UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DISSERTAÇÃO DE MESTRADO

Aspecting: Abordagem para Migração de Sistemas OO para Sistemas OA

RICARDO ARGENTON RAMOS

São Carlos

Setembro/2004

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DISSERTAÇÃO DE MESTRADO

Abordagem *Aspecting*: Migração de Sistemas OO para Sistemas OA

RICARDO ARGENTON RAMOS

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos como parte dos requisitos para obtenção do título de mestre. Área de concentração: Engenharia de Software.

São Carlos

Setembro/2004

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

R175aa

Ramos, Ricardo Argenton.
Abordagem Aspecting: Migração de Sistemas OO para Sistemas OA / Ricardo Argenton Ramos. -- São Carlos : UFSCar, 2005.
126 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2004.

1. Programação em linguagens específicas. 2. Interesses. 3. Programação orientada à aspectos. I. Título.

CDD: 005.262 (20ª)

Pelo amor, compreensão e incentivo que recebi de minha mãe, dedico especialmente este trabalho a ela.
Mãe, obrigado por você existir. Amo você.

Agradecimentos

A Deus por permitir meu ingresso no mestrado e conseqüentemente sua conclusão.

A Prof. Rosângela Aparecida Delloso Penteado, pela constante orientação, paciência e companheirismo durante este período. Obrigado por tornar meu sonho de mestrado uma realidade.

A minha namorada Geisa, pela paciência e compreensão e por aceitar as várias negativas quanto à disponibilidade de atenção.

A meu irmão Lucas pelo apoio e companheirismo.

A meu pai pelo apoio e companheirismo.

Ao meu amigo/irmão Valter, pelo companheirismo e troca de idéias e comentários importantes sobre nossos projetos.

Aos amigos Daniel Delfine, Jéssica, Vinicius, Renato e Ricardo Afonso pelo companheirismo.

Aos meus amigos de turma Anderson Belgamo (Dinho), Pablo, Anderson Pazin, Luiz Fernando (Tuca), Eduardo (Escovar), Wesley (Bel), Moacir, Lúcio (Warta), Fernando (Bocudo), Taciana, Lucas, Matheus, Fabiana e Karina pelo companheirismo.

Aos amigos Frank, Daniel (Lucrédio), Alexandre Cassiano, Fernando Genta, Linder, Erico, Rinaldo, Juliano (Juquinha) e Jully.

Aos amigos de Ibitinga André (TanTan), Marcos (Quinhão), Alisson, Marcio, Robson, Luciana, Stella, Edilaine, Aline´s e Adriana.

A todos que de forma direta ou indireta contribuíram para a realização deste trabalho.

A D. Vera e D. Ofélia pela amizade e pelos cafezinhos.

As secretárias da Pós-Graduação Cristina e Mirian.

Ao CNPq pelo incentivo financeiro.

Resumo

Uma abordagem denominada *Aspecting* é proposta para a elicitação de aspectos em sistemas Orientados a Objetos e posterior implementação desses sistemas no paradigma Orientado a Aspectos. A Abordagem tem três etapas distintas: Entender a Funcionalidade do Sistema, Tratar Interesses e Comparar o Sistema Orientado a Aspectos com o Orientado a Objetos. Cada etapa é desenvolvida apresentando os detalhes que devem ser cuidados pelo engenheiro de software para que a migração dos sistemas seja realizada de forma segura e obtenha-se um produto de qualidade. Técnicas de separação de interesses e de programação Orientada a Aspectos são utilizadas em três estudos de caso para elaborar a Abordagem. Diretrizes auxiliam a modelagem dos interesses identificados, utilizando diagrama de classes com a notação UML. Outras Diretrizes auxiliam a implementação desses interesses em aspectos, utilizando a linguagem AspectJ. A utilização de testes de regressão no sistema após a reorganização do código do sistema agora implementado com aspectos, verificando que a funcionalidade original foi preservada, completa a abordagem. Um quarto estudo de caso, para demonstrar a aplicação da abordagem, é também realizado.

Abstract

An approach, named Aspecting, is proposed for eliciting aspects in Object Oriented systems and subsequently re-factoring these systems as aspect oriented systems. The approach has three phases: Understand the System Functionality, Treat the Interests and Compare the Object Oriented System with the Aspect Oriented System. Each phase is presented with the details to be observed by the software engineer for the migration of the systems to be made safely and to obtain a quality product. Techniques of separation of concerns and aspect oriented programming are used in three prospective study cases, in order to induce formulating the approach. Guidelines are established to model the concerns identified, using class diagrams in UML notation. Other guidelines govern the implementation of concerns as aspects, using the Aspect J language. The use of regression tests in the system after the system code reorganization now implemented with aspects, for checking if the original functionality has been preserved in the migration, completes the approach. A fourth study case, for illustrating the approach application, is also presented.

