorno de um objeto do tipo **ObjetoA**. A captura de todos os métodos pode ser visualizada no código da parte inferior da figura 6 (métodos selecionados estão destacados com flechas laranjas).

Os pontos de corte das figuras 6 e 7 foram definidos utilizando **padrões de assinatura** (*signature patterns*). Um padrão de assinatura é utilizado para especificar quais assinaturas de um programa

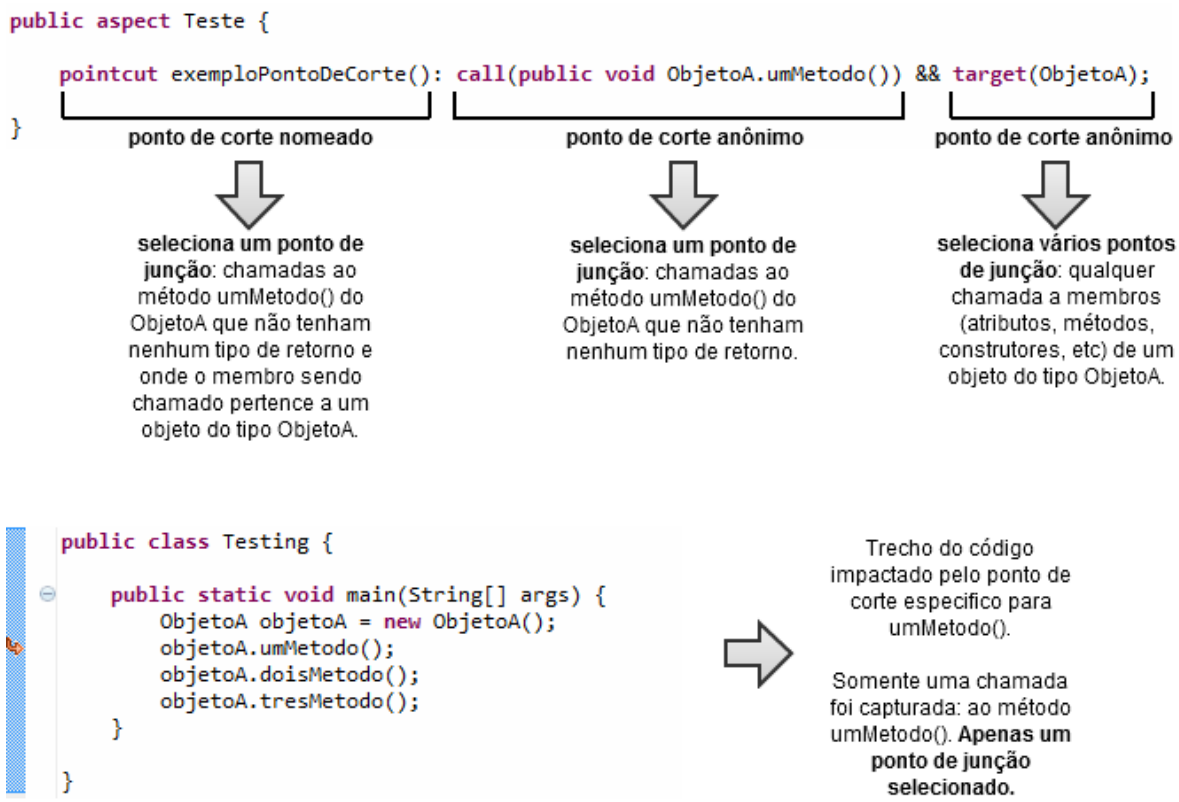

Figura 6 – Exemplo de ponto de corte.

em Java serão capturadas. No exemplo de ponte de corte da figura 6, a assinatura **public void ObjetoA.umMetodo()** permite capturar as chamadas ao método **umMetodo()** do **ObjetoA**. AspectJ disponibiliza padrões de assinatura para especificar pontos de corte para capturar métodos, construtores, tipos, exceções, atribuições, etc. Os seguintes padrões de assinatura estão disponíveis:

- **Assinaturas de Tipo** (*AssinaturaDeTipo*): Permite capturar definições de classes e interfaces. É possível especificar o pacote e o nome do tipo a ser capturado.
- **Assinaturas de Método e Construtores** (*AssinaturaDeMetodo* e *AssinaturaDeConstrutor*): Permite capturar métodos e construtores. É possível especificar escopo, tipo de retorno, localização e nome do método ou construtor e tipos de argumentos.
- **Assinaturas de Atributos** (*AssinaturaDeAtributo*): Permite capturar definições de atributos de classes. É possível especificar o escopo, tipo do atributo, localização e nome do atributo.

A figura 8 mostra exemplos de padrões de assinatura na captura de pontos de junção. O primeiro exemplo mostra o uso de um padrão de método para capturar chamadas ao método **umMetodo()** de objetos do tipo **ObjetoA** que não tenha retorno e com escopo público. O segundo exemplo mostra o padrão de tipo para capturar objetos do tipo **ObjetoA**. O terceiro mostra o padrão de atributo para capturar atribuições ao atributo **name** de objetos do tipo **ObjetoB**, em qualquer o escopo. O último exemplo apresenta o padrão de construtor para capturar a inicialização de objetos do tipo **ObjetoA** sem nenhum parâmetro.

Além do operador **&&** utilizado nos pontos de corte dos exemplos anteriores, AspectJ disponibiliza outros operadores binários. Os operadores disponíveis podem ser visualizados na tabela 1. Esses operadores são utilizados para formar regras complexas combinando diferentes pontos de corte.

Além dos pontos de corte para captura de execução, chamada, tratamento de exceção e atribuição, existem pontos de corte para outras situações mais específicas. Os **pontos de corte para captura do**

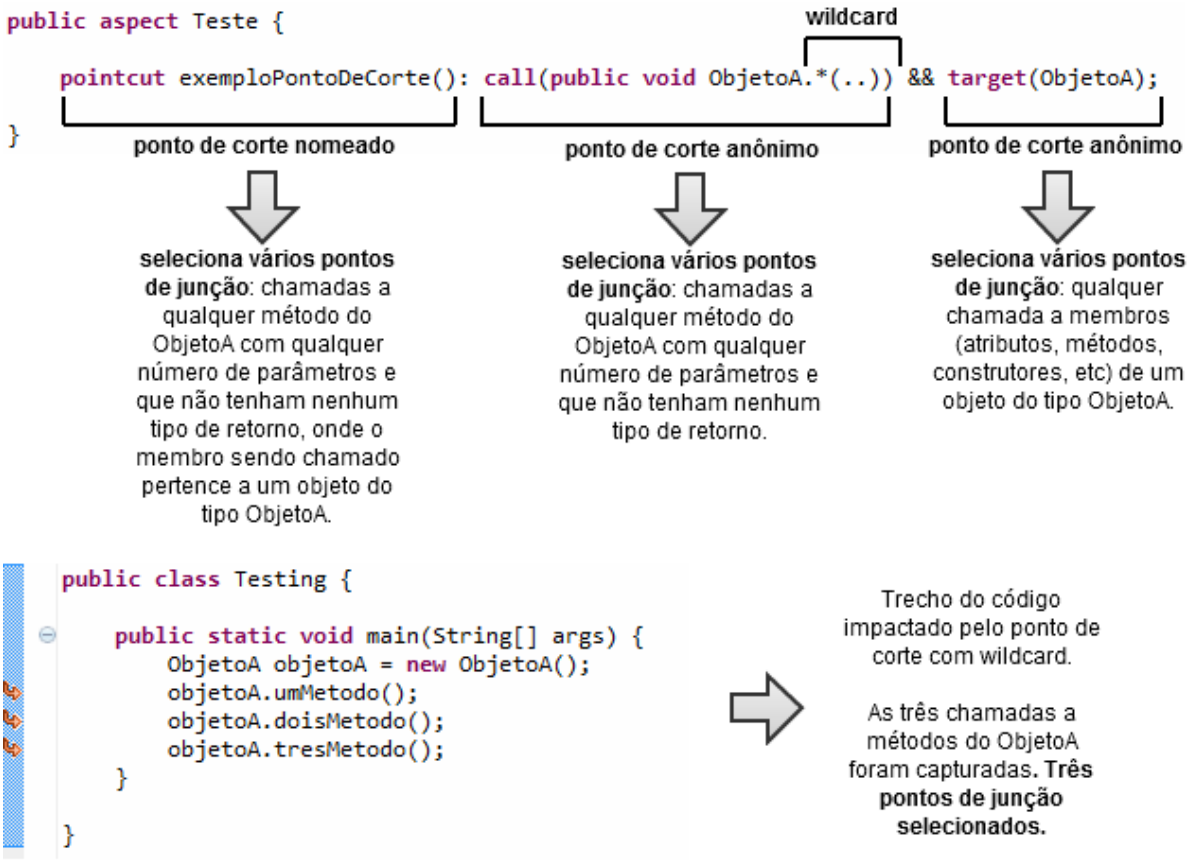

Figura 7 – Exemplo de ponto de corte utilizando wildcards.

Operador	Nome	Exemplo	Explicação do Exemplo
!	Negação	!ClasseA	Qualquer classe exceto ClasseA
||	Ou	ClasseA||ClasseB	ClasseA ou ClasseB
&&	E	ClasseA && ClasseB	ClasseA e ClasseB

Tabela 1 – Tipos de Pontos de Junção

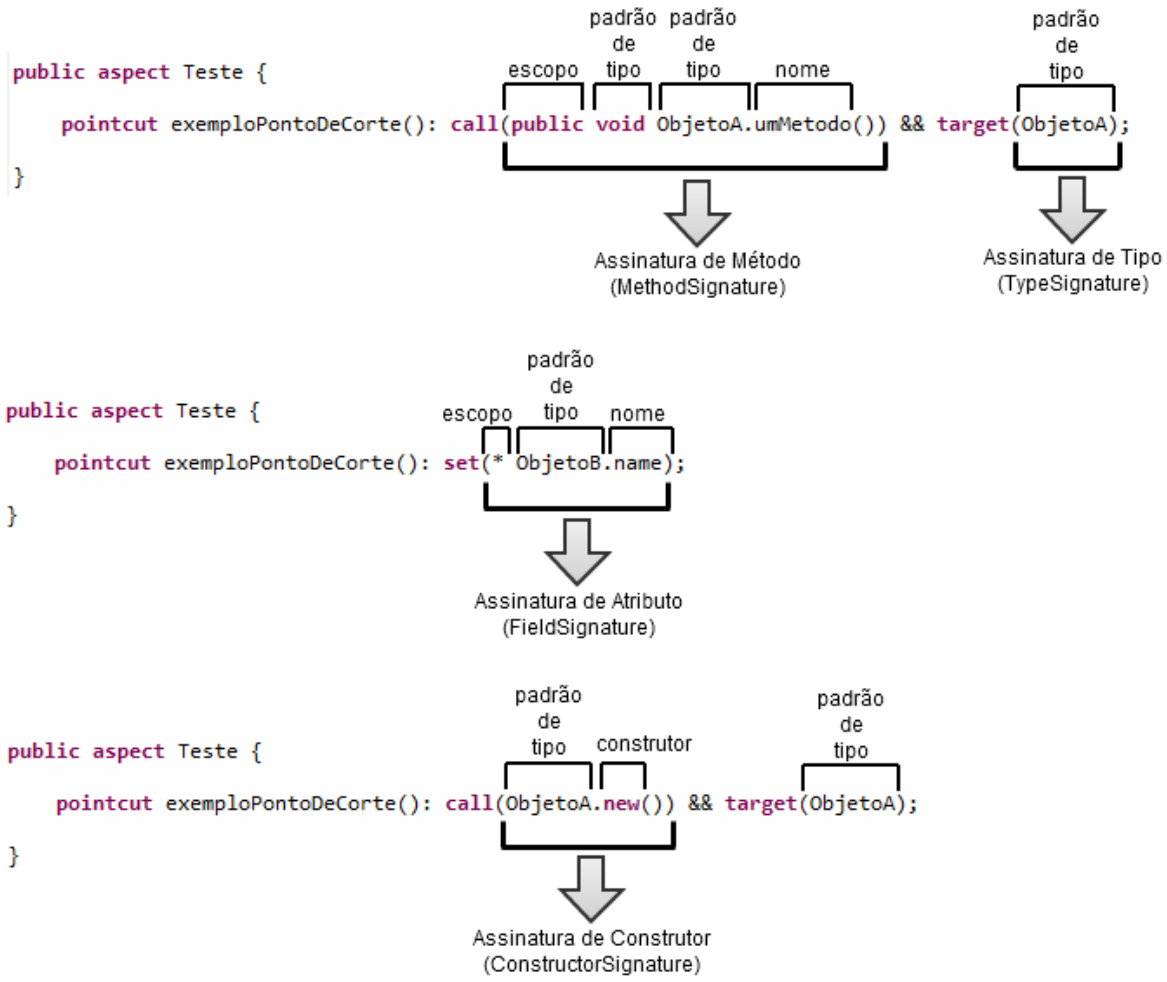

Figura 8 – Exemplo de assinaturas em AspectJ.

```
public aspect Teste {

    pointcut exemploPontoDeCorte(): cflow(call(public void ObjetoA.umMetodo()));

}
```

Figura 9 – Ponto de corte para captura do fluxo de execução inclusivo: cflow().

```
public aspect Teste {

    pointcut exemploPontoDeCorte(): cflowbelow(call(public void ObjetoA.umMetodo()));

}
```

Figura 10 – Ponto de corte para captura do fluxo de execução exclusivo: cflowbelow().

fluxo de execução capturam todos os pontos de junção a partir de um outro ponto de corte. Existem dois tipos: *cflow(PontoDeCorte)* e *cflowbelow(PontoDeCorte)*. O ponto de corte da figura 9 é do tipo *cflow()* e captura todos os pontos de junção disparados a partir do ponto de corte **call(public void ObjetoA.umMetodo())**, inclusive a chamada ao próprio método. O ponto de corte da figura 10 é do tipo *cflowbelow()* e captura os mesmos pontos de junção do anterior, exceto a chamada ao próprio método.

Existem também **pontos de corte baseados na estrutura léxica do código**. Estes pontos de corte capturam pontos de junção que ocorrem dentro de um determinado trecho de código. Existem dois tipos: *within(AssinaturaDeTipo)* e *withincode(AssinaturaDeConstrutor ou AssinaturaDeMétodo)*. O primeiro tipo captura os pontos de junção que ocorrerem dentro de classes, aspectos ou classes aninhadas de um determinado tipo (*AssinaturaDeTipo*. O segundo tipo captura os pontos de junção que estiverem dentro do código de um dado método ou construtor (*AssinaturaDeConstrutor ou AssinaturaDeMetodo*).

Outros tipos de **ponto de corte permitem capturar o contexto de uma execução**. O ponto de corte *this(TipoDoObjeto)* permite capturar todos os pontos de junção onde o objeto que está executando é do tipo *TipoDoObjeto*. Já o ponto de corte *target(TipoDoObjeto)* permite capturas os pontos de junção onde o objeto que está sendo chamado é do tipo *TipoDoObjeto*. Estes pontos de corte são importantes, pois permitem passar o contexto de uma execução, isto é, instâncias de objetos, para um aviso, o que será abordado ainda neste capítulo.

Para finalizar existem os **pontos de corte para argumentos**. Estes pontos de corte tem a seguinte sintaxe: *args(AssinaturaDeTipo, . . . , AssinaturaDeTipo)*. Eles permitem capturar pontos de junção baseados nos argumentos recebidos. Por exemplo, capturar os métodos que recebam três atributos do tipo *String*.

Após identificar quais pontos de junção serão capturados através de pontos de corte, deve-se especificar qual o comportamento que será executado antes, durante ou depois dos locais selecionados. Para isso, AspectJ propõe uma construção denominada **aviso**. Um aviso é uma construção parecida com um método em Java. Ele define um comportamento para ser executado. Existem três tipos de avisos:

- **Antes** (before): Executa antes do ponto de junção capturado.
- **Depois** (after): Executa depois do ponto de junção capturado. Existe uma variação ao aviso *after* que executará apenas se o ponto de junção capturado não lançar nenhuma exceção, isto é, só será executado se a execução do ponto de junção tiver sucesso. Esse tipo de aviso é denominado *after returning*.
- **Durante** (around): É o tipo de aviso mais poderoso, pois pode executar no lugar do ponto de junção capturado, continuar a execução original ou alterar o contexto de execução (**??**).

A figura 11 mostra um exemplo de um aviso que executa **depois**(*after*) do ponto de corte **exemploDePontoDeCorte()**. Este aviso recebe o contexto da execução como parâmetro (um objeto do tipo **ObjetoA**) e imprime uma mensagem com a representação textual deste objeto. O corpo deste aviso é o

```
public aspect Teste {

    pointcut exemploPontoDeCorte(ObjetoA a): call(public void ObjetoA.umMetodo()) && target(a);

    after(ObjetoA a): exemploPontoDeCorte(a) {
        System.out.println("Execução do Aviso, com o objeto: " + a.toString());
    }

}
```

Figura 11 – Exemplo de aviso com contexto de execução.

```
public aspect Teste {

    public String ObjetoA.atributoUm = "um";
    public String ObjetoA.atributoDois = "dois";

    public void ObjetoA.metodoIntroduzido(){
        System.out.println("Novo método no objeto A com dois atributos: " + this.atributoUm + this.atributoDois);
    }

}
```

Figura 12 – Introdução de métodos e atributos.

trecho de código que realiza a impressão da representação do objeto. Em AspectJ, o corpo de um aviso pode conter qualquer código Java. Observa-se também que o objeto passado no contexto de execução é referenciado no corpo do aviso. Os pontos de corte *target()* e *this()* são muito utilizados, pois permitem passar o contexto de execução para um aviso.

2.1.4.2 Construções Estáticas

Uma das construções estáticas propostas por AspectJ é a **introdução**, que permite alterar a estrutura de classes, aspectos e interfaces adicionando novos métodos e atributos. A figura 12 mostra um aspecto que introduz o método **metodoIntroduzido()** e os atributos **atributoUm** e **atributoDois** na classe do tipo **ObjetoA**. Os dois atributos introduzidos são utilizados no próprio método **metodoIntroduzido()** e que também foi introduzido. Isto é possível, pois o compilador AspectJ sabe que o método introduzido pertence ao **ObjetoA** e o objeto que executará este método será um objeto do tipo **ObjetoA**.

Outra funcionalidade disponível na linguagem é a **modificação da hierarquia de classes**, permitindo a definição de relacionamentos de herança, implementação de interfaces, dentre outras alterações (**??**). O exemplo da figura 13 mostra a introdução de um relacionamento de herança entre as classes **ObjetoA** e **ObjetoB**.

```
public aspect Teste {

    declare parents: ObjetoA extends ObjetoB;

}
```

Figura 13 – Introdução de relacionamentos de herança.

2.1.4.3 Aspecto

Resumidamente, para estender um sistema utilizando Java com AspectJ, deve-se identificar os pontos de junção que serão selecionados por um ponto de corte e implementar o aviso que introduzirá o novo comportamento antes, durante ou depois do ponto de corte. O elemento da linguagem que agrupa todas essas construções é o **aspecto**. Um aspecto é uma unidade de modularização semelhante a uma classe, mas com diferenças em relação ao ciclo de vida, pois não pode ser instancializado e não pode estender de um outro aspecto concreto. No entanto, um aspecto pode ser declarado como abstrato e aspectos concretos podem estendê-lo para implementar suas declarações abstratas.

2.1.4.4 Exemplo de Aspecto

O objetivo deste exemplo é utilizar a linguagem AspectJ para implementar de maneira elegante o padrão de projeto **Observador**. Este padrão permite que um ou mais objetos se cadastrem para escutar mudanças de um outro objeto. A implementação do padrão está presente nos exemplos da IDE para desenvolvimento com aspectos: AspectJ Development Tools (AJDT) (ASPECTJ, 2012).

Um dos requisitos deste padrão é que um ou mais objetos (observadores) possam escutar mudanças de um outro objeto (sujeito) e serem atualizados. Assim, identificam-se duas interfaces: *Subject* e *Observer*. A interface *Subject* deve armazenar seus observadores e prover métodos para adicionar, remover e obter os mesmos. A interface *Observer* deve prover métodos para associar e obter o sujeito observado. Esses requisitos são implementados no aspecto como introduções de métodos e atributos. O trecho de código da figura 14 mostra as introduções realizadas. Observa-se a introdução do atributo *observers* e dos métodos *addObserver()*, *removeObserver()* e *getObservers()* na interface *Subject*. Na interface *Observer* foram introduzidos os métodos *setSubject()* e *getSubject()* e o atributo *subject*.

Além de introduzir os métodos e atributos para permitir a associação entre sujeitos e observadores, deve-se implementar a lógica que capture mudanças nos sujeitos e avise os observadores. Essa lógica pode ser implementada com o uso de pontos de corte e avisos. O ponto de corte *stateChanges()* é responsável por escutar mudanças em um sujeito e após (*after*) cada mudança, um aviso é executado para atualizar os observadores. O ponto de corte *stateChanges()* deve ser abstrato, pois este ponto de corte será diferente para cada sujeito a ser observado. O trecho de código que implementa o ponto de corte e o aviso pode ser visualizado na figura 15.

Com esses requisitos implementados, é possível juntar os dois trechos de código em um aspecto abstrato que implementa o padrão **Observador**. Este aspecto é abstrato, pois tem o ponto de corte abstrato *stateChanges()*, que deve ser definido por um aspecto concreto, selecionando quais pontos de junção serão capturados para definir que uma mudança ocorreu. O aspecto abstrato recebe o nome de *SubjectObserverProtocol* e pode ser visualizado na figura 16.

Um desenvolvedor que deseja utilizar o padrão de projeto observador pode reusar o aspecto abstrato *SubjectObserverProtocol*, estendendo-o com a implementação de um aspecto concreto. Este aspecto concreto deve especificar qual classe faz o papel de sujeito, isto é, qual classe implementa a interface *Subject* e qual classe faz o papel de observador, isto é, qual classe implementa a interface *Observer*. Além disso, deve definir o ponto de corte *stateChanges()*, para especificar em quais pontos de junção serão detectadas mudanças.

Considerando como exemplo um sistema de interface gráfica com um botão e um texto com cor variável. Define-se como requisito que a cor deste texto deve modificar toda vez que o botão foi clicado. Este requisito pode ser implementado utilizando o aspecto abstrato. O pequeno sistema de interface gráfica contém a classe *Button* representando o botão e a classe *ColorLabel* representando o texto com cor. A classe *Button* é o sujeito observado, por isso implementa a interface *Subject*. O observador é a classe *ColorLabel* que implementa a interface *Observer*. Além disso, introduz-se o método *update()* na classe *ColorLabel* para atualizar a cor do texto quando houver alguma mudança no botão. O que está faltando definir é quais pontos na execução do programa geram mudanças no botão. Estes pontos são definidos ao implementar o ponto de corte abstrato *stateChanges()*. Define-se que serão capturadas as chamadas ao método *click()* da classe *Button*, onde o objeto alvo é do tipo *Subject* (neste caso é da classe *Button*, pois esta classe implementa *Subject*. O código do aspecto concreto para implementar o padrão observador pode ser visualizado na figura 17. Este aspecto captura cliques em um botão, atualizando a

```
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;
}

private Subject Observer.subject = null;

public void Observer.setSubject(Subject s) {
    subject = s;
}

public Subject Observer.getSubject() {
    return subject;
}
```

Figura 14 – Introduções de métodos e atributos para sujeito e observador.

```
abstract pointcut stateChanges(Subject s);

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

Figura 15 – Especificação do ponto de corte abstrato para capturar mudanças e implementação do aviso para atualizar observadores na ocorrência de uma mudança.

```
abstract aspect SubjectObserverProtocol {

    abstract pointcut stateChanges(Subject s);

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

    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;
    }

    private Subject Observer.subject = null;

    public void Observer.setSubject(Subject s) {
        subject = s;
    }

    public Subject Observer.getSubject() {
        return subject;
    }
}
```

Figura 16 – Aspecto abstrato para implementação do padrão de projeto Observador.

```
aspect SubjectObserverProtocolImpl extends SubjectObserverProtocol {

    declare parents: Button implements Subject;
    public Object Button.getData() { return this; }

    declare parents: ColorLabel implements Observer;
    public void    ColorLabel.update() {
        colorCycle();
    }

    pointcut stateChanges(Subject s):
        target(s) &&
        call(void Button.click());

}
```

Figura 17 – Aspecto concreto implementando o padrão de projeto Observador em um sistema de interface gráfica.

cor de um texto.

2.2 ANÁLISE E PROJETO COM UML

A análise e projeto de sistemas orientados a objetos é fundamental para o desenvolvimento de aplicações complexas. Aplicações complexas necessitam de um planejamento antes da implementação. Usualmente divide-se o desenvolvimento de um sistema em quatro fases: análise, projeto, implementação e testes (PRESSMAN, 2001). As fases de análise e projeto são as fases aonde realiza-se a maior parte do planejamento de um desenvolvimento. Já as fases de implementação e testes são responsáveis pela codificação com o objetivo de obter um programa executável e que cumpra os requisitos do cliente.

Nas fases de análise e projeto utilizam-se modelos que permitem representar o sistema em diferentes níveis de abstração, facilitando a compreensão e reduzindo a complexidade. A fase de análise pretende compreender os principais conceitos do domínio do problema, evitando o uso de termos computacionais. Já a fase de projeto foca na solução que será desenvolvida para produzir um sistema a partir da compreensão do problema.

2.2.1 Múltiplos pontos de vista de um sistema

Segundo (SILVA, 2007), um sistema orientado a objetos pode ser visualizado por diferentes pontos de vista:

- Estrutural de sistema: Essa visão contém o conjunto de elementos de um sistema orientado a objetos e seus relacionamentos.
- Estrutural de classe: Essa visão contém o detalhamento da estrutura de cada um dos elementos de um sistema.
- Comportamental de sistema: Essa visão permite compreender o conjunto de funcionalidades do sistema e como os elementos iteragem em tempo de execução.
- Comportamental de classe: Essa visão permite compreender o comportamento de um elemento isoladamente. Geralmente compreende-se a variação de estados desse elemento.

Uma modelagem que permita representar esses quatro pontos de vista pode ser considerada completa. Uma **modelagem completa** fornece subsídios para a geração de código e facilita a compreensão de um sistema, tornando-o manutenível e possibilitando o reuso.

Figura 18 – Uso de estereótipos em um diagrama de classes.

2.3 UML: SEGUNDA VERSÃO

A segunda versão da UML (UNIFIED...,) permite representar os elementos e o comportamento e a estrutura de um sistema nas fases de análise e projeto através de diagramas estruturais e comportamentais. Esta linguagem é um padrão da *Object Management Group* (OMG), por isso é compreendida e utilizada por grande parte dos desenvolvedores e analistas para realizar a modelagem de sistemas. Os diagramas da segunda versão da UML permitem a representação dos quatro pontos de vista essenciais para programas orientados a objetos.

2.3.1 Diagramas Estruturais

A segunda versão da UML disponibiliza sete diagramas estruturais: diagrama de classes, componentes, estrutura composta, instalação, objetos, pacotes e perfil. Os diagramas estruturais utilizados nesta dissertação são os diagramas de classe e o diagrama de perfil. Este último foi introduzido na segunda versão da linguagem e é fundamental para extensão da linguagem para um domínio específico.

2.3.1.1 Diagrama de Classes

O diagrama de classes permite representar a estrutura e os relacionamentos dos elementos de um sistema. Este diagrama permite visualizar o sistema como um todo, visualizando os relacionamentos entre os elementos e também permite visualizar a estrutura de cada elemento, com seus atributos e métodos.

2.3.1.2 Diagrama de Perfil

O diagrama de perfil foi introduzido na segunda versão da UML, com o objetivo de permitir estender o modelo da linguagem para representar conceitos de um determinado domínio de aplicações. Um perfil é composto por **estereótipos**, **restrições** e **valores rotulados**.

Um **estereótipo** adiciona uma semântica adicional a um elemento da UML. Geralmente adiciona-se um estereótipo para diferenciar os papéis dos elementos de um modelo. Por exemplo, a classe *RoomManager* da figura 18 foi associada ao estereótipo *¡¡Controller¿¿* para representar que esta classe tem o papel de controlador. É possível adicionar mais de um esreótipo a mesma classe. A classe *ReservationManager* foi associada aos estereótipos *¡¡Controller¿¿* e *¡¡Client¿¿* para representar que esta classe é um controlador e que encontra-se do lado do cliente. Os estereótipos *¡¡Client¿¿* e *¡¡Controller¿¿* estendem o elemento do meta-modelo da UML *Class*, podendo assim ser aplicados a qualquer classe. É importante observar que, um estereótipo pode ser associado a qualquer elemento do meta-modelo da UML. O atributo *id* da classe *Room* está associado ao estereótipo *¡¡key¿¿* para representar que este atributo define unicamente uma sala. O estereótipo *¡¡key¿¿* está estendendo o meta-modelo da UML *Attribute*.

Um estereótipo adiciona um papel a um elemento do modelo. Para adicionar mais informações a um elemento, podem-se definir **valores rotulados**. A linguagem permite associar zero ou mais valores rotulados a um estereótipo. Um valor rotulado pode ser um elemento do modelo, um número, um texto,

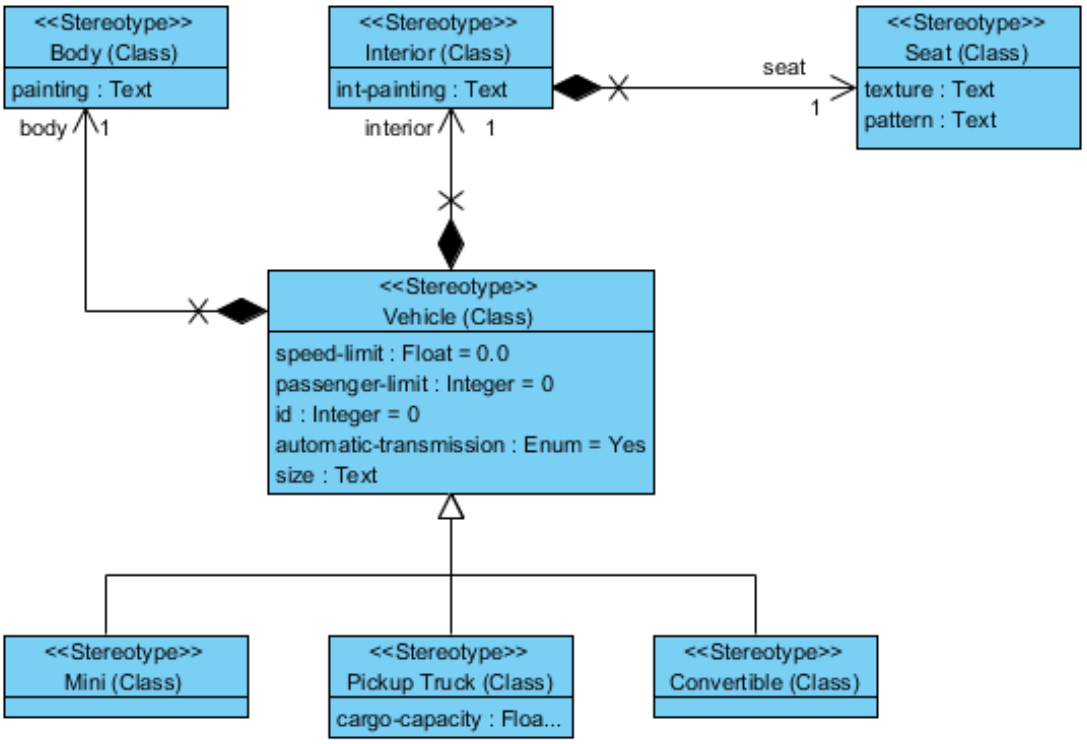

Figura 19 – Diagrama de Perfil para representar veículos.

um booleano ou uma enumeração definida pelo usuário. Ao utilizar um estereótipo em um modelo, deve-se definir os valores rotulados associados ao mesmo. Finalmente, **restrições** podem ser introduzidas ao modelo para garantir a consistência no próprio modelo e nos seus relacionamentos.

A figura 19 mostra a definição de um perfil UML para modelagem de sistemas que desejam representar veículos (PARADIGM, 2011). Foram definidos sete estereótipos que estendem o elemento do meta-modelo *Class*. Observa-se a generalização entre os estereótipos *Vehicle*, *Mini*, *Pickup Truck* e *Convertible*. O estereótipo *Pickup Truck* especializa o estereótipo *Vehicle*. O relacionamento de composição entre os estereótipos *Interior* e *Seat* define que o interior de um veículo deve ter no mínimo um assento. Observa-se também a presença de dois valores rotulados do tipo texto no estereótipo *Seat*: *texture* e *pattern*. Este perfil pode ser exportado no formato *XML Metadata Interchange* (XMI) ((OMG), 2011b) e utilizado por outras ferramentas do tipo *Computer Aided Software Engineering* (CASE) . Assim, é possível intercambiar perfis entre diferentes ferramentas CASE.

2.3.2 Diagramas Comportamentais

A segunda versão da UML possui sete diagramas comportamentais: diagrama de atividades, máquina de estados, casos de uso, comunicação, visão geral de interação, sequência e de tempo. Neste trabalho serão utilizados os diagramas de casos de uso e de sequência.

2.3.2.1 Diagrama de Casos de Uso

Este diagrama permite representar as funcionalidades de um sistema, sem detalhar como o sistema as realiza. Cada interesse do sistema pode ser modelado como um caso de uso. Cada caso de uso deve ser refinado em um menor nível de abstração para representar como o mesmo é realizado. Podem-se utilizar diagramas de atividades e de sequência para representar a realização de um caso de uso.

2.3.2.2 Diagrama de Sequência

O diagrama de sequência representa as trocas de mensagens entre objetos. Este diagrama permite compreender a interação entre objetos com foco no tempo e na ordem das mensagens durante uma execução. O comportamento de cada caso de uso pode ser refinado com um diagrama de sequência.

2.4 META-MODELAGEM

Uma linguagem geralmente é definida através de uma gramática na forma *Backus Naur Form* (BNF) . Uma linguagem bem definida pode ser interpretada de forma automatizada por um computador. Este método de definição de linguagens é utilizado até hoje para representar linguagens baseadas em texto. No entanto, as linguagens para modelagem contém elementos não-textuais e, por isso, necessitam de um novo mecanismo de definição. Este mecanismo é denominado **meta-modelagem**, que permite a descrição de uma linguagem na forma de um modelo (KLEPPE; WARMER; BAST, 2003). Um meta-modelo de uma linguagem define os elementos que podem ser utilizados na criação de modelos utilizando esta linguagem. Considerando a UML como exemplo, o meta-modelo da linguagem define elementos como Classe, Estado, Pacote, Operação, etc. Assim, um modelo definido utilizando UML pode definir instâncias de classes, estados, pacotes, operações, etc.