Sumário

Lista de Figuras

Lista de Quadros e Tabelas

Capítulo 1

Introdução

1.1. Considerações Iniciais

Sistemas legados freqüentemente possuem alto custo de manutenção, lógica desestruturada e com isso aumentam as dificuldades para a interação com novas tecnologias e novos ambientes. Porém, possuem embutidos em seu código as regras de negócio e o conhecimento de vários anos de seus desenvolvedores/mantenedores que não podem ser simplesmente descartados. Desenvolver um novo sistema utilizando novas tecnologias e conceitos, além do alto custo, de tempo e esforço tem-se o receio de que todas as regras de negócio contidas somente no código fonte sejam perdidas. Sendo assim, a engenharia reversa e a reengenharia são alternativas válidas criando um sistema modularizado, permitindo futuras manutenções e evoluções.

Um sistema de software é composto não só por requisitos funcionais, mas também não funcionais. Os interesses funcionais correspondem às condições ou à capacitação que devem ser contempladas pelo software. Geralmente, trata-se de necessidades do cliente e/ou do usuário para resolver um problema ou alcançar um objetivo. Os interesses não funcionais não expressam função (transformação) a ser implementada em um sistema, mas expressam

condições de comportamento e restrições que devem prevalecer, como por exemplo, tratamento de exceções, persistência de dados, segurança e de desempenho.

Técnicas de modularização, como a programação Orientada a Objetos tem êxito, porém sua abordagem de modularização em sistemas de software de acordo com um único interesse inerente é insuficiente, por não prover estruturas suficientes para desenvolver sistemas complexos (Kiczales e outros, 1997), (Tarr e outros, 1999), (Harrison e Ossher, 1993). Kulesza e Silva (2000) citam outro problema, mostrando que apesar das pesquisas em engenharia de software estarem bastante amadurecidas, a manutenção de software permanece como um problema central à área. O processo de manutenção envolve não apenas a correção de erros, mas, sobretudo a adequação do sistema para a integração de novas tecnologias e novos requisitos. O alto custo associado a essas adequações (Coleman, 1994) estimula o desenvolvimento de sistemas mais flexíveis a mudanças futuras. Outros fatores que motivam o desenvolvimento de sistemas mais flexíveis são:

- A dificuldade na especificação e entendimento dos requisitos de determinados domínios de aplicação, por exemplo, sistemas tolerantes a falhas e de tempo real;
- A tendência a constantes e rápidas mudanças nos requisitos de classes específicas de sistemas, como por exemplo, aplicações financeiras;
- A possibilidade de reutilização da estrutura do sistema para a construção de similares;
- A crescente demanda por aplicações que sejam executadas em ambientes multi-plataformas e multi-usuários, constantemente modificados e altamente dinâmicos como a Internet.

Como citado acima, as técnicas de orientação a objetos possuem algumas limitações, como o espalhamento e o entrelaçamento de código de diferentes interesses. Para solucionar essas limitações, surgiram algumas extensões como a programação orientada a aspectos (Kiczales e outros, 1997), a programação orientada a assunto (Ossher e outros, 1999) e a programação adaptativa (Lieberherr e outros, 1994). A Programação Orientada a Aspectos cuida da separação de interesses (requisitos não funcionais) que estão entrelaçados com os requisitos funcionais no código fonte. Existem diversas linguagens para apoiar esse paradigma, entre elas pode-se citar: AspectJ (Kiczales e outros, 2001a e 2001b), HyperJ (Tarr e Ossher, 2003), AspectC (Coady e outros, 2001), AspectS (Hirschfeld, 2003).

1.2. Objetivos

Dada a importância da separação de interesses, para maior reusabilidade e melhor manutenção, este trabalho propõe uma abordagem denominada *Aspecting*, que conduz, de modo ordenado, o engenheiro de software à explicitação de interesses não funcionais em sistemas orientados a objetos já existentes, implementados em linguagem Java e para posterior implementação desses sistemas no paradigma orientado a aspectos utilizando a linguagem AspectJ (Kiczales e outros, 2001a e 2001b).

Aspecting foi criada a partir da extração de interesses de estudos de caso, utilizando três diferentes sistemas. Uma lista denominada Lista de Indícios auxilia o engenheiro de software a identificar os interesses espalhados e entrelaçados no sistema. Diretrizes auxiliam a modelagem dos interesses identificados, utilizando diagrama de classes com a notação UML (UML, 2004). Diretrizes auxiliam a implementação desses interesses em aspectos, utilizando a linguagem AspectJ. A comparação entre o sistema Orientado a Objetos e o sistema agora implementado com aspectos, verifica que a funcionalidade do sistema original foi preservada, completando a abordagem.