A OMG define uma arquitetura em quatro camadas para representar os modelos padrões para definição de modelos. Esta arquitetura pode ser visualizada na figura 20. O modelo M3 define elementos que podem ser utilizados para representar conceitos no modelo M2. O modelo M3 é considerado o meta-meta-modelo da OMG. O *Meta-Object Facility* (MOF)) é um padrão da OMG que define a linguagem que deve ser utilizada para definir linguagens para modelagem ((OMG), 2011a). O MOF está no nível M3. O modelo M2 especifica os elementos que podem ser utilizados no modelo M1. Um modelo no nível M2 é denominado um meta-modelo. Linguagens geralmente são definidas neste nível de modelo. Observa-se na figura 20 a definição de uma classe (*UML Class*) e de um atributo (*UML Attribute*) no nível M2. Estes elementos fazem parte da definição do meta-modelo da UML. O modelo M1 contém instâncias de elementos definidos no modelo M2. Este modelo é o que é definido pelo analista ao realizar a modelagem de um determinado sistema com UML. Na figura 20 observa-se a definição de duas classes: *Customer* e *Order* no nível M1. Além disso, definem-se os atributos *title*, *name* e *number* nessas classes. As classes são instâncias da meta-classe *UML Class*. Os atributos são instâncias da meta-classe *UML Attribute*. Finalmente, o modelo M0 representa as instâncias de um sistema representadas em um modelo. Um exemplo de instância é um cliente do tipo *Customer* com o nome *Joe Nobody*.

O meta-modelo da UML é definido no nível M2 a partir do MOF. Este meta-modelo é uma instância do MOF. Uma parte do meta-modelo pode ser visualizada na figura 21. Observa-se que qualquer elemento de um modelo da UML deve derivar de *ModelElement* e por isso deve ter um nome. Observa-se também a pressença de Classe (*Class*), Atributo (*Attribute*) e Operação (*Operation*). Estes são alguns dos elementos que podem ser utilizados na criação de uma modelagem utilizando UML. Os relacionamentos entre os elementos do meta-modelo podem introduzir restrições. Um exemplo de restrição é que uma operação pode ter zero ou mais parâmetros. Qualquer modelo da UML deve respeitar estas restrições e a estrutura definida no meta-modelo da OMG.

2.4.1 Extensões a UML

É possível estender a UML de duas formas: criação de um Perfil UML para representar os conceitos de um dado domínio de aplicação ou através da definição de um novo meta-modelo para este domínio de aplicação.

2.4.1.1 Extensão pela definição de um Perfil UML

A UML pode ser estendida com a definição de diferentes perfis para determinados domínios de aplicação. É importante observar que, o mecanismo de perfis não é um mecanismo de extensão de

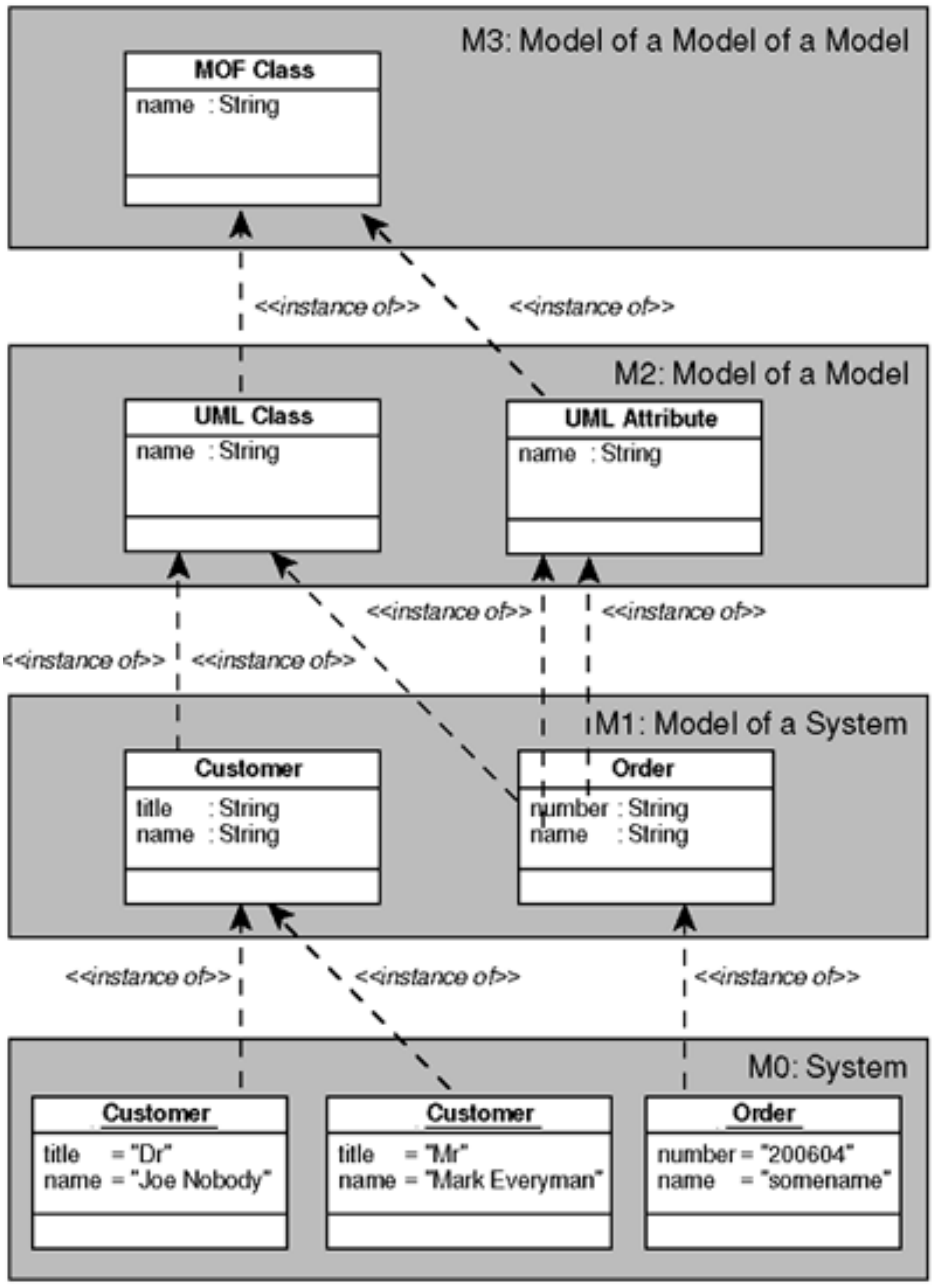

Figura 20 – Meta-modelo da Object Management Group (OMG).

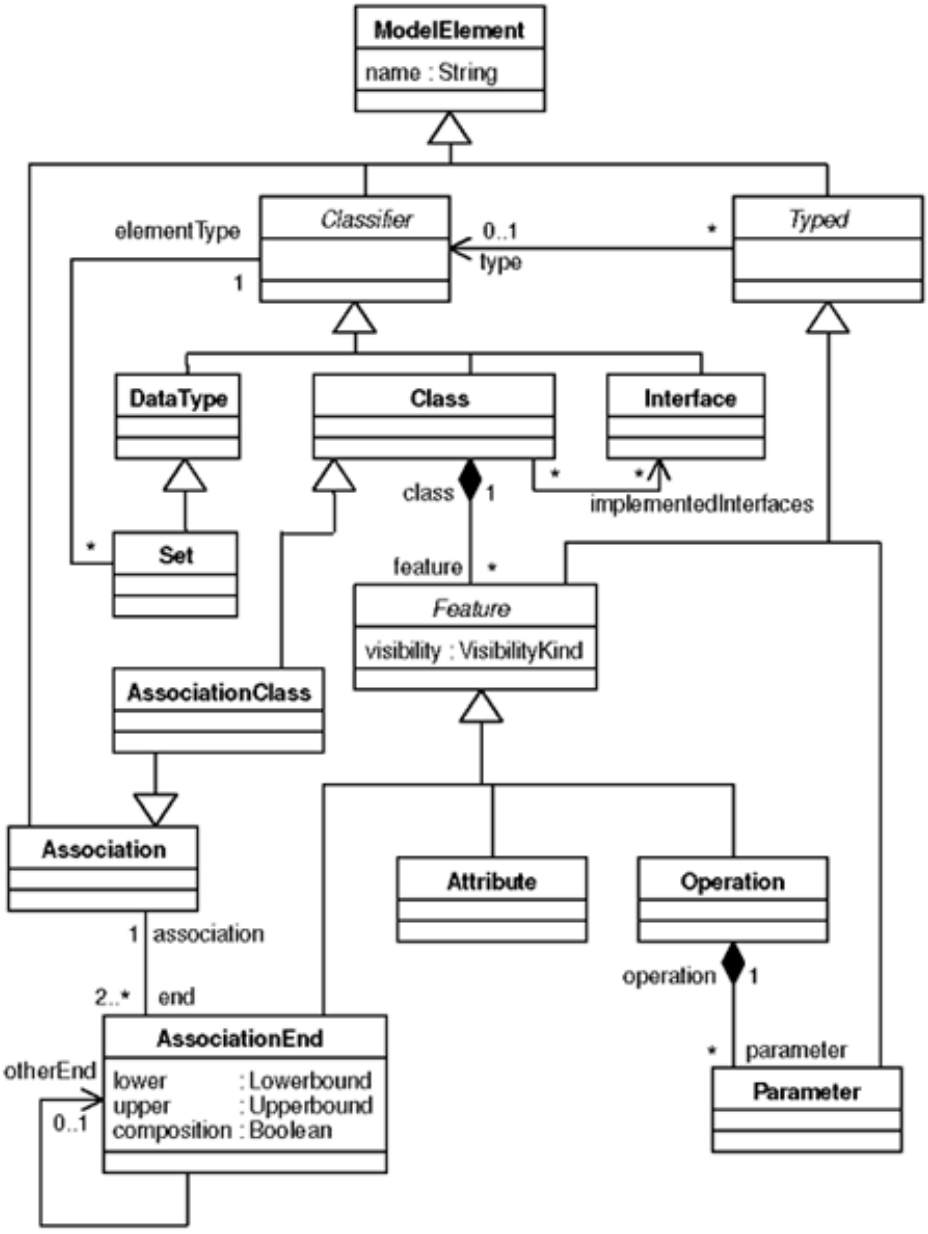

Figura 21 – Meta-modelo da UML.

primeira classe, o que significa que um perfil não pode modificar um meta-modelo (removendo restrições da UML, por exemplo), apenas adaptá-lo com construções específicas do domínio tratado. O mecanismo de extensão por perfis é considerado uma mecanismo leve para extensão da linguagem.

A grande vantagem de estender a UML através de perfis é que qualquer ferramenta que suporte a importação de perfis pode utilizar os conceitos estendidos pelo perfil UML. Como o diagrama de perfil é um padrão da UML, a maior parte das ferramentas CASE já está suportando a definição e importação de perfis. Outra vantagem deste mecanismo é que é possível aplicar mais de um perfil em um mesmo modelo. Além disso um Perfil UML pode ser facilmente modificado, com a introdução de novos estereótipos, valores rotulados e restrições. Esta modificação pode ser realizada em qualquer ferramenta CASE que suporte a importação e definição de perfis.

2.4.1.2 Extensão pela definição de um meta-modelo

Com meta-modelagem, o objetivo é estender o meta-modelo da UML no nível M2 com a adição de novos conceitos relacionados a um domínio de aplicação. Uma extensão neste nível modifica o meta-modelo, podendo adicionar e remover restrições, adicionar e remover meta-classes do modelo, adicionar e modificar relacionamentos, etc.

Este tipo de extensão não permite o reuso dos conceitos em qualquer ferramenta de modelagem, pois as ferramentas CASE suportam apenas a definição de modelos dentro do meta-modelo padrão da OMG ou modelos definidos com o uso de de perfis. Assim, a extensão será específica para uma determinada ferramenta. A extensão através de meta-modelagem pode ser utilizada quando uma extensão tem uma baixa probabilidade de ser modificada no futuro e não existe a necessidade de combinar esta extensão com outras extensões.

3 TRABALHOS RELACIONADOS

A POA tem elementos que não podem ser representados com o meta-modelo padrão da segunda versão da UML. Esta limitação faz com que seja necessário estender a linguagem para modelagem de sistemas orientados a aspectos. Esta extensão pode ser obtida com a definição de perfis UML ou através da definição de um novo meta-modelo. Propostas que estendem a UML através de um perfil podem ser reusadas diretamente em ferramentas CASE que suportem a importação de perfis. Os trabalhos que trabalham em nível de meta-modelagem podem ser utilizados em outras ferramentas CASE apenas através de transformações nos modelos, pois as ferramentas CASE suportam apenas o meta-modelo padrão da UML ou a extensão através de perfis.

Alguns trabalhos estendem a UML para modelagem de programas orientados a aspectos. (**??**) propõe um Perfil UML para AspectJ. O foco deste trabalho é o desenvolvimento de um perfil que estenda a UML através da introdução de estereótipos, valores rotulados e restrições que permitam representar programas orientados a aspectos. Este perfil não remove restrições do meta-modelo padrão da linguagem e também não adiciona novas meta-classes em nível de meta-modelo, apenas estende meta-classes através de estereótipos. Assim, é possível utilizar esse meta-modelo como um Perfil UML em qualquer ferramenta que suporte a importação de perfis.

Nesta proposta, os aspectos são agrupados em um interesse entrecortante. Para tal introduz-se o estereótipo *CrossCutingConcern* que estende a meta-classe *Package*. Um aspecto (*Aspect* é representado estendendo a meta-classe *Class*. Este pode conter características estruturais e dinâmicas. Um aviso representa comportamento e estende a meta-classe *BehavioralFeature*, significando que este aviso pode incluir colaborações e máquinas de estado. Os pontos de corte são características estruturais e são representados estendendo a classe do meta-modelo *StructuralFeature*. Para representar os possíveis tipos de ponto de corte estende-se a meta-classe abstrata *PointCut* em diversas sub-classes. Um aviso pode ter um ou mais pontos de corte associados. Os pontos de junção capturados por um ponto de corte devem ser elementos do modelo, como operações, atributos, etc. Um aspecto pode conter também declarações inter-tipos (*StaticCrossCutingFeatures*), uma construção que possibilita a introdução de novos membros e relacionamentos em classes existentes.

A figura 22 mostra o exemplo de um estereótipo que estende um elemento do meta-modelo da UML: o estereótipo *Aspect* estendendo a meta-classe *Class*. O nome do estereótipo está representado em negrito e a meta-classe a qual ele estende está representada entre colchetes. Um estereótipo pode ter relacionamentos com outros estereótipos, como composições, agregações e associações.

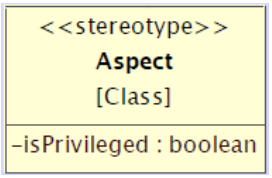

Figura 22 – Exemplo de estereótipo estendendo uma meta-classe em um diagrama de Perfil UML.

O Perfil UML completo proposto por (**??**) pode ser visualizado na figura 23. Como todas as construções são definidas em linguagem de meta-modelo, pode-se realizar a geração de código em AspectJ. É de responsabilidade do modelador elaborar uma modelagem que esteja de acordo com as construções da linguagem alvo para geração do código.

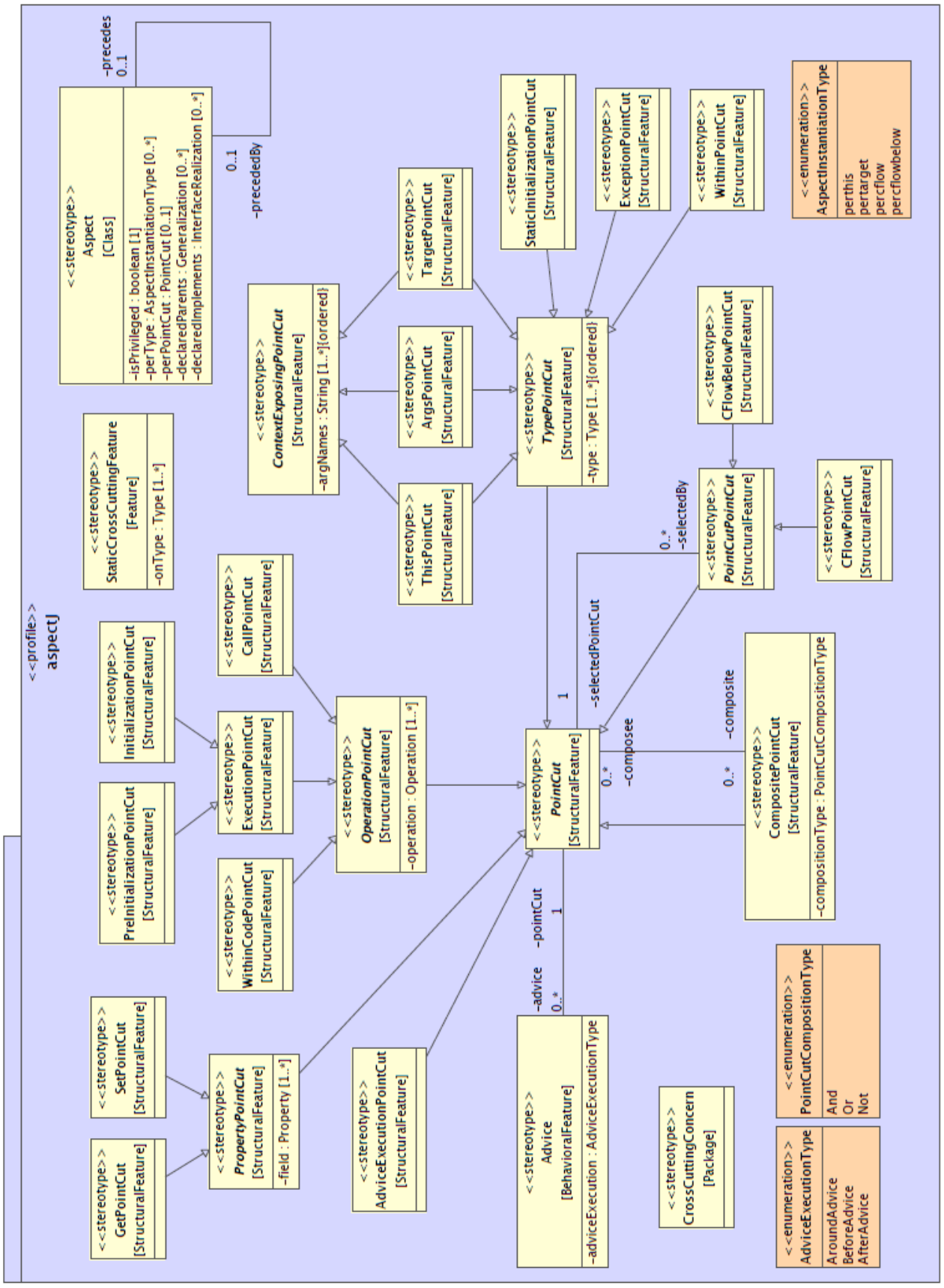

Figura 23 – Perfil UML para modelagem de aspectos

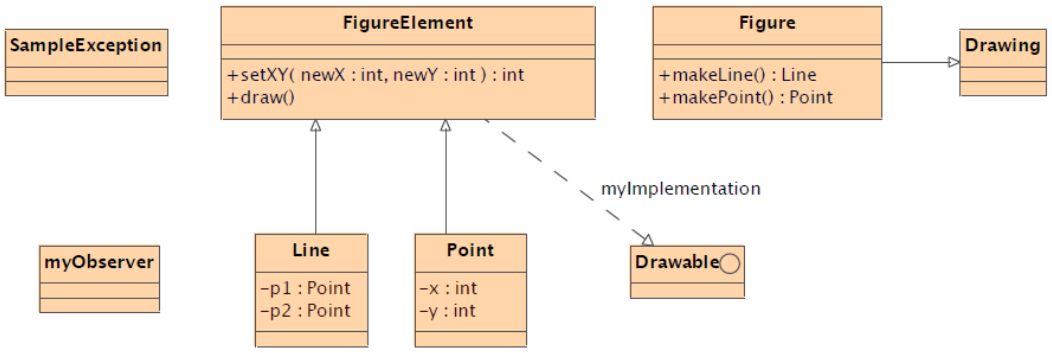

Figura 24 – Interesse núcleo para desenho de interface gráfica.

Para exemplificar o uso deste Perfil UML em uma modelagem, (**??**) implementou o padrão de projeto Observador em um sistema para desenho de interface gráfica. O diagrama de classes do interesse núcleo pode ser visualizado na figura 24.

O interesse entrecortante é o padrão de projeto Observador e a sua modelagem pode ser visualizada na figura 25. O modelo do interesse entrecortante contém um aspecto que especifica quatro pontos de corte, três introduções e dois avisos. O ponto de corte *setXYPointCut* captura modificações nos atributos x e y da classe *Point*. O ponto de corte *twoIntArgsPC* captura chamadas e execuções com dois argumentos do tipo inteiro. O ponto de corte *observePointPC* compõe os dois pontos de cortes anteriores com o operador *And*, capturando modificações nos atributos x e y da classe *Point* com dois argumentos do tipo inteiro. Finalmente, define-se o ponto de corte *makeLinePointCut* que captura chamadas ao método *makeLine()*. As introduções adicionadas são os métodos *addObserver()* e *removeObserver()* e o atributo *theObservers* nas classes *Line* e *Point*. Estes membros permitem adicionar, remover e armazenar os observadores. Para introduzir os novos comportamentos, especifica-se o aviso *pointChange* que é executado antes do ponto de corte *observePointPC* e o aviso *newLine* que é executado antes do ponto de corte *makeLinePointCut*. A composição do interesse entrecortante no interesse núcleo pode ser visualizada na figura 26. A composição é realizada em nível de código.

A principal contribuição deste trabalho é a especificação do meta-modelo apenas em termos da UML, sem a necessidade de descrições textuais e de ferramentas adicionais para interpretação ou geração de código a partir do modelo. Além disso, o meta-modelo pode ser utilizado como um perfil UML em ferramentas que suportem a importação de perfis, pois o mesmo está de acordo com o formato XMI padrão da OMG. Com a representação de elementos apenas em termos de meta-modelo, perde-se a possibilidade de representar padrões para captura de pontos de junção: os *wildcards*. A especificação por padrões é uma importante funcionalidade da programação por aspectos, pois simplifica a forma de capturar pontos de junção, sem a necessidade de explicitar cada elemento que será capturado. Assim, é importante que a extensão à UML permita representar essas características de uma maneira simples e praticável, para que seja possível expressar todas as características da POA na modelagem.

O trabalho de (**??**) permite representar a estrutura de um programa orientado a aspectos através das meta-classes *CrossCutingConcern*, *Aspect*, *PointCut* e *StaticCrossCutingFeatures*. É importante observar que apenas a parte estrutural é especificada por esta proposta. Em relação às características comportamentais, como a meta-classe *Advice* estende a meta-classe *BehavioralFeature*, é possível representá-las com colaborações e diagramas de máquinas de estados, mas o trabalho não demonstra como realizar a modelagem de colaborações e nem permite a composição automatizada entre modelos que representam aspectos dinâmicos do sistema. Assim, conclui-se que o foco do trabalho é a representação da estrutura de um sistema orientado a aspectos. A parte dinâmica deve ser implementada manualmente no código gerado.

Uma modelagem por múltiplos pontos de vista é proposta por (**??**). Propõe-se RAM (*Reusable Aspects Models*), uma abordagem para especificar aspectos com dois diagramas dinâmicos (diagrama de máquina de estados e de sequência) e um diagrama estrutural (diagrama de classes). O objetivo deste trabalho é a melhora da escalabilidade de um sistema, mantendo a consistência entre as diferentes visões de um interesse entrecortante. RAM define modelos base para representar interesses núcleo e de aspectos

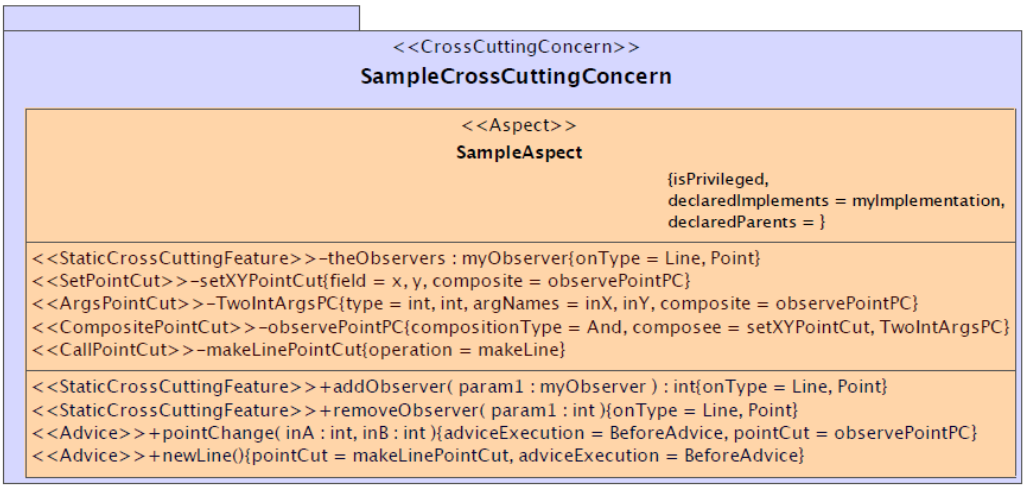

Figura 25 – Interesse entrecortante para implementação do padrão de projeto Observador.

```
package SampleCrossCuttingConcern;

privileged aspect SampleAspect {
    declare parents: Figure extends Drawing;
    declare parents: FigureElement implements Drawable;

    pointcut setXYPointCut (): (
        set(private int Point.x) ||
        set(private int Point.y));

   pointcut TwoIntArgsPC (int inX, int inY):
        args(inX, inY);

   pointcut observePointPC (int inX, int inY): (
        (set(private int Point.x) ||
         set(private int Point.y)) &&
        args(inX, inY));

   pointcut makeLinePointCut (): (
        call(public Line Figure.makeLine ()) );

   before(int inA, int inB) throws SampleException
                        : observePointPC(inA, inB ) {}
   before() : makeLinePointCut() {}

    private myObserver Line.theObservers;
    private myObserver Point.theObservers;
    public int Line.addObserver (myObserver param1) {};
    public int Point.addObserver (myObserver param1) {};
    public void Line.removeObserver (int param1) {};
    public void Point.removeObserver (int param1) {};
}
```

Figura 26 – Composição do padrão observador no sistema de interface gráfica (em nível de código).

para representar interesses entrecortantes.

Duas ferramentas são utilizadas para a composição dos modelos. *Kompose* (FLEUREY et al., 2007) é utilizado para composição da parte estrutural (diagramas de classe). Para composição, os elementos do modelo devem ser instâncias da mesma classe do meta-modelo. A composição é realizada comparando a assinatura de tipo dos elementos. Cada elemento do modelo deve possuir uma assinatura de tipo que o representa unicamente na modelagem. Dois elementos que tiverem a mesma assinatura podem ser compostos. A composição de diagramas de sequência e de estado é realizada utilizando uma outra ferramenta denominada GeKo (MORIN; KLEIN; BARAIS, 2008). Com essa ferramenta, para inserção de um novo comportamento em um modelo base deve-se definir um modelo de aspecto. Esse modelo de aspecto é composto por um diagrama refinando o ponto de corte e outro diagrama refinando o aviso. A composição acontece em duas fases: primeiramente são detectados os elementos do modelo base que são impactados pelo ponto de corte do modelo do aspecto. Com os elementos capturados executa-se um mecanismo de composição que gera o modelo composto com o comportamento (aviso) do modelo do aspecto inserido no modelo base (antes, durante ou depois). Essas duas ferramentas permitem representar aspectos e a composição entre um modelo base e um modelo de aspecto.

Um interesse em RAM é modelado através de um pacote UML. Este pacote contém três visões: estrutural, de estado e de mensagens e é denominado modelo de um aspecto, podendo ser reusado em diferentes aplicações. A visão estrutural é composta por diagrama de classes. Nesta visão, as classes não precisam ser completamente especificadas, pois só precisam expressar o que é relevante para o interesse em questão. A visão de estado descreve o protocolo de uso de uma classe. Para classes completas, utiliza-se o diagrama de máquina de estados da UML. Classes incompletas são modeladas com um diagrama de máquina de estados para aspectos, composto por uma parte representando o ponto de corte e outra representando o aviso. O ponto de corte determina quais estados devem existir para o aviso ser executado. A visão de troca de mensagens utiliza o diagrama de sequência, onde cada método público das classes modeladas deve ser representando. Aqui também pode-se especificar o comportamento de aspectos através de dois diagramas de sequência: um para o ponto de corte e outro para o aviso. O ponto de corte determina a sequência de mensagens que deve ocorrer pra ativar o aviso. O aviso descreve a sequência de mensagens que substituem o ponto de corte em uma execução. Nas três visões, alguns elementos podem estar incompletos, o que significa que eles não são especificados no modelo de aspecto e deverão ser especificados por algum outro modelo na composição de modelos. Esses elementos são denominados **parâmetros de instanciação mandatória**, identificados pelo prefixo — e modelados como parâmetros de *template* UML.

RAM permite estabelecer dependências entre aspectos, com o objetivo de possibilitar o reuso de modelos. Se um aspecto A depende de um aspecto B, A deve instanciar todos os parâmetros de instanciação mandatória de B através de **diretivas de instancialização**. Por exemplo, para uma classe incompleta em B, A deve especificar uma classe que possa completá-la com os métodos e atributos faltantes. Esta regra vale também para as visões de estado e de mensagens. Além disso, A pode definir **diretivas de ligação** que mapeiam entidades incompletas de A em entidades completas de B. Nesse caso, as entidades incompletas de A não podem ser parâmetros de instanciação mandatória. É importante observar que diretivas de ligação e parâmetros de instanciação mandatória podem ser definidos com *wildcards*, permitindo a captura de padrões, uma funcionalidade importante da programação por aspectos.

Com as dependências definidas entre aspectos, executa-se o algoritmo de composição que gerará um único modelo com os aspectos compostos. A figura 27 mostra um exemplo de composição entre três aspectos.

RAM também faz a verificação de consistências entre as diferentes visões e modelos de aspectos. São realizadas verificações em diferentes níveis:

- **No modelo de aspecto**: Pode-se verificar se existe um diagrama de máquina de estados para cada classe na visão estrutural.

- **Entre modelos de aspectos**: Pode-se verificar que um aspecto A que depende de B deve inicializar todos os parâmetros de instanciação mandatórios de B.

- **No modelo final**: Pode-se comparar a sequência de mensagens na visão de mensagens com o diagrama de máquina de estados. As mensagens devem obedecer o protocolo do diagrama de máquina de estados.

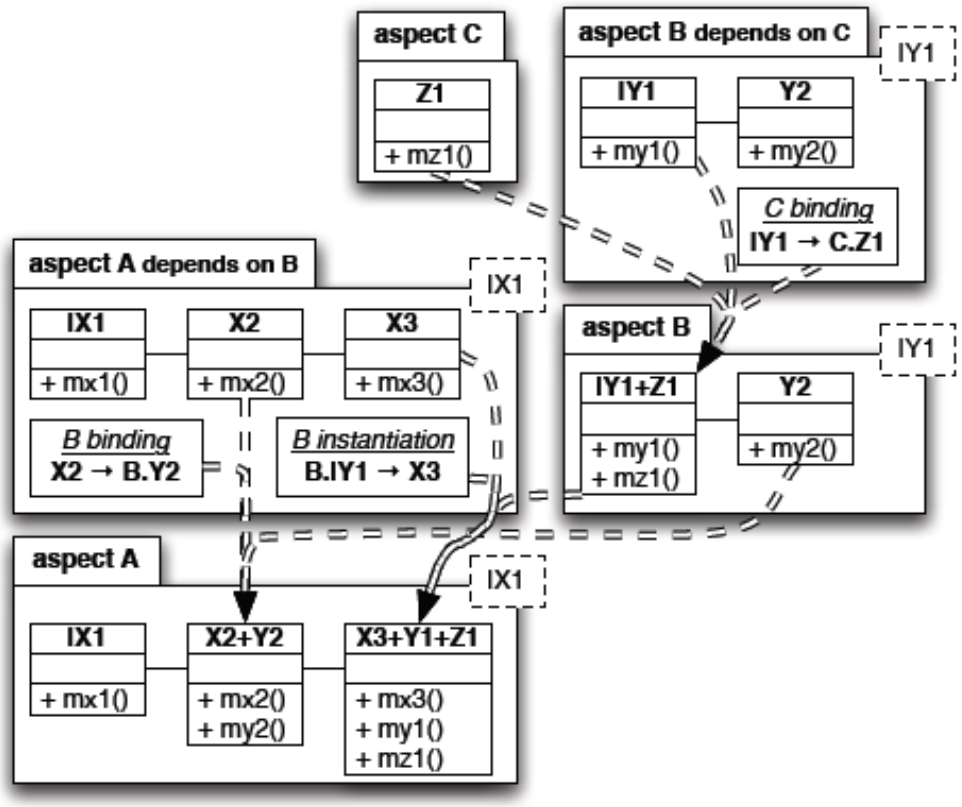

Figura 27 – Composição da parte estrutural de aspectos na ferramenta RAM

Um estudo de caso foi realizado para avaliar a modelagem por múltiplos pontos de vista. No estudo adiciona-se a funcionalidade de garantia de atomicidade em um modelo de transações. Para garantir a atomicidade de transações utiliza-se o aspecto *Recovering* que tem dependência com nove aspectos. Este aspecto pode ser visualizado na figura 28. Uma das dependências indiretas é o aspecto *Checkpointable*, que pode ser visualizado na figura 29 (*Recovering* depende de *Checkpointing* que depende de *Checkpointable*). Este aspecto permite estabelecer pontos de verificação de um objeto, armazenando o seu estado e permitindo a restauração do mesmo. O aspecto *Checkpointable* depende de um outro aspecto denominado *Copyable* e ele deve instancializar os elementos incompletos de *Copyable*. Um exemplo de instancialização pode ser visualizado na visão de mensagens do aspecto *Checkpointable*. A diretiva de instancialização *clone.ICaller-ICheckpointable* indica que o objeto *ICaller* da visão de mensagens do método *clone* em *Copyable* está sendo instancializado com o objeto *ICheckpointable* de *Checkpointable*. O aspecto *Recovering* também contém diretivas de instancialização para os elementos incompletos dos aspectos os quais depende.

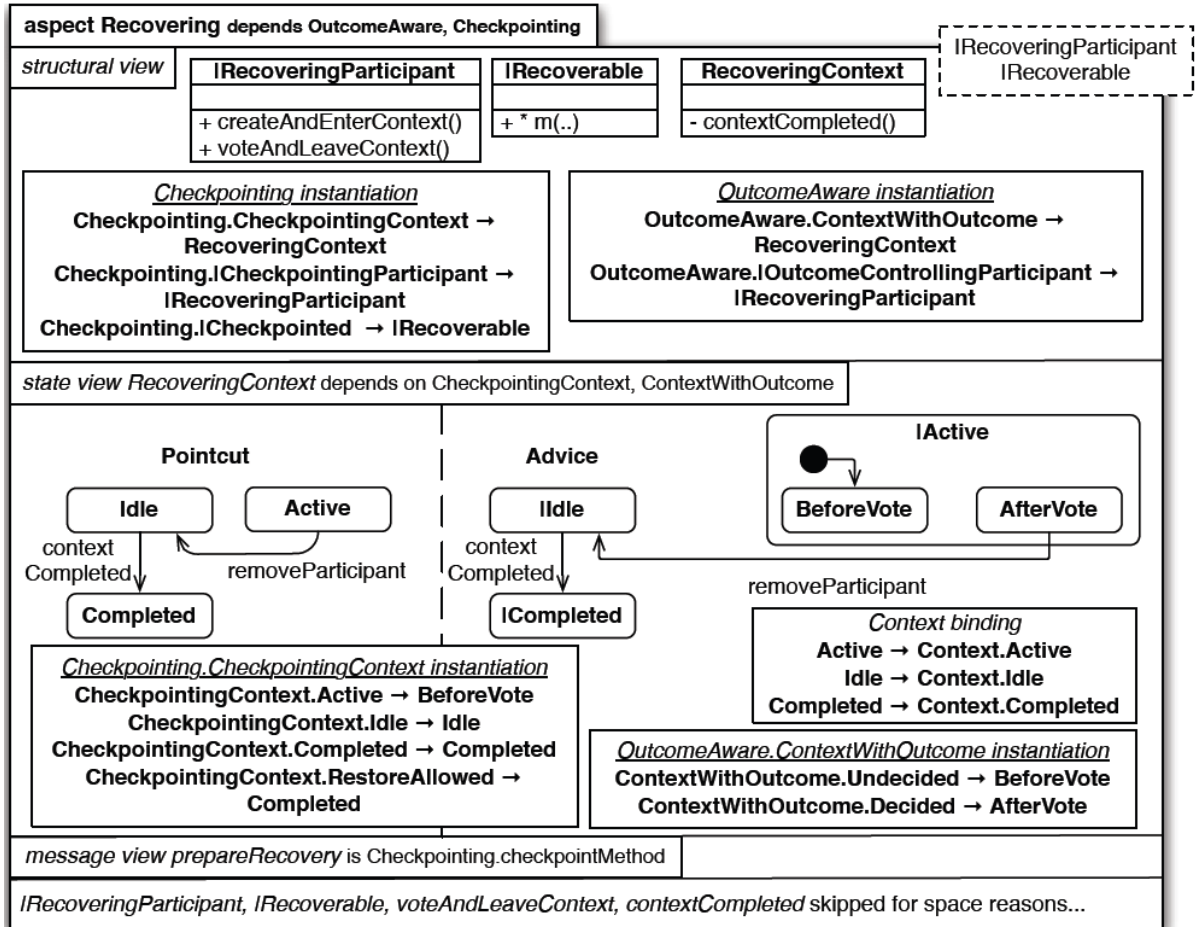

Figura 28 – Aspecto para Recuperação (Garantia de Atomicidade)

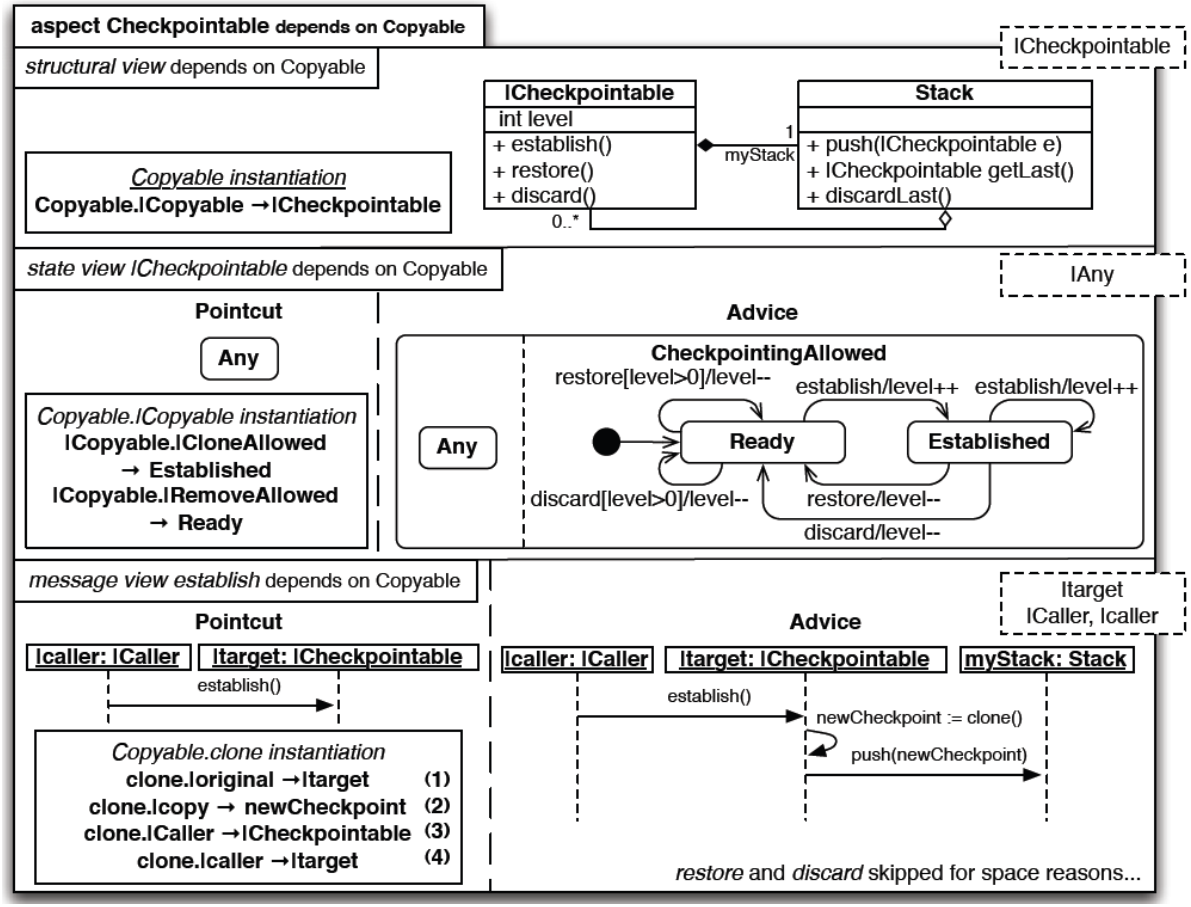

Figura 29 – Modelo de Aspecto para Pontos de Verificação

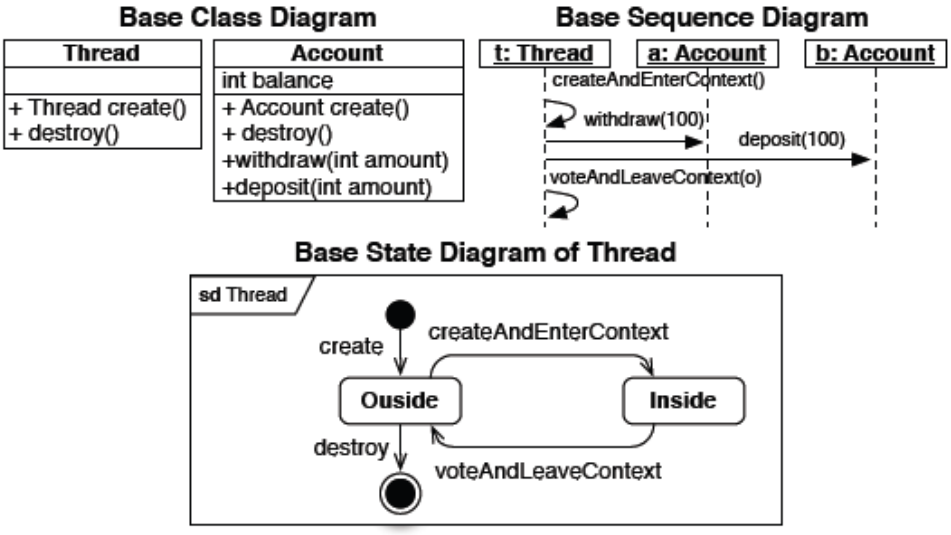

Figura 30 – Modelo Base para Transações

Com o aspecto para garantia de atomicidade definido, deve-se realizar a composição com o modelo base. O modelo base é uma aplicação para realização de transações bancárias e pode ser visualizado na figura 30. O modelo final após a composição pode ser visualizado nas figuras 31 (visão estrutural), 33 (visão de estados) e 32 (visão de mensagens). Observa-se a presença de alguns elementos referentes ao aspecto *Checkpointable* no modelo final como a classe *Stack* e a inserção dos métodos *establish(), restore()* e *discard()* na visão de mensagens. Na visão de troca de mensagens, as mensagens referentes ao aspecto *Checkpointable* também estão destacadas. Nesta visão, as mensagens de cada aspecto dependente estão destacadas. Assim, o modelo final contém uma parte de cada aspecto, compostas com o modelo base, resultando em um modelo que representa o comportamento desejado para garantia de atomicidade de transações.

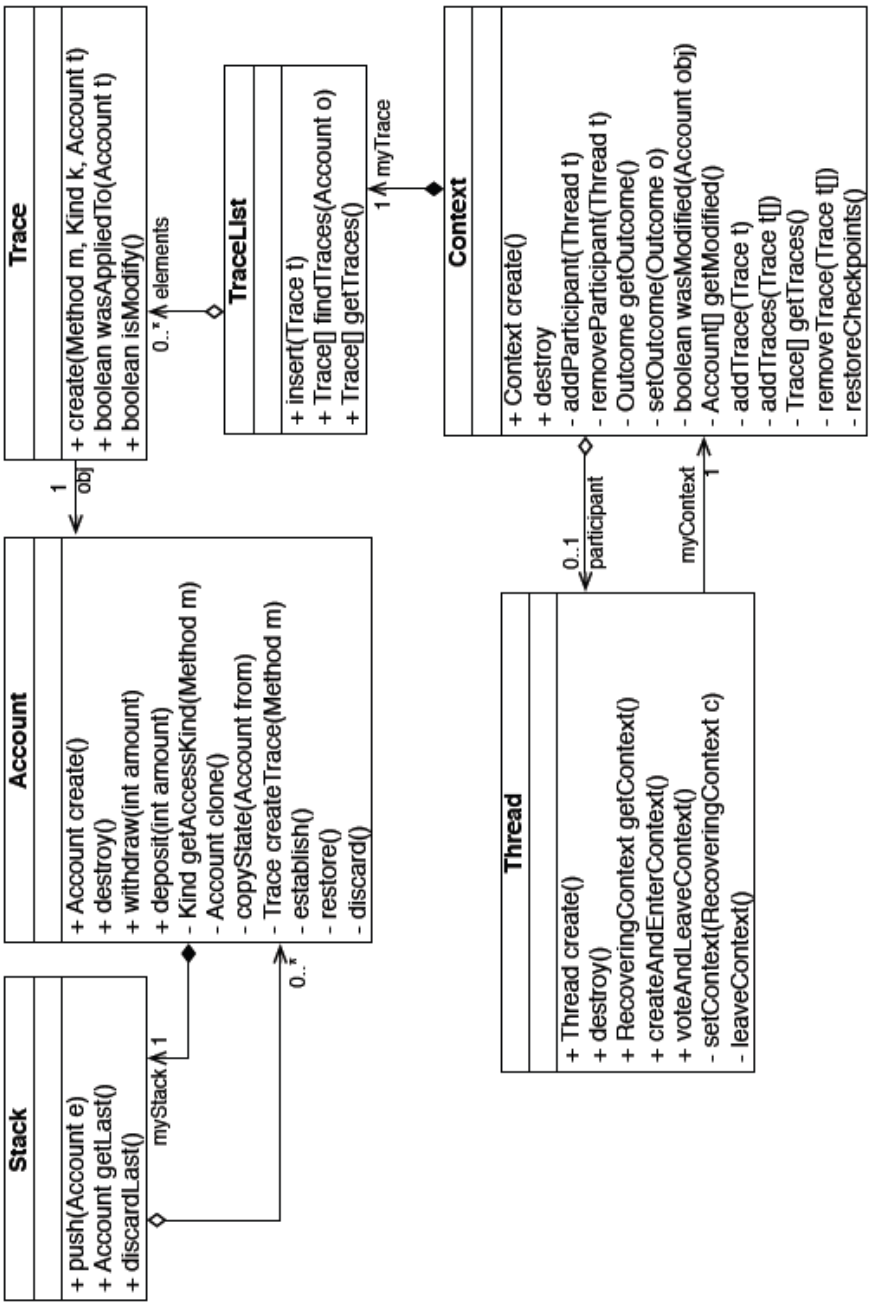

Figura 31 – Visão Estrutural do Modelo Final para Garantia de Atomicidade de Transações

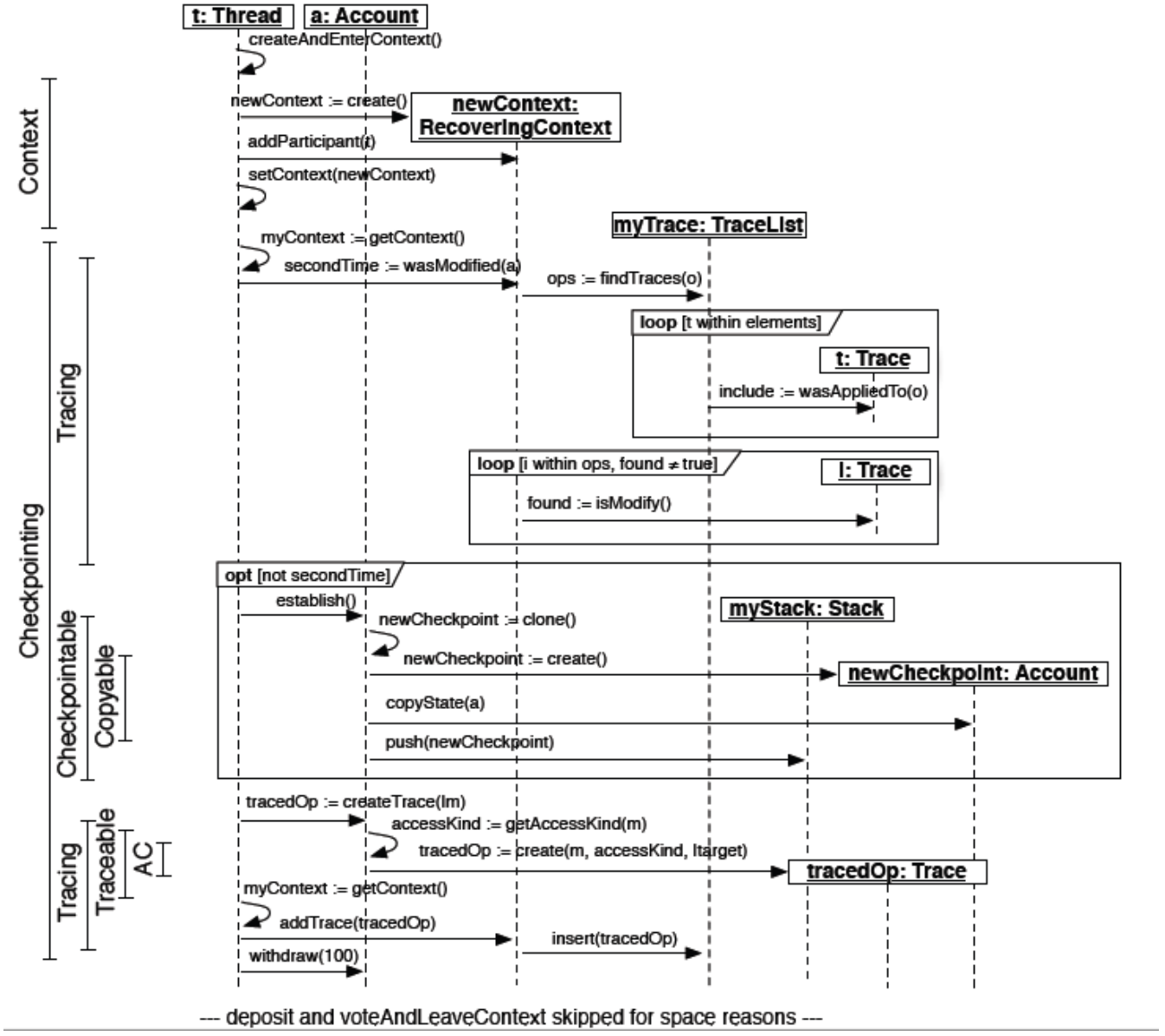

Figura 32 – Visão de Mensagens do Modelo Final para Garantia de Atomicidade de Transações

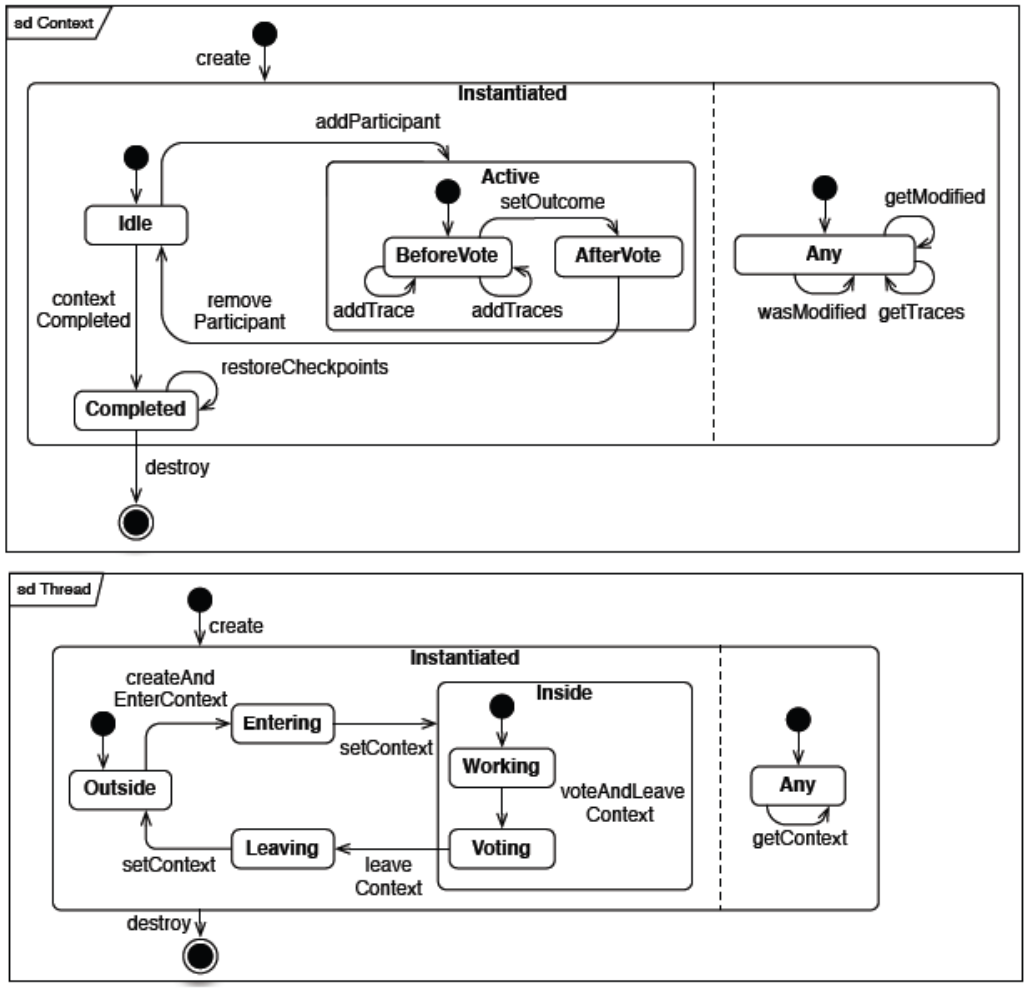

Figura 33 – Visão de Estados do Modelo Final para Garantia de Atomicidade de Transações

A abordagem RAM permite representar a estrutura de ums sistema através da visão estrutural e a dinâmica do sistema através das visões de estado e de troca de mensagens. A maior contribuição deste trabalho é o foco na representação das caracaterísticas dinâmicas de um sistema, utilizando diagramas de máquinas de estado para representar o protocolo de um aspecto e diagramas de sequência para representar as possíveis interações entre objetos. Por isso, uma modelagem criada com a ferramenta pode ser considerada completa. A especificação de pontos de corte com padrões (*wildcards*) também é suportada pela ferramenta, permitindo assim o uso de uma importante funcionalidade da programação orientada a aspectos. Outra característica importante de RAM é a garantia de consistência entre as diferentes visões e a orientação ao reuso de aspectos através da definição de dependências. Uma limitação da proposta é que nem todos os diagramas podem ser elaborados apenas estendendo a UML, pois a abordagem realiza modificações no meta-modelo da UML, introduzindo novos conceitos que não estão presentes na versão padrão da OMG. Exemplos são as diretivas de instancialização e as diretivas de ligação. Além disso, são necessárias ferramentas adicionais para realizar a composição entre aspectos e a verificação de dependências (Kompose e GeKo), o que é algo plausível, pois hoje nenhuma ferramenta CASE comercial suporta composição de aspectos em nível de modelo.

A complexidade do modelo composto também pode ser considerada uma limitação da abordagem, pois o modelo gerado é extenso, o que dificulta a compreensão, já que existem muitas dependências indiretas entre aspectos e muitos elementos sintáticos na modelagem. Segundo (FARRINGTON, 2011), quando uma pessoa entra em contato com novas informações, o limite de sua memória de trabalho é entre três ou quatro elementos de informação. A modelagem proposta por (**??**) é difícil de ser compreendida, pois o modelo final (composto) contém muitos elementos de informação. No exemplo de composição de aspectos, o modelo final contém elementos que vieram de nove diferentes aspectos. Além disso, contém elementos que não são padrões na modelagem de sistemas com UML, como as diretivas de instancialização e de ligação. Para facilitar a compreensão do comportamento dinâmico de uma modelagem, este trabalho poderia disponibilizar um visualizador de aspectos, que permitisse habilitar e desabilitar modelos de aspectos dinamicamente em um modelo base.

O trabalho de (**??**) propõe uma solução para modelagem de aspectos no domínio de requisitos relacionados a segurança. Esta abordagem engloba desde a especificação de requisitos de segurança até a composição dos aspectos. O objetivo é propor um mecanismo elegante para injeção de soluções de segurança, aonde os interesses de segurança ficarão separados em módulos, evitando o emarranhamento com os demais interesses do sistema. A composição entre interesses de segurança e interesses núcleo é realizada em nível de modelo. A abordagem é composta por quatro fases:

1. **Especificação dos requisitos das soluções de segurança**: Especialistas na área de segurança especificam quais são os requisitos que devem ser implementados no sistema.

2. **Modelagem das soluções de segurança com um aspecto para cada requisito**: O modelo do aspecto é elaborado utilizando um Perfil UML proposto para área de segurança. Este Perfil UML é semelhante ao perfil proposto por (**??**) e permite representar a estrutura dos aspectos e pontos de corte.

3. **Definição de quais partes do sistema serão impactadas pelos aspectos de segurança**: Devem-se definir quais pontos de junção serão capturados. Uma das contribuições deste trabalho é a identificação dos pontos de junção em alguns diagramas da UML: diagrama de atividades, diagrama de sequência e diagrama de máquina de estados. A figura 34 mostra a identificação de pontos de junção para os principais diagramas comportamentais da UML.

4. **Composição entre os modelos de aspecto e o modelo base do sistema**: Primeiramente devem-se selecionar manualmente os métodos do modelo base que terão o comportamento de segurança (aviso) introduzido antes, durante ou depois. Além disso, cada aviso já tem um ponto de corte associado que foi especificado pelo especialista de segurança. Após a seleção manual, os pontos de junção e os pontos de corte especificados pelos especialistas são identificados no modelo base e associados com os avisos correspondentes. É importante observar que o desenvolvedor não necessita ser especialista em soluções de segurança, pois ele apenas especifica em qual método do modelo base o novo comportamento será inserido. Mas, o especialista em segurança precisa conhecer o modelo base ao definir os pontos de corte, o que gera uma incoerência, pois uma das afirmações do trabalho é que os especialistas especificam bibliotecas independentemente do modelo base. O

Join Point	Activity Diagrams	Sequence Diagrams	State Machine Diagrams
Operation Call	*Call-Operation-Action*	*SendOperationEvent*	Call operations inside states and transition effects
Operation Execution	Execution of the behavior invoked by call actions	*Execution-Specification*	Execution of the behavior invoked by call operations inside states and transition effects
Object Creation	*CreateObject-Action*	*Creation-Event*	None
Object Destruction	*DestroyObject-Action*	*Destruction-Event*	None
Field Reference	*ReadStructural-FeatureAction*	None	None
Field Assignment	*WriteStructural-FeatureAction*	None	None
Exception Handler	*Exception-Handler*	None	None

Figura 34 – Definição de Pontos de Junção em Diagramas Comportamentais da UML

passo final é a composição dos modelos, resultando em um modelo final com o comportamento dos avisos introduzido nos pontos especificados. No modelo final, aparecem referências aos modelos dos avisos (ao invés do comportamento completo), diminuindo o número de elementos presentes em um diagrama a fim de facilitar a visualização e diminuir a complexidade.

Um estudo de caso foi realizado utilizando a abordagem proposta por (**??**). O modelo base é uma aplicação que processa requisições de usuário a um servidor através de um canal de comunicação. A requisição somente será respondida se as credenciais do usuário estiverem corretas. O modelo de aspecto introduzirá comunicação segura entre o cliente e o servidor através do uso de Secure Sockets Layer (SSL). O modelo base pode ser visualizado na figura 35 e o modelo de aspecto está descrito na figura 36. Após a composição, é obtido o modelo final que pode ser visualizado na figura 37.

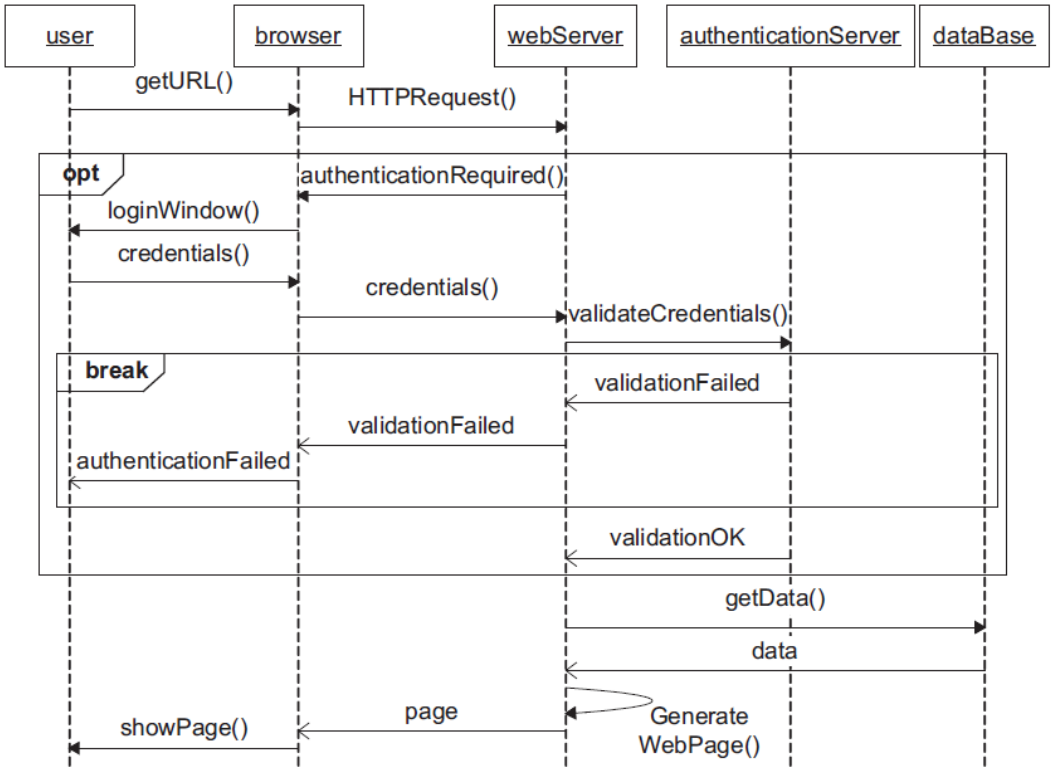

Figura 35 – Modelo Base

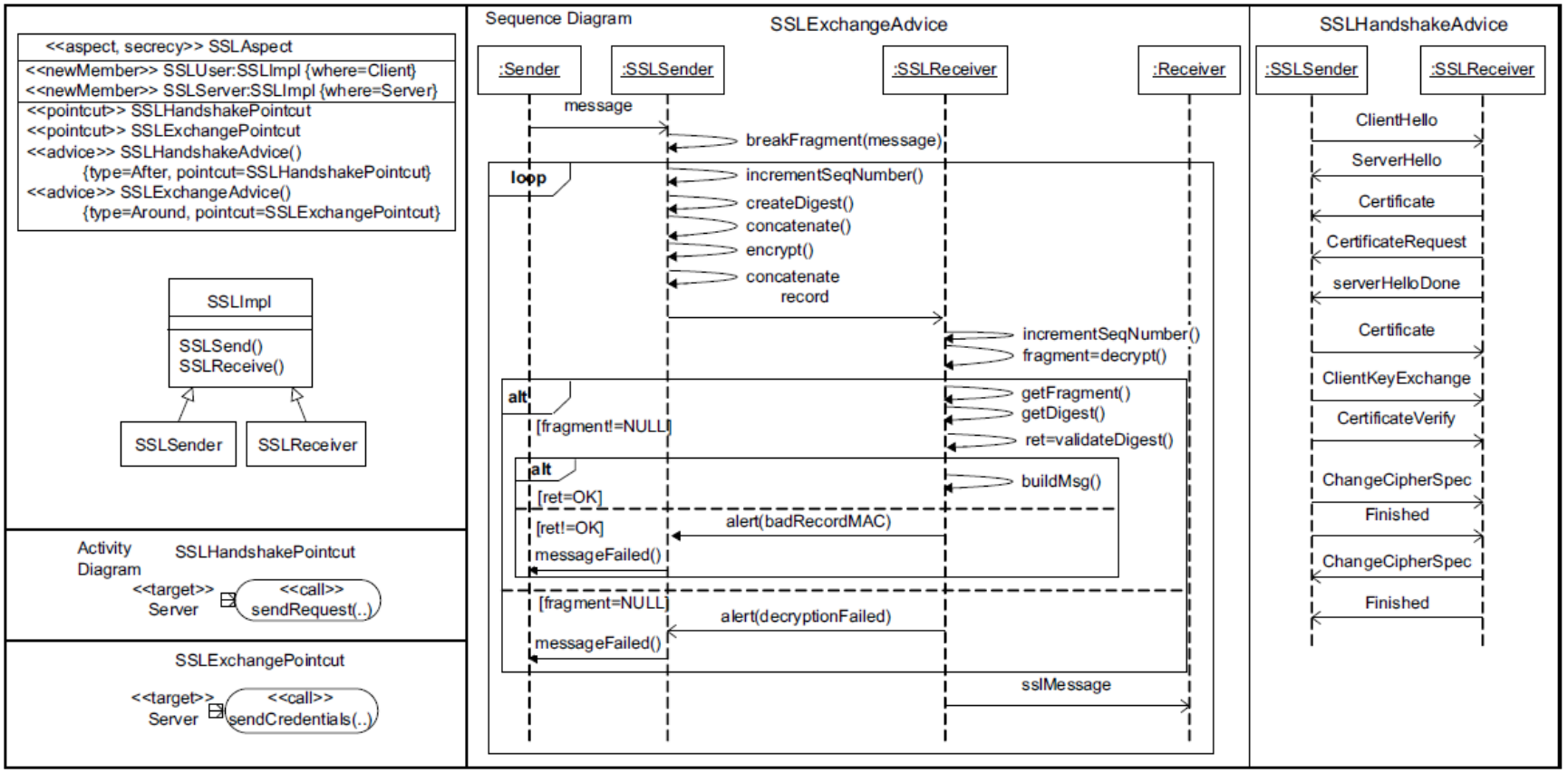

Figura 36 – Modelo de Aspecto SSL

A proposta para modelagem de aspectos de requisitos de segurança de (**??**) é uma leve extensão à segunda versão da UML. Uma parte da extensão é realizada com a definição de um Perfil UML para representar a estrutura dos aspectos através do diagrama de classes estendido, permitindo representar a estrutura do sistema. O perfil proposto pode ser reutilizado em ferramentas CASE que suportem a importação de perfis. No entanto, este perfil não permite representar todas as características estruturais da POA. Não é posível especificar a captura de pontos de junção de inicialização e pré-inicialização de classes e nem o tratamento de exceções. O perfil de (**??**) é mais completo do que o proposto por (**??**), sendo o recomendado para representação da estrutura de POA. Além disso, esta proposta não permite representar a composição de pontos de corte através dos operadores ! (negação), && (adição) e ||(opção). A parte dinâmica do sistema é representada com o diagrama de sequência, utilizado também para especificação do comportamento do aspecto. A proposta disponibiliza ferramentas para composição dos modelos de interesses entrecortantes com os modelos de intereses núcleo e realiza a composição em nível de modelo. Uma das limitações da abordagem é o foco apenas em sistemas de segurança. O exemplo de uso da abordagem é simples e não utiliza algumas construções da POA como *wildcards* e dependência entre aspectos. Seria interessante realizar um caso de uso utilizando a abordagem em um exemplo mais complexo, como o que foi proposto pelo trabalho de (**??**) e pelo trabalho de (**??**).

O trabalho de (**??**) modela a dinâmica de um sistema com diagramas de atividades da segunda versão da UML. Os interesses núcleo são modelados com a versão pura da UML. No caso dos interesses entrecortantes, é proposta uma extensão ao diagrama de atividades com estereótipos e valores rotulados. Um interesse entrecortante é representado por dois modelos: um para o ponto de corte e outro para o aviso. Adiciona-se o estereótipo *Pointcut* para indicar que um diagrama de atividades representa um **modelo de ponto de corte**. O estereótipo *Pointcut* tem um valor rotulado denominado *advice* que aponta para o nome do modelo de aviso associado a este ponto de corte. Para representar os pontos de execução de um programa que serão capturados adiciona-se o estereótipo *Joinpoint*. Esta abordagem permite a seleção de pontos de junção com o uso de *wildcards*. Um elemento no modelo de ponto de corte estereotipado com *Argument* define os argumentos que serão passados para o modelo de aviso. O estereótipo *Argument* tem um valor rotulado denominado *parameter* que define o nome dos parâmetros que serão preenchidos no modelo de aviso. O estereótipo *Advice* é adicionado para especificar que um diagrama de atividades representa um **modelo de aviso**. O valor rotulado *type* está associado com este estereótipo indicando o tipo de aviso: antes ou depois. O estereótipo *Parameter* é adicionado para representar os parâmetros que são aceitos pelo modelo de aviso. Estes parâmetros são utilizados para possibilitar o reuso de avisos, apenas modificando os parâmetros passados na inicialização. Finalmente, adicionam-se dois estereótipos *Entry* e *Exit* para especificar o início e o fim de uma execução de um modelo de aviso.

A figura 38 mostra o modelo base de um caso de uso para realizar a transação bancária de saque. Este modelo será estendido utilizando a abordagem proposta por (**??**) com a definição de dois pontos de corte e dois avisos. Os modelos do primeiro ponto de corte e do primeiro aviso podem ser visualizados na figura 39. O ponto de corte *Pointcut1* seleciona os elementos no modelo base aonde o aviso de autorização deve ser aplicado. O aviso *Advice1* representa o comportamento de autorização. A figura 40 mostra o ponto de corte *Pointcut2* que pretende capturar os elementos no modelo base aonde o aviso de envio de e-mail será aplicado. O aviso *Advice2* representa o comportamento de enviar um e-mail. Observa-se uma diferença no comportamento dos dois avisos: a autorização é realizada antes (aviso do tipo *before*) da execução dos pontos de junção selecionados pelo primeiro ponto de corte. No entanto, o envio de e-mail é realizado depois (aviso do tipo *after*) dos pontos de junção selecionados pelo segundo ponto de corte, e, o envio de e-mail é realizado paralelamente ao comportamento dos pontos de junção. A opção de executar o aviso paralelamente aos pontos de junção é uma opção dos autores desta proposta para avisos do tipo depois *after*.

Após a definição do modelo base e dos modelos para os interesses entrecortantes, deve-se realizar a composição entre os modelos. Esta composição consiste em três passos:

- **Captura**: Encontrar os pontos de junção no modelo base.
- **Inicialização**: Inicializar os modelos de aspectos com os parâmetros obtidos do modelo base.
- **Composição**: Realizar a composição entre os modelos de aspectos e o modelo base.

Após a composição, obtém-se o modelo final com os interesses entrecortantes introduzidos. O modelo final pode ser visualizado na figura 41.

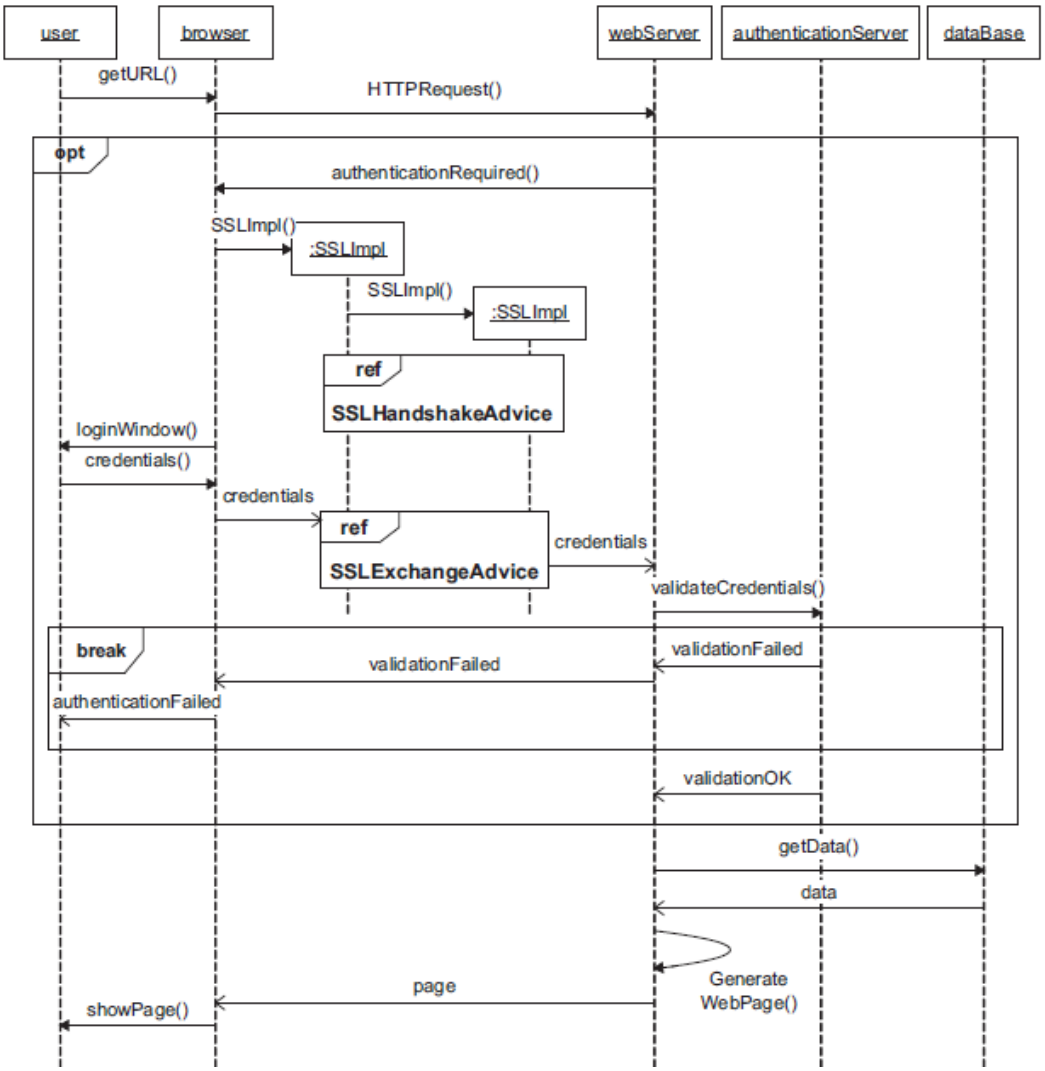

Figura 37 – Modelo Final (Composto)

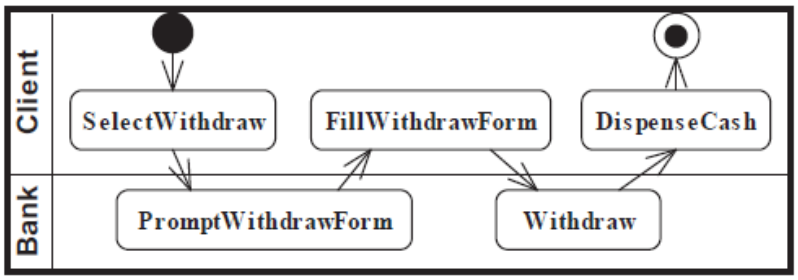

Figura 38 – Modelo base para realizar a transação de saque

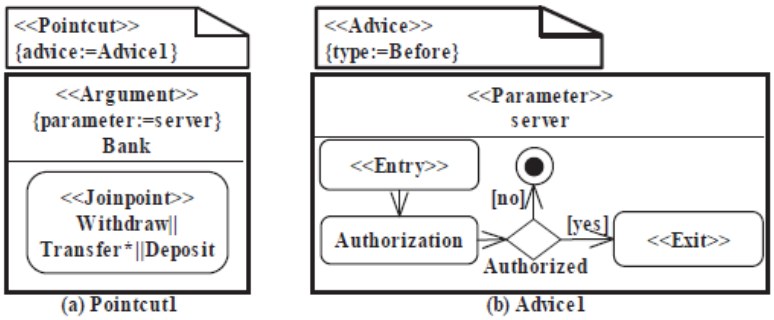

Figura 39 – Ponto de corte e aviso para capturar pontos de junção que necessitam de autorização antes de serem executados.

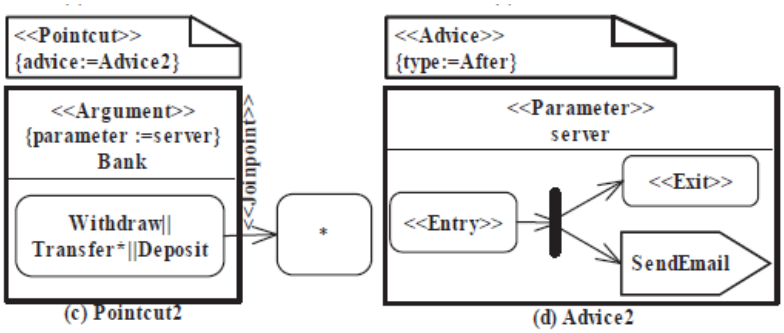

Figura 40 – Ponto de corte para capturar pontos de junção onde é necessário enviar um e-mail após a execução dos mesmos.

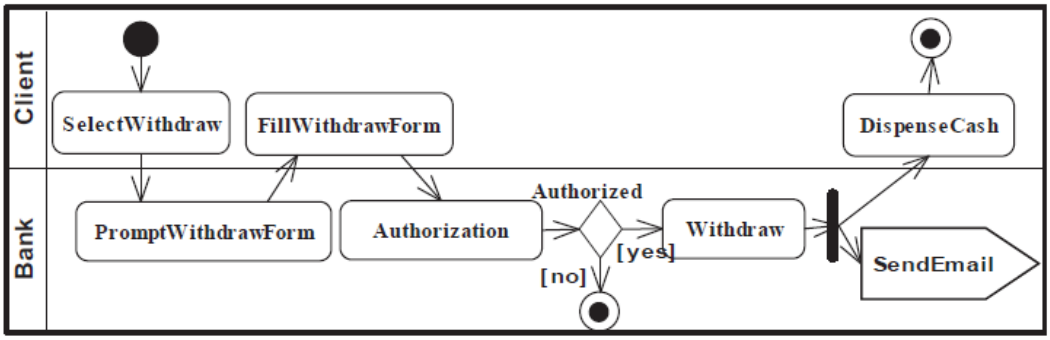

Figura 41 – Modelo composto para transação bancária de saque com autorização e envio de e-mail.

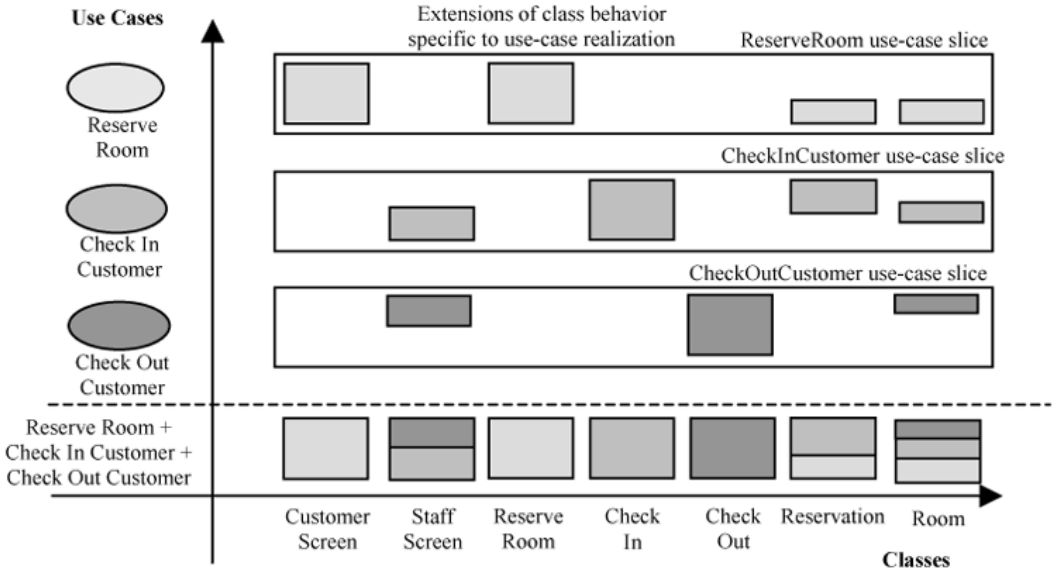

Figura 42 – Fatias de casos de uso em um sistema de gerenciamento de hotel.

A proposta de (**??**) é uma extensão à segunda versão da UML, estendendo apenas o diagrama de atividades com a adição de novos estereótipos e valores rotulados para representar interesses entrecortantes. A extensão segue o padrão do meta-modelo definido pela OMG para a segunda versão da UML e pode ser utilizada em qualquer ferramenta que suporte a definição de perfis. É possível realizar a composição entre modelos de interesses núcleo e modelos de interesses entrecortantes e a mesma é realizada em nível de modelo. Em relação a POA, a abordagem não permite representar as principais características do paradigma. Não é possível adicionar introduções a classes e interfaces nos modelos de interesses núcleo. Também não é possível representar todos os tipos de ponto de junção e nem adicionar avisos do tipo *around*, que permite substituir a execução de um ponto de junção ou modificar o comportamento de execução do mesmo. Em relação as visões representadas pela modelagem, a parte estrutural não pode ser representada apenas com diagrama de atividades e a abordagem não utiliza nenhum diagrama estrutural para representar o sistema. Assim, esta proposta representa apenas a parte dinâmica de um sistema orientado a aspectos. A principal contribuição desta proposta é uma extensão leve e simples aos diagramas de atividades da UML para representação da dinâmica de aspectos em alto nível de abstração.

O trabalho de (**??**) utiliza casos de uso para modelagem de interesses entrecortantes. São identificados dois tipos de caso de uso:

- *Peer Use Cases*: São casos de uso que não tem relacionamentos com outros casos de uso, mas sua implementação impacta mais de uma classe. São os interesses núcleo de um sistema.

- *Extension Use Cases*: São casos de uso que estendem o comportamento de um caso de uso base. Tem relacionamentos com outros casos de uso e sua implementação pode impactar mais de uma classe. São os interesses entrecortantes de um sistema.