1.3. Motivação

A principal motivação para a realização deste trabalho é a existência de sistemas orientados a objetos e implementados na linguagem Java, que possuem o entrelaçamento do código relativo aos interesses não funcionais com o código relativo ao interesse funcional da aplicação. A reusabilidade do código desses sistemas é baixa, sua manutenção é difícil e o custo de adição de novas funcionalidades e novos requisitos é alto. Migrando tais sistemas para o paradigma Orientado a Aspectos objetiva-se maior manutenibilidade, reusabilidade de código e facilidade na inserção de novas funcionalidades (manutenção adaptativa e evolutiva).

1.4. Relevância

O processo juntamente com as diretrizes aqui criadas auxilia engenheiros de software na migração de sistemas orientados a objetos para sistemas orientados a aspectos. A revitalização desses sistemas permite a redução de custos de manutenção. A abordagem atende tanto aos engenheiros de software que tem sistemas Orientados a Objetos e desejam migra-los para Orientados a Aspectos quanto no desenvolvimento de novos sistemas.

1.5. Organização da Dissertação

O Capítulo 2 aborda o levantamento bibliográfico considerando os assuntos relevantes para este trabalho. São feitas considerações sobre a manutenção de software, o desenvolvimento orientado a objetos utilizando a linguagem Java, a separação de interesses, a linguagem AspectJ; trabalhos relacionados com a extensão da UML para a programação orientada a aspectos e, também, são discutidos os trabalhos relacionados com testes funcionais.

O Capítulo 3 apresenta os estudos de caso realizados nos sistema de: caixa de banco (Portal Java, 2004); estação flutuante (Weiss e Lai, 1999); e reconhecimento de cadeias de caracteres em texto, GNU Grep, (GNU, 2003) que serviram de apoio para elaboração das diretrizes para modelagem e implementação dos interesses e a criação da Lista de Indícios.

O Capítulo 4 apresenta as etapas e passos da abordagem *Aspecting*.

No Capítulo 5, o resultado da aplicação da abordagem *Aspecting* em um sistema orientado a objetos implementado na linguagem Java como estudo de caso é mostrado. Além disso, algumas comparações baseadas em métricas de tamanho do software são realizadas para avaliar o sistema antes e após a aplicação da abordagem *Aspecting*.

No Capítulo 6 encontram-se as considerações finais comentando os resultados obtidos e sugerindo trabalhos futuros.

Capítulo 2

Assuntos Relacionados

2.1. Considerações Iniciais

A manutenção de software consome mais esforço do que qualquer atividade de engenharia de software, assim, engenheiros de software procuram processos para diminuir a quantidade de esforço dessa fase.

Atividades como a reconstrução e a reengenharia de sistemas podem ser uma solução para migrar sistemas legados com alto custo de manutenção para sistemas que tenham seu código mais organizado e melhor documentado.

A existência de novas linguagens que incorporam o princípio da separação do código que implementam os requisitos funcionais do código que implementam os requisitos não funcionais e a tendência de migrar os sistemas existentes para essa nova linguagem, despertam a necessidade de abordagens que facilitam essa migração, nesse contexto é que está inserida esta pesquisa.

Este capítulo está organizado da seguinte forma: a Seção 2.2 apresenta os trabalhos relacionados à manutenção de software; a Seção 2.3 aborda o desenvolvimento orientado a objetos utilizando a linguagem Java; a Seção 2.4 trata de separação de interesses; a Seção 2.5

foca a Programação Orientada a Aspectos; a Seção 2.6 apresenta a linguagem AspectJ; a Seção 2.7 apresenta os trabalhos relacionados com a extensão da UML para a programação orientada a aspectos; a Seção 2.8 aborda o teste funcional Orientado a Objetos e na Seção 2.9 estão as considerações finais.

2.2. Manutenção de Software

A manutenção de software é uma das fases mais problemáticas no ciclo de vida do software, caracterizando-se por um alto custo e baixa velocidade de implementação. Porém, é uma atividade inevitável, principalmente tratando-se de softwares, para os quais os usuários constantemente solicitam mudanças e agregações de novas funcionalidades (Bennet e Rajlich, 2000).

Segundo Pressman (2001), pode-se classificar a manutenção de software em quatro categorias principais:

_ **Manutenção Corretiva:** alteração de um software de forma a corrigir seus defeitos e assim garantir a continuidade de sua operação;

_ **Manutenção Adaptativa:** adição ou modificação de funcionalidades um software para adaptá-lo às mudanças ocorridas em seu ambiente;

_ **Manutenção Evolutiva:** inclusão de novas funcionalidades ao software além das originais a pedido do usuário;

_ **Manutenção Preventiva:** mudança do software para tornar sua manutenção mais fácil (aumento da manutenibilidade).

Como forma de manutenção preventiva tem-se engenharia reversa e reengenharia de sistemas.

Engenharia Reversa é o processo de análise de um sistema a partir de seu código fonte para identificar os seus componentes e inter-relacionamentos, a fim de criar uma representação de outra forma ou em um nível mais alto de abstração (Chikofsky, 1990). Realiza-se engenharia reversa, como o primeiro passo, quando se deseja modificar a linguagem de implementação e/ou o paradigma de desenvolvimento de um software por outro, com as mesmas características, mas não se têm todas as informações sobre ele.