Propõe-se uma construção denominada **fatia de caso de uso** (*use-case slice*) que deve modelar apenas as especificidades de um caso de uso. Para tal adiciona-se o estereótipo *use-case slice* ao meta-modelo da UML. Uma fatia de caso de uso contém: definições de novas classes necessárias para realizar o caso de uso, extensões a classes existentes (apenas a extensão será representada na fatia do caso de uso) e colaborações para representar a realização do caso de uso. O gráfico da figura 42 mostra o impacto dos casos de uso nas classes de um sistema de gerenciamento de hotel. Observa-se que os casos de uso *Reserve Room*, *Check In Customer* e *Check Out Customer* modificam a mesma classe *Room*. Cada fatia de caso de uso define uma parte desta classe. A classe completa é obtida realizando a composição das fatias de caso de uso.

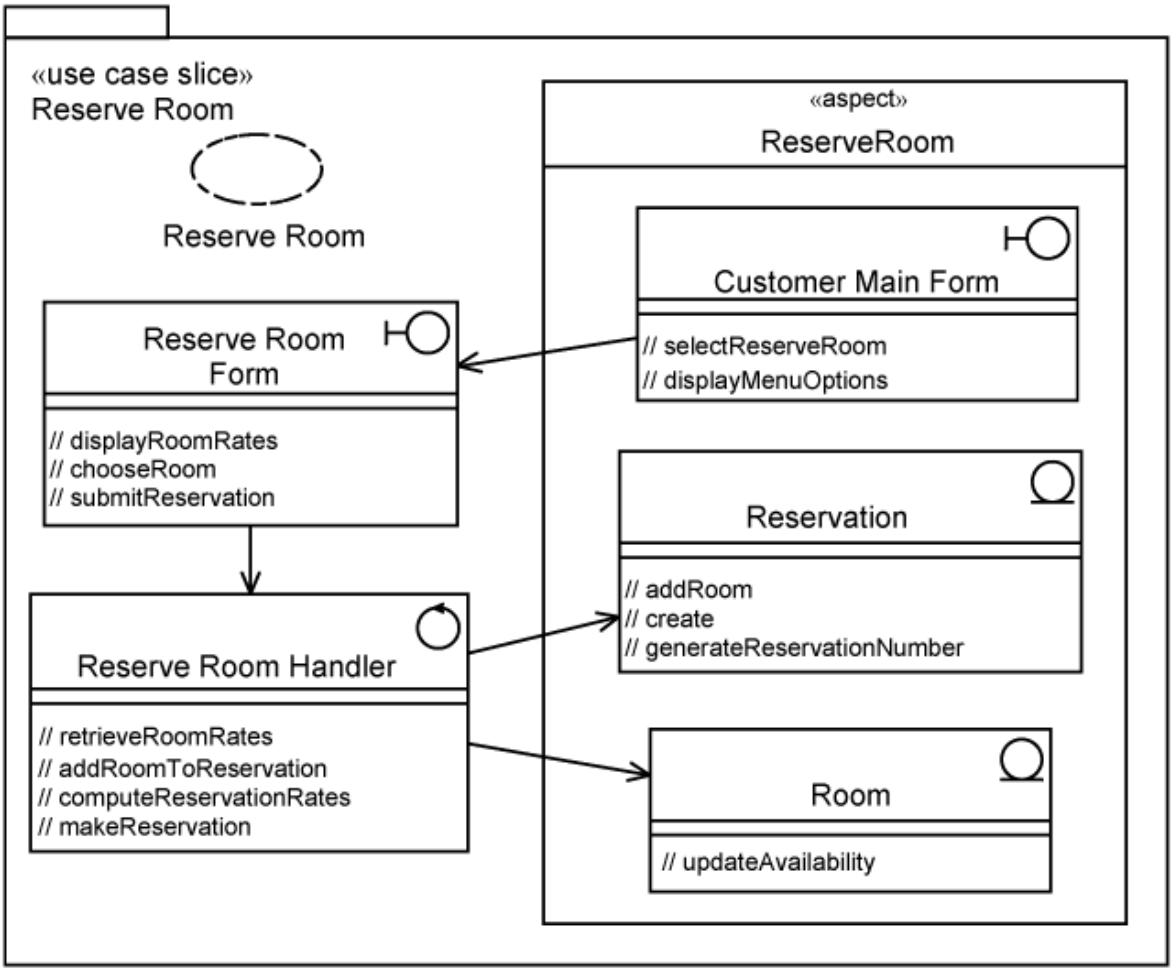

Figura 43 – Fatia de caso de uso para o caso de uso *Reserve Room*.

A figura 43 mostra a fatia do caso de uso *Reserve Room*. Observa-se nesta figura um aspecto representado pelo estereótipo *aspect*. Dentro deste aspecto definem-se extensões à três classes do modelo: *CustomerMainForm*, *Reservation* e *Room*. A representação de extensões à classes é equivalente às introduções da linguagem AspectJ. Esta fatia de caso de uso define também duas novas classes: *ReserveRoomForm* e *ReserveRoomHandler*. Este caso de uso é do tipo *Peer Use Case*, pois ele não estende nenhum caso de uso, apenas define novas classes e introduz métodos em classes já existentes.

Os casos de uso de extensão (*Extension Use Cases*) estendem o comportamento de um caso de uso base. A figura 44 mostra o caso de uso *Handle Waiting List* que estende o caso de uso *Reserve Room*. Um caso de uso base define um conjunto de **pontos de extensão** representando os pontos que podem ser estendidos por outros casos de uso. Um caso de uso de extensão define **pontos de corte de extensão** para representar quais pontos de extensão do caso de uso base serão estendidos.

A fatia de caso de uso da figura 45 mostra a modelagem do caso de uso de extensão *Handle Waiting List*, que adiciona o cliente em uma lista de espera se não for possível reservar um quarto. Para tal, o caso de uso de extensão define o ponto de corte *updatingRoomAvailability* para capturar as chamadas ao método *UpdateAvailability()* da classe *Room*. Dentro do aspecto define-se uma extensão à classe *ReserveRoomHandler*, adicionando o método *makeReservation()*. Este método será executado depois do ponto de corte *updatingRoomAvailability* quando forem lançadas as exceções *NoRoomAvailable* ou *QueueForRooms*. Além disso, são introduzidos dois novos métodos à classe *Reservation* relativos a reserva de quartos. Esta fatia de caso de uso cria duas novas classes: *WaitingListHandler* e *WaitingList*.

A proposta de (**??**) propõe a definição de **modelos de caso de uso** para agrupar as fatias de caso de uso em diferentes níveis de abstração. Define-se o estereótipo *use-case module* no meta-modelo.

Figura 44 – Caso de uso de extensão *Handle Waiting List*.

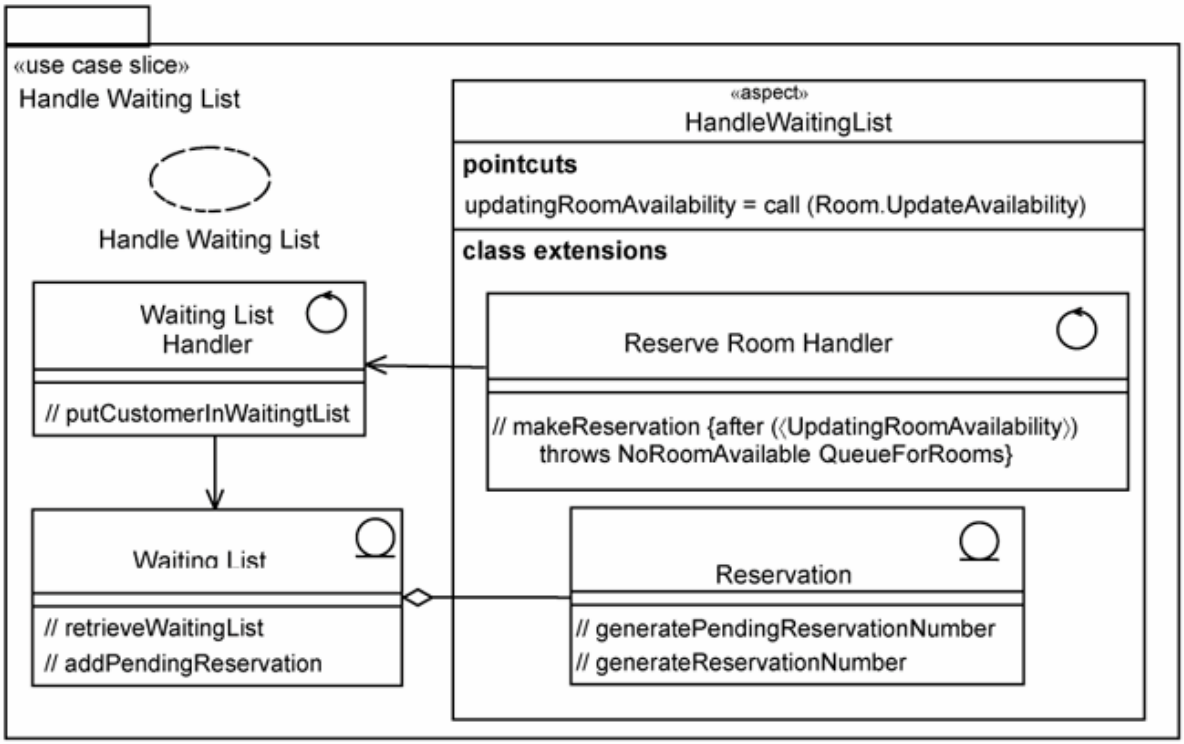

Figura 45 – Fatia do caso de uso de extensão *Handle Waiting List*.

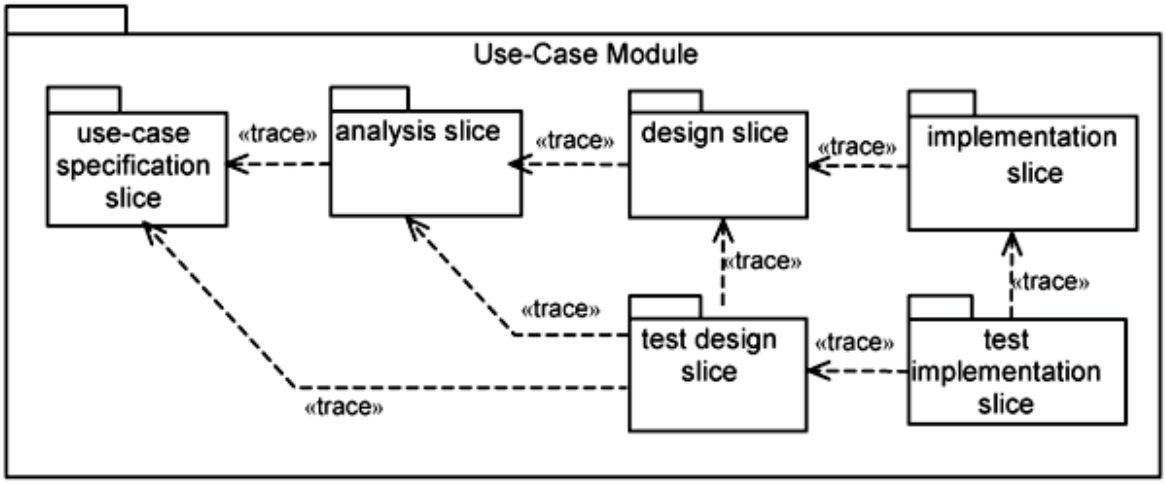

Figura 46 – Rastreamento completo de um requisito por todos modelos.

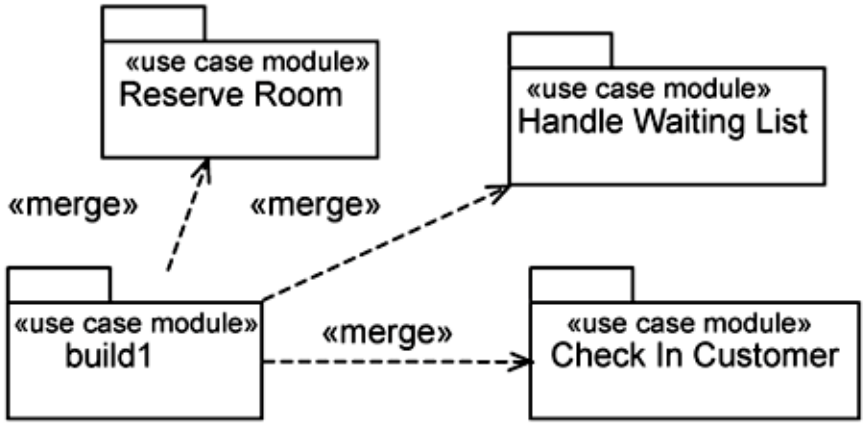

Figura 47 – Composição entre modelos de caso de uso.

São propostos quatro níveis de modelo: caso de uso, análise, projeto e implementação. A fatia de caso de uso será refinada de acordo com o nível de abstração. Esta proposta consegue obter o rastreamento completo do requisito para o código que o implementa, passando pelos modelos de caso de uso, análise e projeto, como pode ser visualizado na figura 46.

Após a definição das fatias de caso de uso em diferentes níveis de abstração, deve-se compor os modelos de caso de uso para se obter um sistema completo com o comportamento de cada caso de uso inserido nas respectivas classes. A figura 47 mostra a composição de três modelos de casos de uso (cada modelo representado como um pacote UML) para gerar uma construção válida do sistema. Para tal utiliza-se o estereótipo *merge* que representa dependência entre pacotes.

A proposta de (**??**) representa as principais características da POA. Permite representar *wildcards* na definição de pontos de corte dentro de uma fatia de caso de uso. Uma contribuição importante deste trabalho é a possibilidade de obter o rastreamento completo de um requisito até o código que o implementa. Segundo (WIMMER et al., 2011), o mapeamento de requisitos entre as diferentes fases do desenvolvimento é importante na modelagem de sistemas orientados a aspectos, como para sistemas desenvolvidos em outros paradigmas como programação orientada a objetos. Em relação as diferentes visões representadas por uma modelagem, os modelos propostos por este trabalho permitem representar a estrutura de um sistema com fatias de caso de uso e diagramas de classes. A dinâmica do sistema é representada através de colaborações que são associadas às fatias de caso de uso. Colaborações podem ser utilizadas para descrever a dinâmica de casos de uso e avisos. Outro ponto importante a ser destacado

é a representatividade do exemplo utilizado para realizar uma modelagem com a proposta. O sistema de gerenciamento de hotel é complexo e contém interesses entrecortantes de diversos tipos, possibilitando a representação de boa parte das características inerentes a programas orientados a aspectos.

Em relação as limitações do trabalho de (**??**), destaca-se a falta de ferramental para composição dos modelos, o que é importante para que seja possível visualizar o impacto dos interesses entrecortantes nos interesses núcleo do sistema. A geração automatizada de modelos também diminui o tempo necessário para realização de uma modelagem, diminuindo o tempo de entrega ao cliente. Nesta proposta, a composição entre interesses núcleos e interesses entrecortantes deve ser realizada manualmente, com o uso de diagramas de sequência, o que demanda um esforço adicional de modelagem. Outra limitação é a forma de extensão à UML, com a definição de um meta-modelo que não pode ser reusado em diferentes ferramentas CASE. Este meta-modelo tem construções específicas, como fatias de caso de uso, que não são suportadas por ferramentas CASE padrão segunda versão da UML.

Uma das unidades de negócio mais importantes da Motorola também está focada no desenvolvimento de soluções para modelagem de programas orientados a aspectos. O trabalho de (**??**) foi desenvolvido na unidade de negócios empresariais e de redes da Motorola. O projeto é denominado de **Motorola WEAVR** e é focado na especificação de sistemas de telecomunicações, que são sistemas reativos discretos e orientados a eventos. Um sistema reativo é um sistema que recebe uma entrada e deve emitir uma reação a este estímulo. Já um sistema discreto é um sistema cuja interação ocorre em eventos discretos no tempo. A motivação para este trabalho foi a percepção de que existem muitas mudanças nos requisitos de sistemas de telecomunicação ao longo do desenvolvimento. A modularização de interesses facilita a manutenabilidade e a inserção de novos requisitos ao longo do ciclo de vida desse tipo de sistema.

Motorola WEAVR permite a composição de aspectos que são modelados **com diagramas de máquina de estados focadas em transições**. Este tipo de máquina de estado é uma extensão à máquina de estados da UML focada em estados, com o objetivo de prover um maior nível de detalhe na dinâmica das transições. Segundo (BJöRKANDER, 2000), o diagrama de máquina de estados focado em estados da UML não dá a atenção que as transições merecem. Estas máquinas de estado ignoram as ações que ocorrem durante uma transição. A linguagem Specification and Description Language (SDL) (ITU-T, 2000) foca nas ações que acontecem durante uma transição. Esta linguagem é amplamente utilizada para especificar sistemas orientados a eventos na área de telecomunicações. Assim, a extensão proposta foca nas ações que ocorrem durante uma transição e estende o diagrama de máquina de estados da UML com novas construções que permitem representar as ações de uma transição. O diagrama de máquina de estados focado em transições permite diminuir o nível de abstração de uma modelagem, aproximando-o ao código e permitindo a geração de código em uma linguagem alvo. A modelagem de ações que ocorrem durante transições já está disponível na especificação do diagrama de máquina de estados da segunda versão da UML através deu um Perfil UML (UNIFIED...,).

Esta abordagem também utiliza diagramas de estrutura composta da segunda versão da UML para especificar as interfaces do sistema e os componentes em termo de sinais necessários e realizados. Este tipo de diagrama raramente é modificado durante o ciclo de vida de um sistema, por isso não são utilizados na modelagem de programas orientados a aspectos. O diagrama da figura 48 mostra um diagrama de estrutura composta para representar a estrutura de um servidor de recursos. Um servidor de recursos é composto por um despachante *Dispatcher* e por um ou mais tratadores de requisições *RequestHandler*. O despachante tem a responsabilidade de passar requisições externas para um dos tratadores de requisições. Um tratador de requisições é responsável por controlar o acesso a recursos, permitindo o acesso apenas se todos os recursos necessários estiverem disponíveis. Esse tipo de controle pode ser implementado com o protocolo *Two Phase Commit* .

Após modelar a estrutura deste sistema, deve-se modelar a dinâmica de cada um dos componentes do servidor. Para tal, utilizam-se os diagramas de máquina de estado focados em transições. Estes diagramas são utilizados para representar o comportamento de um componente em detalhes, com precisão e sem ambiguidades. Um diagrama deste tipo destaca o fluxo de controle e as ações executadas durante as transições entre os diferentes estados. O diagrama da figura 49 representa a modelagem do comportamento de um tratador de requisições.

Observando o primeiro diagrama de máquina de estados, verifica-se a presença de um conjunto de ações entre os estados *Init* e *Ready*. A primeira ação *request(rid)* indica o recebimento de um sinal para iniciar uma requisição. A segunda ação dentro de um retângulo tenta adquirir o canal e armazena o resultado na variável *status*. Após estas ações está presente um nodo decisão para verificar a variável

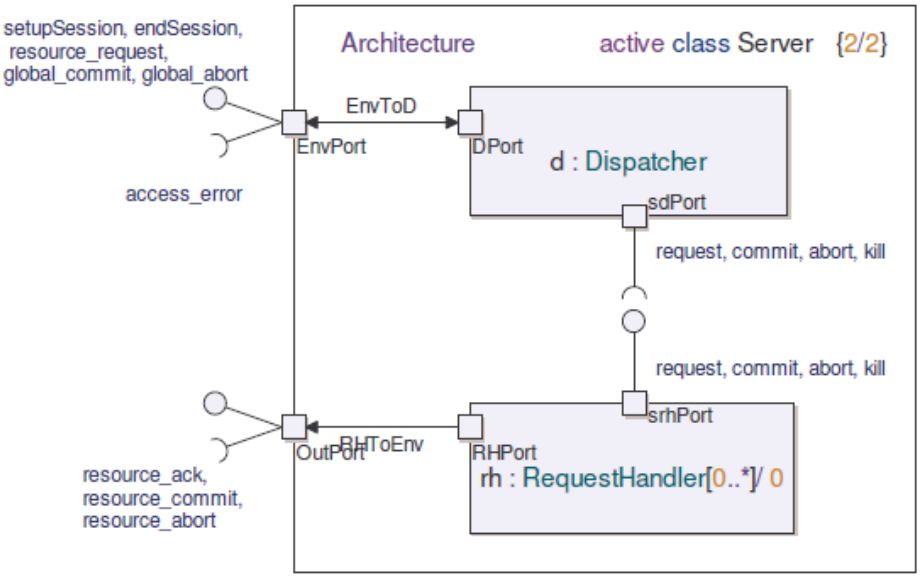

Figura 48 – Diagrama de Estrutura Composta representando um servidor de recursos.

status. Se o resultado for um *OK*, significa que o canal foi adquirido com sucesso e executa-se uma nova ação para adquirir o recurso desejado, armazenando o resultado na variável *status*. Se o valor da variável *status* no nodo decisão for *OK*, executa-se a ação *resource_commit(cid,rid)* que indica o envio de um sinal. Este sinal indica que o recurso foi adquirido. Finalmente, atinge-se o estado *Ready*. Se em algum momento a variável *status* retornar uma falha, o canal e o recurso serão liberados, abortando a acquisição do recurso. Todas estas ações ocorrem durante o disparo de uma transição.

No ambiente distribuído da Motorola existem vários sistemas que utilizam o protocolo 2PC no tratamento de requisições. Cada sistema implementa uma variação deste protocolo. A implementação do protocolo apenas com POO é possível, mas gera um código emarranhado e de difícil manutenção, pois o código do protocolo fica misturado com o código da aplicação. A POA permite implementar de maneira modular o protocolo 2PC. Esta foi uma das motivações para criação da ferramenta **Motorola WEAVR**.

Esta ferramenta define um perfil UML para modelagem de aspectos. A extensão proposta por (**??**) pode ser reutilizada em outras ferramentas de modelagem. O perfil adiciona o estereótipo *Aspect* que estende o elemento *Class* do meta-modelo da UML. Um aspecto pode conter conectores e pontos de corte. Um conector é o equivalente a um aviso na terminologia de aspectos. O estereótipo *Connector* foi criado para representar um conector. Para representar pontos de corte utiliza-se o estereótipo *Pointcut*. Estes estereótipos estendem o elemento do meta-modelo *Operation*. Conectores são associados a pontos de corte através do relacionamento de dependência *Binds*. A ordem de precedência de conectores é definida pelo relacionamento de dependência *Follows*. O estereótipo *Crosscuts* define o escopo de aplicação de um aspecto. Se nenhum escopo for definido, indica que o aspecto é aplicado a todo o sistema.

A figura 50 mostra a implementação de um aspecto de controle de tempo (*timeout*) para o protocolo 2PC. A motivação para implementação deste aspecto é que o tratador de requisições da figura 49 tem um problema: se uma instância entrar no estado *Ready*, mas não receber nenhum sinal do tipo *Commit* ou *Abort*, ela nunca terminará e não poderá receber novas requisições. Para solucionar este problema, adiciona-se um tempo máximo para que esta instância receba um sinal. Ao final deste tempo, a instância será destruída. Este comportamento é introduzido através do aspecto *2PCTimeoutAspect* que pode ser visualizado na figura 50.

Um dos elemenos definidos neste aspecto é o ponto de corte *requestCommitTransition*. A ferramenta WEAVR permite definir pontos de corte com o uso do diagrama de máquina de estados. Existem duas categorias de pontos de junção que que foram identificados nos diagramas de máquinas de estado:

- **Pontos de junção de ação**: Englobam as chamadas de operações, ações temporais e chamadas de construtores.
- **Pontos de junção de transição**: Compreendem o conjunto de caminhos de execução dentro de

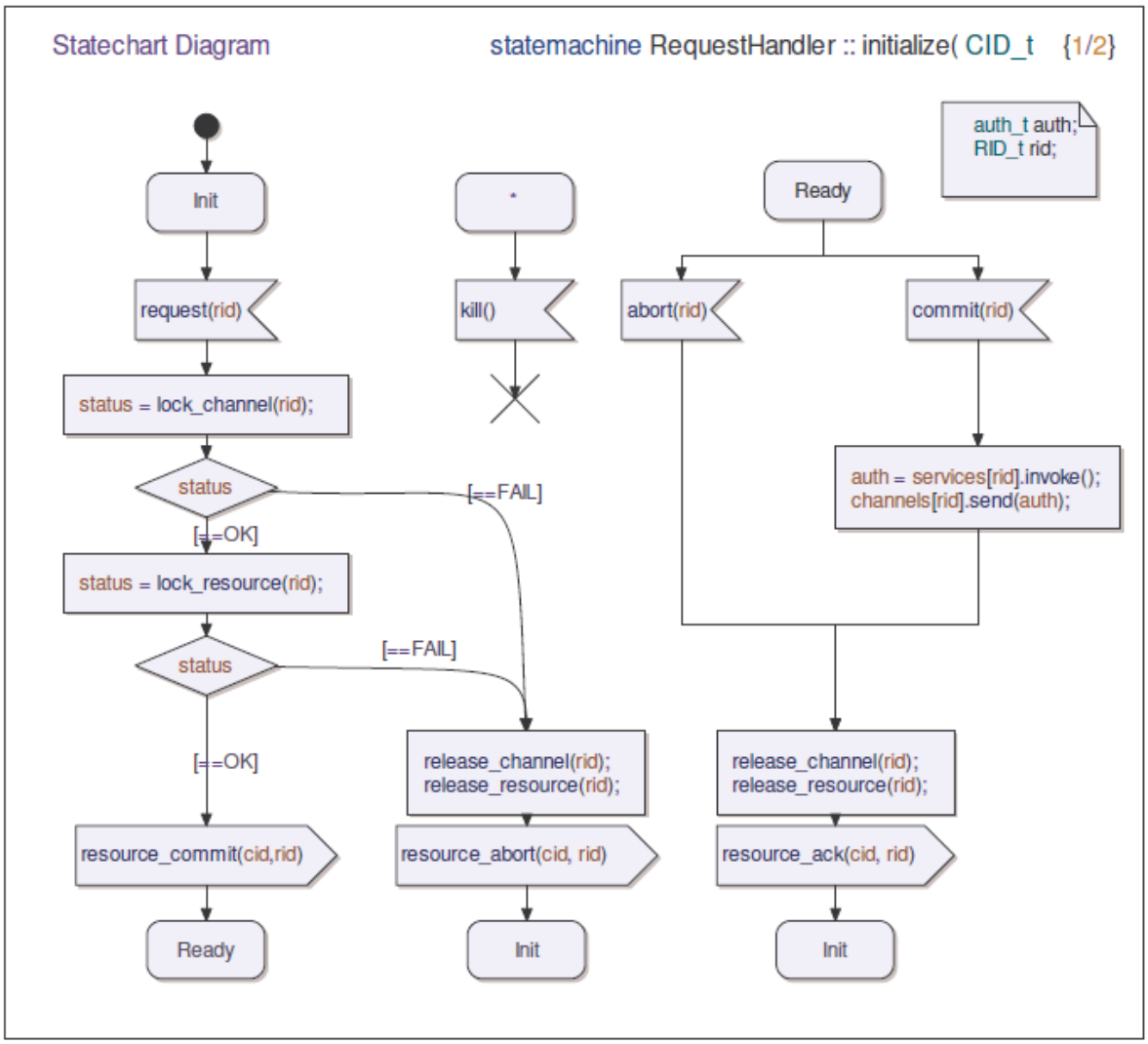

Figura 49 – Diagrama de máquina de estados focado em transições para um tratador de requisições.

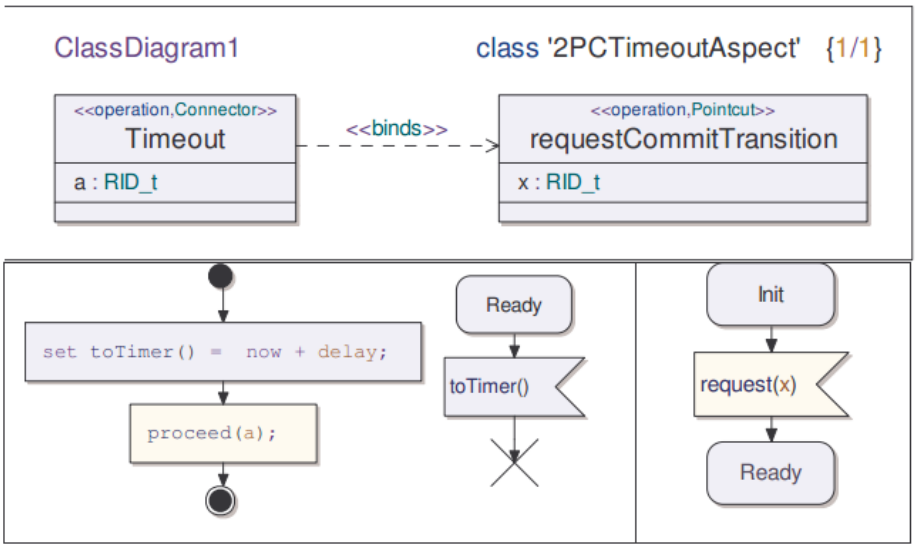

Figura 50 – Aspecto para controle de tempo ao protocolo 2PC.

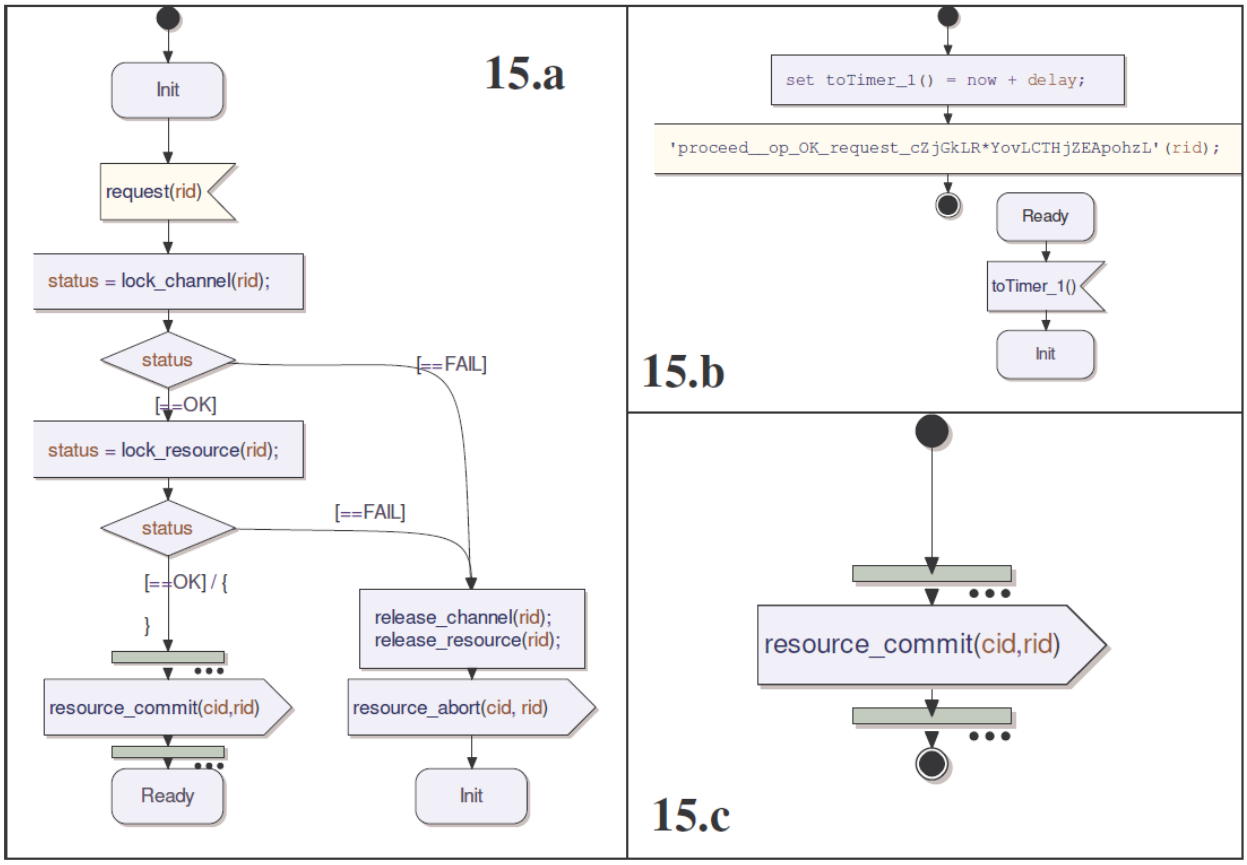

Figura 51 – Visualização da composição do aspecto para controle de tempo no tratador de requisições.

uma máquina de estados.

O ponto de corte *requestCommitTransition* captura pontos de junção de transição, pois captura execuções ao método *request(x)* enquanto uma instância estiver entre os estados *Init* e *Ready*, capturando um conjunto de caminhos de execução. Quando o ponto de corte for disparado, executa-se o comportamento descrito no conector *Timeout*. A ferramenta WEAVR representa a dinâmica de conectores através de diagrama de máquina de estados. Este conector é refinado em duas máquinas de estado. A primeira máquina de estado representa a reinicialização do contador de tempo antes das transições capturadas pelo ponto de corte. A segunda representa a destruição da instância ao atingir o tempo máximo sem receber nenhum sinal.

Para visualizar o impacto de aspectos em um modelo base, a ferramenta WEAVR disponibiliza um **visualizador do efeito de aspectos**. O visualizador é uma importante contribuição deste trabalho, pois permite visualizar em quais pontos do modelo base serão inseridos novos comportamentos. Por exemplo, ao aplicar o aspecto *2PCTimeoutAspect* (figura 50) no tratador de requisições *RequestHandler* (figura 49), o visualizador de aspectos gera a máquina de estados que pode ser visualizada na figura 51. Esta máquina de estado mostra os locais da máquina de estado que são impactados pelo aspecto de controle de tempo. Os pontos de junção de ação são marcados com a cor roxo, enquanto os pontos de junção de transição são marcados com uma marca horizontal verde ao longo das transições. O usuário pode clicar em um determinado ponto de junção para visualizar a máquina de estado do conector (aviso) que será executada naquele ponto. Observa-se na máquina de estados 15.a que a transição capturada é a transição que ocorre após o a variável *status* retornar OK até o estado *Ready* (dois marcadores horizontais verdes). O conector que será disparado pode ser visualizado na máquina de estados 15.b e uma representação da transição capturada pode ser visualizada na máquina de estados 15.c.

Motorola WEAVR também disponibiliza uma ferramenta para executar modelos de aspectos. Ao encontrar um conector, a ferramenta executa a máquina de estado referente ao mesmo e retorna o controle

a máquina de estado base. A possibilidade de executar um modelo de aspectos facilita a compreensão do fluxo de execução de um programa orientado a aspectos.

O trabalho de (**??**) estende a segunda versão da UML através de um Perfil UML para representar aspectos com diagramas de classes e diagramas de máquina de estados. Esta extensão pode ser importada em diferentes ferramentas CASE. Diagramas de classes permitem representar a parte estrutural de um sistema. Os diagramas de máquinas de estado são responsáveis por representar a dinâmica do sistema. Motorola WEAVR permite representar a maior parte das características de programas orientados a aspectos, como *wildcards*, pontos de corte, avisos, introduções e dependência entre aspectos. Uma limitação desta proposta é que apenas avisos do tipo durante (*around*) são suportados. No entanto, este tipo de aviso é o mais poderoso e permite executar comportamentos antes, durante ou depois de determinados pontos de um programa. A ferramenta também disponibiliza vasto ferramental para visualização, composição e execução de modelos de aspectos. Este ferramental é uma das principais contribuições do trabalho, pois automatiza grande parte do processo de modelagem e permite gerar código em um linguagem alvo. Uma das desvantagens desta proposta, é o baixo nível de abstração dos modelos. Isto acontece, pois um dos focos do trabalho é a execução do sistema em nível de modelo, deixando os modelos próximos do nível de código. Os diagramas de máquinas de estado desta proposta contém trechos de código e código referente a POA, como chamadas ao método *proceed()*. O baixo nível de abstração dificulta a compreensão dos modelos, pois o desenvolvedor precisa compreender chamadas próximas do nível de código. A utilização de diagramas como os diagramas de caso de uso e de sequência permitiriam representar um sistema em um maior nível de abstração, facilitando a compreensão e manutenção do mesmo.

O trabalho de (**??**) propõe uma extensão aos diagramas de sequência da UML para permitir a modelagem de sistemas que utilizam o paradigma de POA. Esta extensão é realizada no meta-modelo da UML, adicionando novas classes. Separam-se os diagramas de sequência em dois tipos:

- **bSD**: Diagrama de sequência básico que descreve um número finito de interações entre um conjunto de objetos. É equivalente ao diagrama de sequência padrão da UML.
- **cSD**: Diagrama de sequência composto que permite compor diagramas de sequência básicos através de operadores como nodos decisão e iterações. Este tipo de diagrama permire representar comportamentos infinitos e é semelhante ao diagrama de visão geral de interação.

O meta-modelo para diagramas de sequência pode ser visualizado na figura 52. Observa-se a presença de duas classes que derivam da super classe **SD**(*SequenceDiagram*): **bSD**(*BasicSequenceDiagram*) e **cSD**(*ComposedSequenceDiagram*). Um diagrama de sequência composto contém um conjunto de nodos e um conjunto de transições que estão associadas a diagramas de sequência básicos.

No lado esquerdo da figura 53 estão dispostos quatro bSD's: *Propose*, *Accept*, *Retry* e *Rejected*. Estes bSD's representam etapas na interação com um servidor para realizar a autenticação de um usuário(*log in*). O cSD central e o cSD a direita são equivalentes e compõem os quatro bSD's a esquerda. Ambos cSD's especificam a autenticação de usuário(*log in*). O cSD central utiliza nodos decisão e o cSD a direita utiliza fragmentos do tipo *alt* para representar diferentes caminhos de execução.

Para modelagem de aspectos deve-se definir um **aspecto comportamental** que é composto por dois bSD's: um para representar o ponto de corte e outro para representar o novo comportamento (aviso). Um ponto de corte é representado como uma sequência de mensagens entre um conjunto de objetos. A ferramenta não suporta a utilização de *wildcards* na definição de pontos de corte, uma funcionalidade importante da POA, que permite a captura de múltiplos pontos de junção em uma única declaração. Um aviso também é representado com bSD's e pode ser executado antes, durante ou depois dos pontos de junção capturados por um ponto de corte. A figura 54 mostra três aspectos comportamentais: registro de mensagens, segurança e atualização de interface gráfica. O aspecto de segurança(*Aspect Security*) define um ponto de corte que captura a troca das mensagens *log in* e *try again* entre os objetos *customer* e *server*. O aviso que implementa o comportamento do aspecto de segurança adiciona a mensagem *save bad attempt* entre as mensagens *log in* e *try again*. Uma limitação nesta abordagem é que o diagrama de sequência dos avisos é redundante, pois repete as mensagens capturadas no ponto de corte. No exemplo do aspecto de segurança, as mensagens *log in* e *try again* são repetidas no aviso. Assim, se o ponto de corte for modificado, o aviso também deve ser modificado, gerando um re-trabalho desnecessário. Na linguagem AspectJ, a modificação de um ponto de corte não gera modificações aos avisos associados aquele ponto de corte.

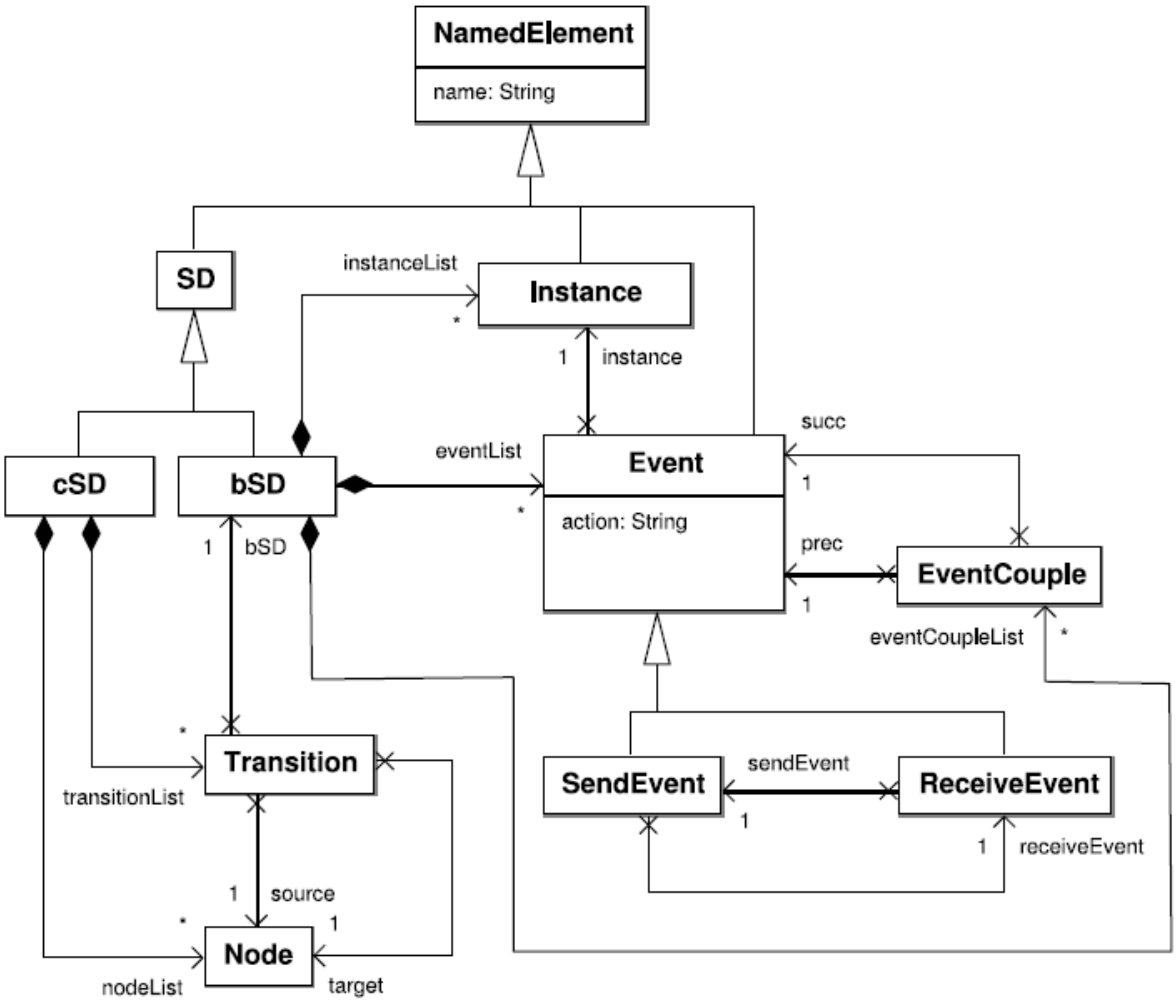

Figura 52 – Meta-modelo de extensão aos diagramas de sequência da UML.

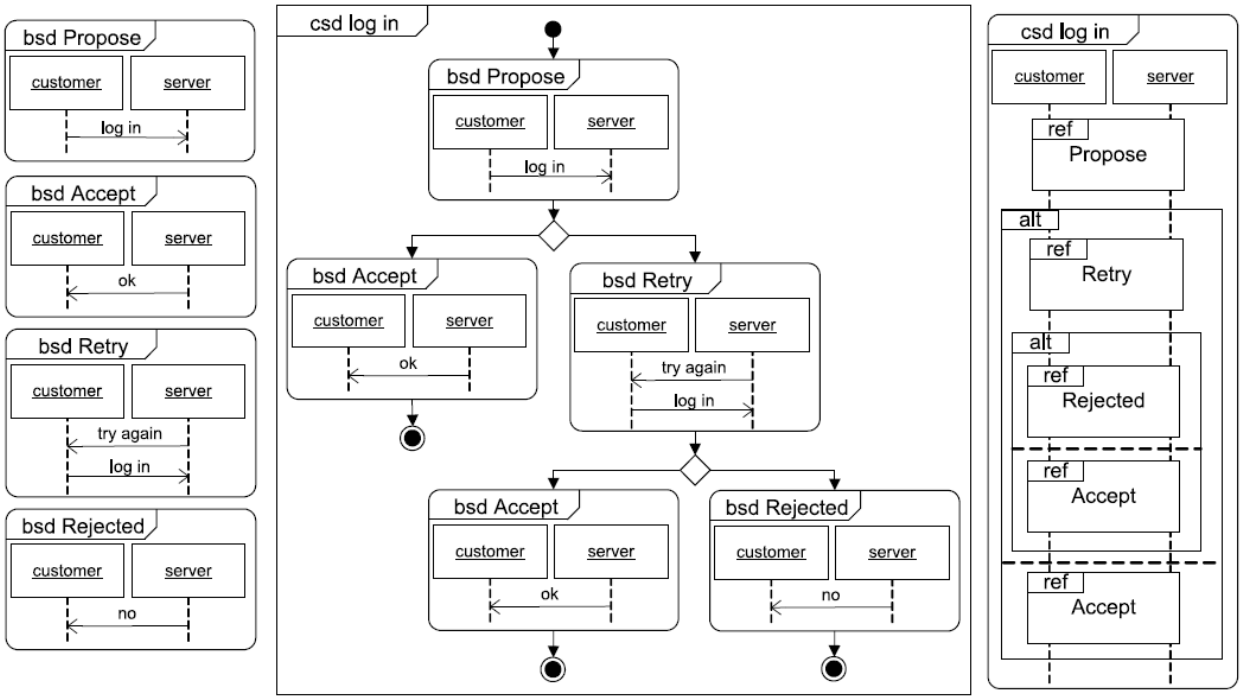

Figura 53 – Exemplos de diagramas de sequência estendidos: bSD's e cSD's.

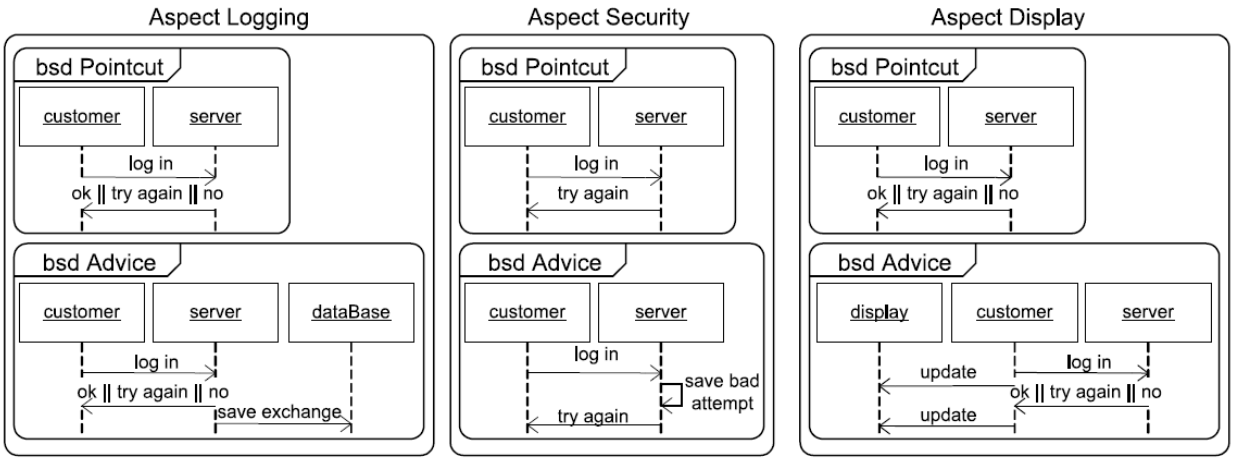

Figura 54 – Exemplos de aspectos comportamentais.

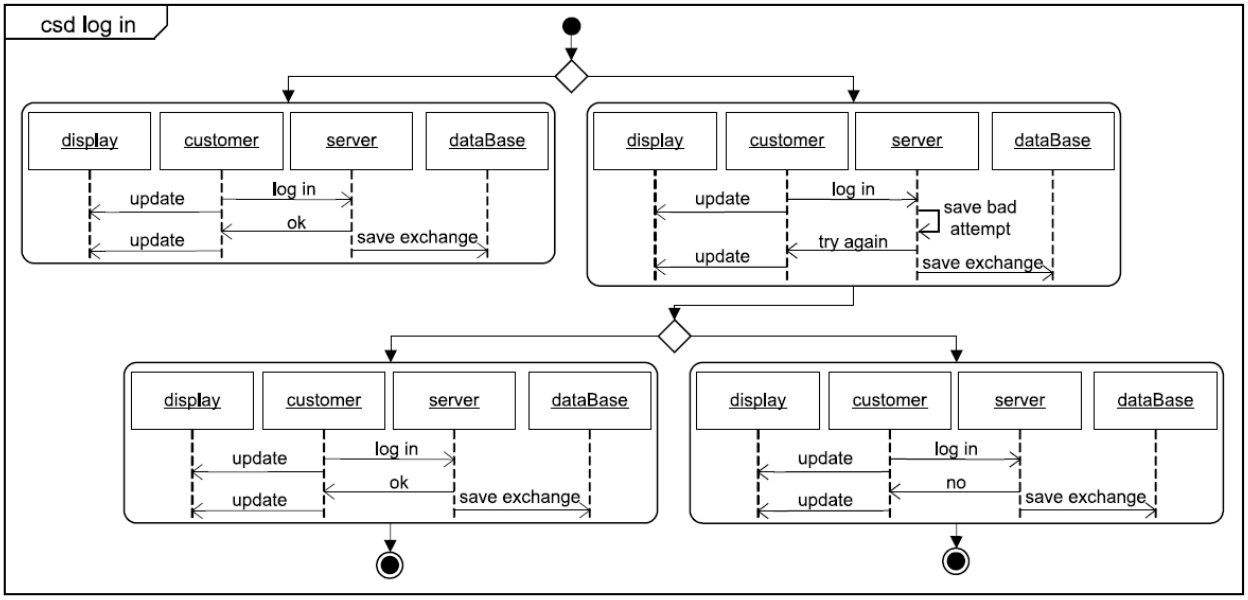

Figura 55 – Composição entre aspectos de registro de mensagens, segurança e atualização de interface gráfica com modelo base para autenticação de usuário.

Após definir os aspectos, deve-se realizar a composição dos aspectos em um modelo base. O foco deste proposta é a composição de múltiplos aspectos em um mesmo ponto de junção. Na composição de aspectos, o primeiro aspecto que for composto pode modificar a sequência de mensagens entre objetos no modelo base, fazendo com que o próximo aspecto não detecte o ponto de junção no qual o novo comportamento deveria ser inserido. Considerando a composição entre o diagrama de sequência (cSD) *log in* da figura 53 como modelo base e os aspectos para registro de mensagens, segurança e atualização de interface. Se o aspecto de segurança for o primeiro a ser composto, ele mudará a sequência de mensagens entre os objetos *customer* e *server*, adicionando a mensagem *save bad attempt* entre as mensagens *log in* e *try again*. Se não existir um tratamento no algoritmo de detecção de pontos de junção, os aspectos de registro de mensagens e atualização de interface não capturarão os pontos de junção definidos por seus pontos de corte e a composição não será realizada corretamente. Assim, o algoritmo de detecção de pontos de junção deve ser capaz de capturar pontos de junção, mesmo que existam mensagens introduzidas por outros aspectos durante a composição de aspectos. A proposta de (**??**) resolve este problema detectando mensagens inseridas por outros aspectos durante a composição de aspectos e permitindo a composição de múltiplos aspectos em um mesmo ponto de junção. A composição entre os aspectos e o modelo base pode ser visualizada na figura 55.