Chikofsky (1990) define reengenharia como a análise e a alteração de um sistema a fim de reconstituí-lo e implementá-lo em uma nova forma. Assim, reengenharia é composta pela engenharia reversa seguida da engenharia avante.

Existem diversas abordagens para a realização de reengenharia:

a) Somente com a mudança da linguagem de programação – a funcionalidade e o paradigma em que o sistema foi desenvolvido são preservados, sendo somente a linguagem de implementação alterada (Pressmam, 2001).

b) De sistemas procedimentais para sistemas Orientados a Objetos (OO) – há necessidade de construir um modelo de análise OO, a partir de um código implementado em linguagem procedimental. Algumas experiências nesse sentido foram realizadas (Penteado, 1996), (Penteado, 1998), (Cagnin e outros, 1999), (Camargo, 2001), (Recchia, 2002), (Lemos, 2002).

c) Reengenharia de sistemas Orientados a Objetos - Demeyer e outros (1999, 2000a e 2000b), Salientam que os sistemas legados não estão limitados ao paradigma procedimental e a linguagens como COBOL. Assim criaram linguagens de padrões, as quais podem ser utilizadas não somente no processo de engenharia reversa, como também durante a reengenharia de sistemas como alternativa de prover a reestruturação de sistemas orientados a objetos. Os padrões apresentam o seguinte formato: Nome, Intuito, Contexto, Problema, Solução, Exemplos, Avaliação, Justificativa, Padrões Relacionados, Usos Conhecidos e Contexto Resultante e foram desenvolvidos com base na teoria de padrões de projeto de Gamma e outros (1995).

2.3. Desenvolvimento Orientado a Objetos e Linguagem Java

O desenvolvimento OO organiza os problemas e as suas soluções como um conjunto de objetos distintos (Pfleeger, 2004). Vários métodos apóiam o desenvolvimento de softwares nesse paradigma. Uma linguagem, chamada de UML (*Unified Modelling Language*) é a consolidação de outras técnicas de modelagem (UML, 2004), sendo apoiada por ferramentas CASE (Computer Aided Software Engineer), entre elas a *Rational Rose* (Rational Rose, 2004) e a Omondo (Omondo, 2004).

A UML tem quatro visões:

- Visão de Casos de uso: descreve a funcionalidade do sistema desempenhada pelos atores externos a ele.

  Jacobson e outros (1992) citam que a abordagem dirigida a Caso de Uso é importante para a elicitação e validação dos requisitos junto aos usuários, além de servir como um modelo central que é utilizado para todos os demais modelos das

próximas fases do processo de desenvolvimento: Análise, Projeto, Implementação, Testes e Manutenção de acordo com as necessidades.

Para auxiliar a identificação dos Casos de Uso, as seguintes questões devem ser respondidas (Jacobson e outros, 1992), (Schneider e Winters, 2001):

- Que funções o ator quer do sistema?
- O sistema armazena informação? Quais atores irão criar, ler, atualizar, ou excluir aquela informação?
- O sistema precisa notificar um ator sobre mudanças em seu estado interno?
- Há quaisquer eventos externos que o sistema deva conhecer? Qual ator informa ao sistema sobre esses eventos?

A Figura 2.1 mostra um Caso de Uso e a interação do Ator com o Caso de Uso (ou vice-versa) é representada por uma seta.

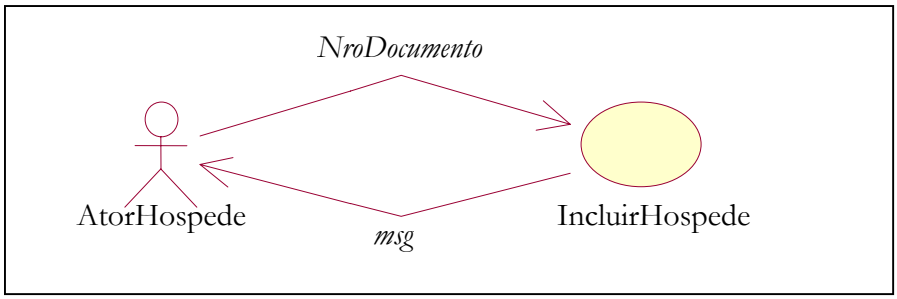


Figura 2.1. Interação entre Atores e Casos de Uso

O estereótipo <<extend>> é usado para estender o comportamento de um Caso de Uso existente. Dessa forma, adiciona-se comportamento ao Caso de Uso sem mudar o Caso de Uso original (Schneider & Winters, 20001). Jacobson (1992) menciona alguns exemplos de quando o <<extend>> é usado para modelar:

- Partes opcionais de Casos de Uso.
- Cursos complexos e alternativos os quais raramente ocorrem.
- Sub-cursos separados, os quais são executados somente em certos casos.
- A situação cujos Casos de Uso podem ser inseridos em um Caso de Uso especial.