Uma das limitações da proposta de (**??**) é a não representação da estrutura do sistema. Este trabalho utiliza apenas diagramas de sequência para modelagem da dinâmica do sistema. Em relação a POA, não é possível representar introduções de membros em classes, uma funcionalidade importante da linguagem. Não é possível utilizar *wildcards* na definição de pontos de corte, o que torna difícil a captura de múltiplos pontos de junção em um mesmo ponto de corte. Além disso, a especificação de avisos contém redundâncias, pois replicam-se as mensagens referentes ao ponto de corte capturado. A principal contribuição do trabalho é a possibilidade de realizar a composição de múltiplos aspectos em um mesmo ponto de junção, mantendo as mensagens disparadas em cada aviso e identificando corretamente os pontos de junção, mesmo com mensagens introduzidas após a introdução de um aviso. Em relação a forma de extensão à UML, as modificações da proposta não podem ser usadas diretamente em qualquer ferramenta CASE. Os modelos gerados por esta proposta devem ser convertidos para o meta-modelo padrão da UML (através de transformações). A conversão de modelos não é tratada neste trabalho, dificultando ainda mais o reuso da proposta em diferentes ferramentas CASE.

A proposta de (BANIASSAD; CLARKE, 2004) permite realizar a análise e projeto de sistemas orientados a aspectos, desde o eliciamento de requisitos, passando pela modelagem e a automatizando parte do processo com a composição automatizada de modelos base com modelos de aspecto. A proposta

permite representar *wildcards* utilizando *templates* da UML e realiza a composição automatizada de modelos através de um plug-in do Eclipse. Uma das limitações da proposta é a forma de extensão à UML. Theme/UML estende o meta-modelo da UML no nível M2. O meta-modelo depende da versão 1.3 *beta* da UML e não pode ser utilizado em qualquer ferramenta CASE. Com o objetivo de padronizar a abordagem, foi desenvolvido um perfil UML baseado na versão 2.1 da UML (CARTON et al., 2009). No entanto, este trabalho ainda depende do meta-modelo original, pois para realizar a composição deve-se transformar os estereótipos e valores rotulados para o meta-modelo original do Theme/UML.

Criticar que com UML 1.3 não pode ter fragmentos combinados.

Dois possíveis problemas: definição de quando acontece um aspecto: antes, durante ou depois e também não conseguiram fazer a composição com diagrama de sequência no novo trabalho (tiveram que desenhar na mão por problemas no MagicDraw). Também não usaram um exemplo com wildcards.

3.1 ANÁLISE DOS TRABALHOS RELACIONADOS

Como visto na seção 3, existem várias propostas para modelagem de sistemas orientados a aspectos. Os trabalhos estudados utilizam a segunda versão da UML para modelagem. Este trabalho considera a utilização da segunda versão da UML como um pré-requisito para modelagem de programas orientados a aspectos. Algumas propostas estendem a primeira versão da UML e não foram estudadas neste trabalho ((ALDAWUD; ELRAD; BADER, 2003), (BANIASSAD; CLARKE, 2004),(STEIN; HANENBERG; UNLAND, 2002) e (LAURENCE et al., 2002)).

Identificam-se alguns pontos importantes na modelagem de programas orientados a aspectos. Estes pontos servem como base para a comparação entre as propostas estudadas e a proposta deste trabalho. É desejável que uma proposta satisfaça o maior número de pontos. A seguir serão descritos cada um destes pontos.

3.1.1 Pontos para comparação de abordagens

O primeiro ponto a ser observado é a **forma de extensão à UML**. Algumas modelagens estendem o meta-modelo da UML através de perfis, facilitando o reuso em diferentes ferramentas CASE. Outras modificam o meta-modelo da UML com novas meta-classes, relacionamentos e restrições, permitindo maior flexibilidade na definição dos conceitos, mas dificultando o reuso. É recomendado a utilização de perfis para representar sistemas orientados a aspectos, pois os mesmos podem ser compartilhados e estendidos em outras ferramentas CASE. Os trabalhos de (**??**), (**??**), (**??**) e (**??**) definem perfis UML, enquanto as propostas de (**??**), (**??**) e (**??**) modificam o meta-modelo da UML. O trabalho de (**??**) define um Perfil UML completo para especificação da estrutura de programas orientados a aspectos, superior ao Perfil UML proposto por (**??**).

Um outro aspecto importante são as **visões representadas na modelagem**. (SILVA, 2000) destaca a importância de representar as visões estruturais e dinâmicas de um sistema, com o objetivo de permitir a geração de código, automatização de processos de modelagem e facilitar a manutenção e compreensão do sistema. O trabalho de (**??**) destaca-se neste quesito, pois representa a estrutura e a dinâmica de um sistema com diferentes visões: estrutural, mensagens e de estados. As propostas de (**??**), (**??**) e (**??**) também representam as visões estruturais e dinâmicas. Os trabalhos de (**??**) e (**??**) representam apenas a visão dinâmica e o trabalho de (**??**) apenas a parte estrutural.

A proposta de modelagem deve permitir realizar a **composição dos modelos de aspectos (interesses entrecortantes) em modelos base (interesses núcleo)**. Esta composição pode ser realizada em nível de modelo ou de código. Recomenda-se que a composição seja realizada em nível de modelo, pois o analista pode compreender mais facilmente o sistema em um maior nível de abstração. Além disso, o próprio compilador AspectJ já realiza a composição em nível de código. Os trabalhos de (**??**), (**??**), (**??**), (**??**), (**??**) disponibilizam ferramentas para composição em nível de modelo. Destaca-se o trabalho de (**??**), que disponibiliza um visualizador de aspectos para avaliar o impacto dos aspectos em um sistema dinamicamente.

Um ponto fundamental é a **possibilidade de representar as características inerentes a POA** em uma modelagem. As abordagens devem permitir representar pontos de corte, avisos, introduções e *wildcards* na captura de pontos de junção. Dentre os trabalhos analisados, apenas os trabalhos de (**??**),

	(??)	(??)	(??)	(??)
Tipo de Extensão	Perfil UML	Meta-modelo	Meta-modelo	Perfil UML
Visões Representadas	Estrutural	Estrutural e Dinâmica	Estrutural e Dinâmica	Dinâmica
Permite Composição	Sim	Sim	Não	Sim
Nível de Composição	Código	Modelo	-	Modelo
Representação da POA	Parcial	Total	Total	Parcial
Difícil Compreensão	Não	Sim	Não	Não
Visualização Dinâmica	Não	Não	Não	Não

Tabela 2 – Comparação entre trabalhos para modelagem de programas orientados a aspectos (1).

	(??)	(??)	(??)
Tipo de Extensão	Meta-modelo	Perfil UML	Perfil UML
Visões Representadas	Dinâmica	Estrutural e Dinâmica	Estrutural e Dinâmica
Permite Composição	Sim	Sim	Sim
Nível de Composição	Modelo	Modelo	Modelo
Representação da POA	Parcial	Parcial	Total
Difícil Compreensão	Não	Não	Sim
Visualização Dinâmica	Não	Não	Sim

Tabela 3 – Comparação entre trabalhos para modelagem de programas orientados a aspectos (2).

(??) e (??) permitem representar todas as características da POA.

Também comparam-se os trabalhos em relação a **dificuldade para compreensão e manutenção** dos modelos. A modelagem de (??) é de difícil compreensão, pois os modelos são representados em baixo nível de abstração (próximos ao nível de código). O trabalho de (??) contém muitas dependências entre aspectos e conceitos novos que dificultam a compreensão de modelos. Além disso, o modelo composto contém muitos elementos sintáticos em um único diagrama. Os outros trabalhos podem ser compreendidos sem maiores problemas.

Finalmente, destaca-se a importância de de realizar a **alternância entre visões** diretamente no modelo final composto, ora visualizando o comportamento do sistema com aspectos, ora visualizando o sistema sem aspectos. Um visualizador dinâmico deste tipo é importante, pois permite visualizar e analisar o efeito de modelos de aspectos em modelos base, isto é, permite compreender em qual parte o comportamento do aspecto é inserido no modelo base. O trabalho de (??) é o único que disponibiliza um visualizador dinâmico e uma ferramenta para execução de modelos de aspectos.

As tabelas 2 e 3 destacam cada um dos pontos previamente apresentados e a comparação entre os trabalhos relacionados. Analisando a tabela, observa-se que existe uma divisão entre propostas que estendem o meta-modelo com perfis ou definem um novo meta-modelo. A maior parte das propostas disponibiliza ferramentas para composição e permitem representar a estrutura e a dinâmica de programas orientados aspectos. Algumas propostas falham na representação das características inerentes à POA. Apenas dois trabalhos geram modelos de difícil compreensão pelo analista e apenas um trabalho implementa um visualizador dinâmico de aspectos, que permite visualizar o impacto de aspectos em um sistema.

Este trabalho estende a UML através de um Perfil UML, representando as visões estruturais e dinâmicas de um sistema. Representando todas as características inerentes à POA com modelos de fácil compreensão e manutenção, disponibilizando ferramentas para composição em nível de modelo. Finalmente, um visualizador dinâmico de aspectos é implementado para permitir a visualização do efeito de modelos de aspectos em modelos base, juntamente com a inserção e remoção dinâmica de modelos de aspectos, considerada a principal contribuição deste trabalho.

REFERÊNCIAS

ALDAWUD, O.; ELRAD, T.; BADER, A. Uml profile for aspect-oriented software development. In: *The Third International Workshop on Aspect Oriented Modeling*. [S.l.: s.n.], 2003.

ASPECTJ. *The AspectJ Project*. june 2012. Disponível em: <http://eclipse.org/aspectj/>.

BANIASSAD, E.; CLARKE, S. Theme: An approach for aspect-oriented analysis and design. In: *Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004. (ICSE '04), p. 158–167. ISBN 0-7695-2163-0.

BJöRKANDER, M. Graphical programming using uml and sdl. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 33, p. 30–35, December 2000. ISSN 0018-9162.

CARTON, A. et al. Transactions on aspect-oriented software development vi. In: KATZ, S. et al. (Ed.). *AOSD*. Berlin, Heidelberg: Springer-Verlag, 2009. cap. Model-Driven Theme/UML, p. 238–266. ISBN 978-3-642-03763-4.

FARRINGTON, J. Seven plus or minus two. *Performance Improvement Quarterly*, John Wiley Sons, Inc., v. 23, n. 4, p. 113–116, 2011. ISSN 1937-8327.

FLEUREY, F. et al. A generic approach for automatic model composition. In: *In Proc. AOM at MoDELS*. [S.l.: s.n.], 2007. p. 2007.

ITU-T. *ITU Recommendation Z.100: The Specification and Description Language (SDL)*. [S.l.], 2000.

KLEPPE, A. G.; WARMER, J.; BAST, W. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 032119442X.

LAURENCE, R. P. et al. A uml notation for aspect-oriented software design. In: *in Workshop on Aspect-Oriented Modeling with UML (AOSD-2002*. [S.l.: s.n.], 2002.

MORIN, B.; KLEIN, J.; BARAIS, O. J.m.: A generic weaver for supporting product lines. In: *In: Early Aspects Workshop at ICSE*. [S.l.: s.n.], 2008.

(OMG), O. M. G. *Meta-Object Facility (MOF)*. [S.l.], aug 2011.

(OMG), O. M. G. *Meta-Object Facility XMI Specification*. [S.l.], aug 2011.

PARADIGM, V. *UML Profile Management*. dez. 2011. Disponível em: <http://www.visual-paradigm.com/product/vpuml/tutorials/umlprofile.jsp>.

PRESSMAN, R. S. *Software Engineering: A practitioner's approach*. [S.l.]: McGraw-Hill, 2001.

SILVA, E. M. M. E. L. D. *Metodologia da pesquisa e elaboração de dissertação*. [S.l.]: UFSC, 2001.

SILVA, R. P. e. *Suporte ao Desenvolvimento e Uso de Frameworks e Componentes*. Tese (Doutorado) — UFRGS/PPGC, march 2000.

SILVA, R. P. E. *UML 2 em Modelagem Orientada a Objetos*. [S.l.]: Visual Books, 2007.

STEIN, D.; HANENBERG, S.; UNLAND, R. A UML-based aspect-oriented design notation for AspectJ. In: *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2002. p. 106–112. ISBN 1-58113-469-X.

UNIFIED Modeling Language Superstructure 2.4.1. [S.l.], aug.

WIMMER, M. et al. A survey on uml-based aspect-oriented design modeling. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 43, p. 28:1–28:33, out. 2011. ISSN 0360-0300.