O estereótipo <<include>> é usado para evitar a duplicação de passos através de múltiplos Casos de Uso. Esse tipo de estereótipo é empregado como uma estratégia de reuso para o contexto de Caso de Uso. Ao invés de colocar os passos em múltiplos documentos de especificação de Caso de Uso, criam-se Casos de Uso que possuem associação com o estereótipo <<include>> (Kulak e Guiney, 2000). Este tipo de

associação garante uma melhor consistência entre os Casos de Uso, pois evita a possibilidade da mesma funcionalidade ser especificada de forma inconsistente.

Para este trabalho somente serão utilizadas as especificações de curso normal e curso alternativo. Os diagramas de casos de uso serão elaborados para o entendimento da funcionalidade de um sistema Orientado a Objetos já existente, e para a elaboração de testes de funcionalidade.

- Visão Lógica: Descreve como a funcionalidade do sistema será implementada. É feita principalmente pelos analistas e desenvolvedores. Em contraste com a visão de casos de uso, a visão lógica observa e estuda o sistema internamente. Ela descreve e especifica a estrutura estática do sistema (classes, objetos, e relacionamentos) e as colaborações dinâmicas quando os objetos enviarem mensagens uns para os outros para realizarem as funções do sistema.
- Visão de Componentes: É uma descrição da implementação dos módulos e suas dependências. É principalmente executado por desenvolvedores, e consiste nos componentes dos diagramas.
- Visão de Organização: Finalmente, a visão de organização mostra a organização física do sistema, os computadores, os periféricos e como eles se conectam entre si. Essa visão apóia os desenvolvedores, integradores e testadores, e é representada pelo diagrama de execução.

2.3.1. A Linguagem Java

A linguagem Java implementa o paradigma Orientado a Objetos e dispõe de facilidades para o desenvolvimento de aplicações que são executadas por navegadores (*browsers*) na Internet e têm a característica de ser multi-plataforma.

O código fonte de um programa Java consiste de uma ou mais unidades de compilação. Cada unidade de compilação pode conter, além dos espaços em branco e comentários:

a) **Pacotes** são conjuntos de classes relacionadas, reusados pela inclusão de uma linha no topo de cada arquivo indicando o pacote que contém as classes. A forma geral da declaração de pacote é: `package pac1[.pac2[.pac3]];`

b) Classe é um tipo definido pelo usuário que envolve atributos e operações. A forma geral de definição de uma classe é:

```
Class <nome classe> extends <nome superclasse> implements <nome
interface> {
      tipo <variável instancia1>; ...
      tipo <variável instanciaN>;
      tipo <nome operacao1> (<lista de parâmetros>)
            {  <corpo do operacao>;}  ...
      tipo <nome operacaoN> (<lista de parâmetros>)
            {     <corpo do operacao>;  }
   }
```

c) **classe abstrata** declara uma estrutura, sem fornecer completamente a implementação de cada operação.

```
abstract class <nome classe1> {
    abstract void <nome metodo1> ( );
}
class <nome classe2> extends <nome classe1> {
    void <nome metodo1> ( ){
    <codigo> }
}
```

d) Operação **construtora** inicializa um objeto imediatamente após sua criação. Tem o mesmo nome da classe na qual reside. Depois de definido, o construtor é automaticamente chamado, imediatamente após a criação do objeto e antes que o operador **`new`** termine sua execução.

```
class <nome classe> {
   <mesmo nome classe>(){   }
}
```

e) Operação **destrutora** é diferente de seus equivalentes em outras linguagens, devido à coleta automática de lixo (*garbage collection*). Em Java, operações destrutoras `finalize()` são chamadas pelo sistema quando a memória de um objeto está para ser liberada pelo coletor automático de lixo. f) Operações **Estáticas** usadas fora do contexto de qualquer instância, são declaradas como `static`. Estas operações referem-se a outras operações `static` e não podem se referir à própria classe ou a classe superior. As variáveis declaradas como `static` levam à declaração de uma variável global.

```
class <nome classe>
{     static tipo <nome variável>;
     static modificadores <nome operacao> (){}
}
```

2.4. Separação de Interesses

O desenvolvimento de um sistema é diretamente dependente da elicitação de requisitos. Essa atividade é essencial para que a funcionalidade do sistema seja completamente conhecida e corresponda às expectativas de seus usuários. Outro ponto a ser observado é com relação aos requisitos não funcionais. Eles, tanto quanto os requisitos funcionais, são vitais para o completo desenvolvimento de sistemas.

Os requisitos funcionais podem ser definidos como as funções ou atividades que o sistema faz (quando pronto) ou fará (quando em desenvolvimento). Devem ser definidos claramente e relatados explicitamente (Cysneiros e Leite, 1997). Podem ser elaborados a partir do relato das necessidades do cliente e/ou usuário, em que uma equipe de projeto pode especificar efetivamente um sistema, suas funções, desempenho, interfaces, restrições, etc., conforme as fases e subfases de uma metodologia de desenvolvimento de sistemas ou software.

Requisitos não funcionais (Chung e outros, 2000), ao contrário dos funcionais, não expressam função (transformação) a ser implementada em um sistema, expressam condições de comportamento e restrições que devem prevalecer. Como exemplo, pode-se citar tratamento de exceções, segurança, persistência de dados, desempenho, entre outros. Chung e outros (2000) citam a dificuldade de tratar requisitos não funcionais em alguns sistemas, pois um requisito pode ser não funcional em um sistema e ser funcional em outro sistema. Para o contexto deste trabalho o termo “interesses” refere-se aos requisitos não funcionais em nível de projeto.

Czarnecki e Eisenecker (2000) comentam que a necessidade de manipular um requisito importante de cada vez, durante o desenvolvimento de um sistema, é chamado de princípio da separação de interesses. Linguagens de programação geralmente fornecem construtores para organizar um sistema em unidades modulares que representam os interesses funcionais da aplicação. Essas unidades são expressas como objetos, módulos e procedimentos. Mas, também, há interesses em um sistema que abrangem mais de um componente funcional, tais como: sincronização, interação de componentes, persistência e controle de segurança. Esses interesses são geralmente implementados por fragmentos de código espalhados pelos componentes funcionais. Eles são chamados de aspectos (em nível de código) e de interesses (em nível de projeto) e alguns são dependentes de um domínio específico, enquanto outros são mais gerais.

Outra técnica que procura auxiliar a separação de interesses é a utilização de padrões de projeto, como os propostos por Gamma e outros (1995). Porém, segundo Noda e Kishi (2001), as técnicas atuais de programação não são adequadas para a utilização dos padrões de projeto por tornarem a aplicação dependente deles, diminuindo as chances de reuso da parte funcional da aplicação. O reuso do código que implementa os padrões de projeto é dificultado, pois com as técnicas de programação OO existe o entrelaçamento do código relativo ao padrão como código funcional da aplicação.

Com a utilização de técnicas da programação orientada a aspectos, com a intenção de melhorar a separação de interesses de sistema implementados com padrões de projeto, Camargo e outros (2003) implementam com aspectos o padrão de projeto Camada de Persistência (Yoder e outros, 1998) e comentam as vantagens de se ter os interesses (padrões e sub-padrões que compõe a camada de persistência) separados do código funcional. Vantagens como reuso em nível de código do padrão devido à inversão das dependências. O código do padrão depende dos participantes e não ao contrário, como é na implementação orientada a objetos. Isso possui um impacto direto na localidade de código, pois todas as dependências entre os padrões e a aplicação são localizadas no código do padrão.

A implementação em aspectos do padrão de projeto Camada de Persistência foi reutilizada em um outro sistema OO que continha o interesse de persistência entrelaçado e espalhado pelo código fonte (Ramos e outros, 2004a). Assim, os autores inferem a maior facilidade do reuso quando o padrão de projeto está implementado em aspectos.

Portanto, para se obter vantagens como a independência de interesses, facilidades de reuso, facilidade de manutenção por se obter um código melhor organizado, é imprescindível a utilização de técnicas de separação de interesses, como a programação orientada a aspectos, que é o assunto da próxima Seção.

2.5. Programação Orientada a Aspectos

Segundo Elrad e outros (2001a) (2001b), a programação orientada a aspectos consiste na separação dos interesses de um sistema em unidades modulares e posterior composição (*weaving*) desses módulos em um sistema completo. Esses interesses podem variar de noções de alto nível, como segurança e qualidade de serviço, a noções de baixo nível, como sincronização e manipulação de *buffers* de memória.

O paradigma da programação orientada a aspectos (POA) surgiu com Gregor Kiczales (1997) com o intuito de fornecer uma forma de desenvolvimento que considera

separadamente os requisitos funcionais e os não funcionais de um sistema, para sua posterior composição (*weaving*) em um único sistema. O objetivo foi introduzir uma nova dimensão na programação orientada a objetos para que existam tanto objetos quanto aspectos, sendo que o primeiro encapsula os requisitos funcionais e o segundo os não funcionais (Kiczales e outros, 1997)

As técnicas de modularização utilizadas na programação Orientada a Objetos, provaram terem êxito, porém sua forma de modularização em sistemas de software, de acordo com um único interesse inerente, é insuficiente e pode não prover estruturas suficientes o bastante para que se desenvolvam sistemas complexos (Tarr e outros, 1999), (Harrison e Ossher, 1993). No mesmo contexto Kiczales e outros (1997), citam alguns problemas de implementação que as técnicas de programação orientadas a objetos ou procedimentais não conseguem resolver, como: o emaranhamento de código funcional com o não funcional e o espalhamento de código não funcional pelos módulos funcionais. Os autores afirmam que as unidades da programação orientada a objetos concentram-se nos interesses funcionais da aplicação, que são expressas como objetos, módulos e procedimentos. Mas, também há interesses de um sistema envolvido em mais de um componente funcional, tais como: sincronização, interação de componentes, persistência e controle de segurança. Esses interesses são geralmente expressos por fragmentos de código espalhados pelos componentes funcionais. Eles são chamados de aspectos e alguns são dependentes de um domínio específico enquanto outros são mais gerais. Kiczales e outros (1997) definem um aspecto como sendo uma unidade modular que entrecorta outras unidades modulares.

Elrad e outros (2001a), discutem as limitações da técnica de Programação Orientada a Objetos (POO) quanto à captura de interesse e citam em seu trabalho algumas técnicas de programação pós-objeto (POP, *Post-Object Programming*) para melhorar a POO. Alguns exemplos da tecnologia POP utilizam linguagens específica de domínio tais como: programação generativa (*generative programming*) (Czarnecki e Eisenecker, 2000), metaprogramação (*metaprogramming*) (Eisenecker e Czarnecki, 1999) desenvolvimento orientado a características (*feature-oriented development*) (Kang, 1998).

Kiczales e outros (1997) definem os interesses ortogonais (*crosscuting concerns*) como interesses que são encontrados espalhados e emaranhados nos módulos (classes em POO) de um sistema, como por exemplo, persistência, segurança, concorrência, manipulação de exceções, entre outros.

POA trata os interesses que entrecortam as classes de modo análogo ao que a programação orientada a objetos faz para o encapsulamento e a herança. Ou seja, provê mecanismos de linguagem que explicitamente capturam a estrutura de entrecorte, alcançando, assim, os benefícios de melhor modularidade, tais como: código mais simples, mais fácil de se manter e se alterar e conseqüentemente, aumentando a reusabilidade do código (Kiczales e outros, 1997).

A Figura 2.2 mostra como um determinado interesse pode entrecortar determinadas unidades modulares de um sistema. O diagrama de classes UML refere-se a um editor de figuras que possui duas classes concretas, `Point` e `Line`, com boa modularidade, pois suas funcionalidades não estão espalhadas em outras classes. Porém, necessita-se adicionar um novo interesse que notifique algum mecanismo de gerenciamento de exibição toda vez que um elemento do tipo `Figure` for movimentado. Isso é, toda vez que for invocado algum método de atribuição de valores a um objeto da classe `Point` ou da classe `Line`, tipo (`setX(int)`, `setY(int)` e `setP1(Point)`, `setP2(Point)`) deve-se, chamar um método `Display.updating`. Isso torna o código funcional da classe entrelaçado e espalhado com o código desse novo interesse. Utilizando a programação orientada a aspectos pode-se modularizar esse novo interesse de forma que o código que o implementa não afete o código que trata de outros interesses. A solução para esse tipo de problema com a programação orientada a aspectos será exibida na Figura 2.5, na Seção 2.6.

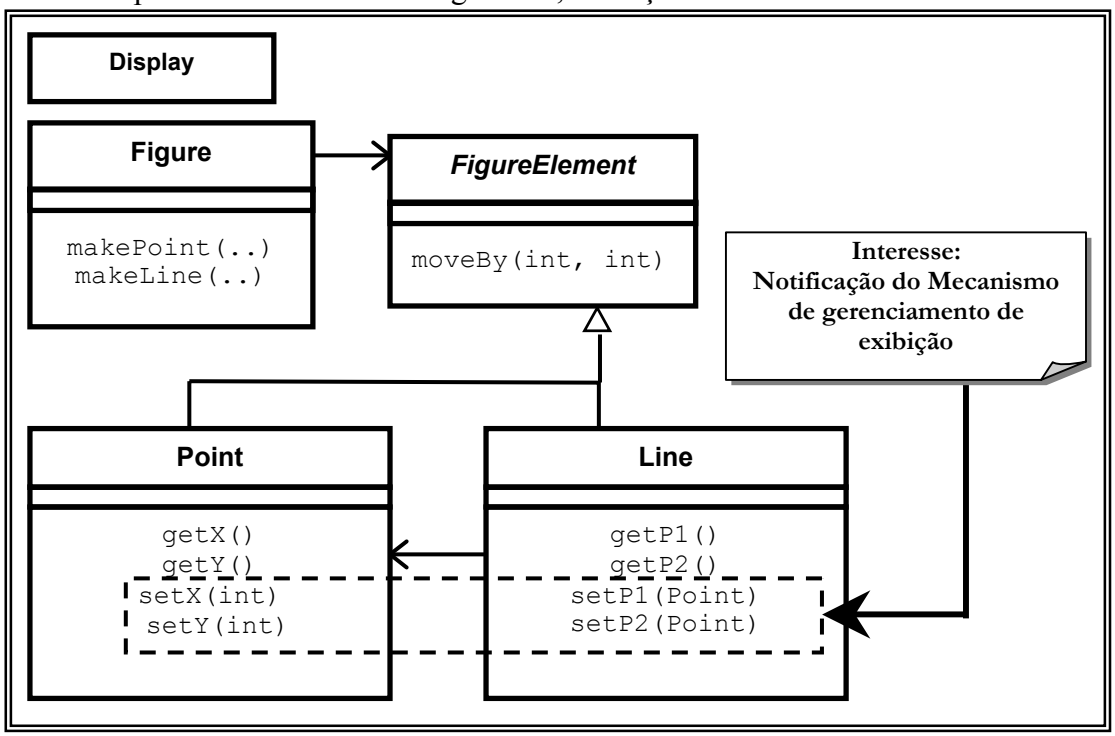


Figura 2.2 - Diagrama de Classes UML de Um Editor de Figuras.

2.6. AspectJ

AspectJ foi a primeira linguagem orientada a aspectos de propósito geral criada, surgiu em 1997 no *Xerox Palo Alto Research Center* – PARC. Foi desenvolvida como uma extensão compatível com a linguagem Java, de forma a facilitar sua adoção por programadores já habituados com essa linguagem. Em AspectJ os componentes são representados por classes Java e uma nova construção, denominada *aspect*, é responsável por implementar os interesses que entrecortam a estrutura das classes (Kiczales e outros, 2001a). Em 2002 essa linguagem foi incorporada ao projeto Eclipse (IBM e outros, 2001).

A composição dos aspectos com as classes é executada em tempo de compilação, sobre *bytecode* ou código fonte Java. Uma vez executada a compilação, os entrecortes estarão permanentemente incorporados às classes, não sendo possível alterar a composição durante a execução do programa. Hilsdale e Hugunin (2004) afirmam que o processo de composição segue duas etapas: i) o código Java e o código AspectJ são compilados em *bytecode* Java, instrumentado com informações sobre as construções do AspectJ (como sugestões e pontos de corte); ii) o compilador utiliza essas informações para gerar as classes compostas (*bytecode* compatível com a especificação Java), Figura 2.3.

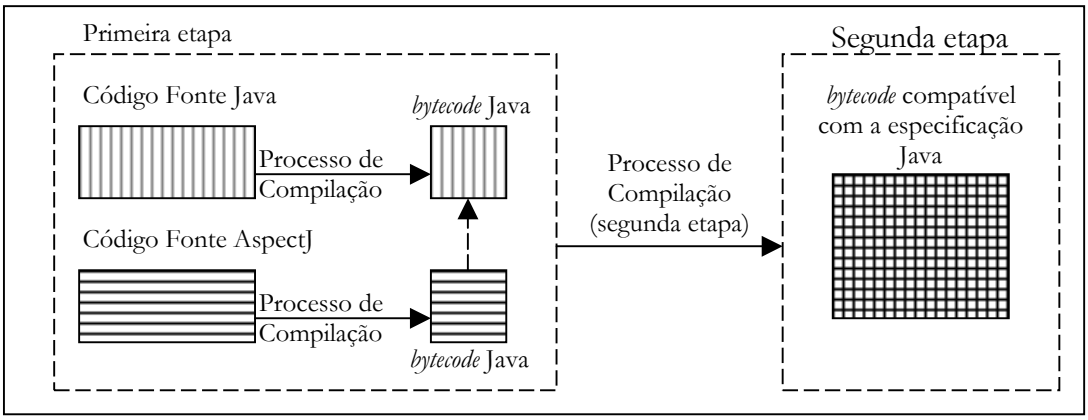


Figura 2.3 - Processo de Composição (*Weaving*) do AspectJ Segundo Hilsdale e Hugunin (2004).

A modularização de interesses de entrecorte é realizada usando pontos de junção (*join points*) e sugestões (*advice*). Pontos de junção são métodos bem definidos na execução do fluxo do programa e sugestão (*advices*) define o código que é executado quando os pontos de junção são alcançados. A Tabela 2.1 mostra todos os possíveis pontos de junção da linguagem AspectJ.

Tabela 2.1 - Tipo de pontos de junção do AspectJ (AspectJ Team, 2003)

Categoria	Descrição
Chamada de método **Chamada de construtor**	Um método ou um construtor de uma classe é chamado. Pontos de junção deste tipo encontram-se no objeto chamador ou possuem valor nulo (se a chamada é feita a partir de um método estático)
Execução de método **Execução de construtor**	Um método ou construtor é chamado. Pontos de junção deste tipo ocorrem no objeto chamado, dentro do contexto do método
Leitura de atributo	Um atributo de um objeto, classe ou interface é lido.
Escrita de atributo	Um atributo de um objeto ou classe é escrito.
Execução de tratador de exceção	Um tratador de exceção é invocado
Iniciação de classe	Iniciadores estáticos de uma classe (se existirem) são executados.
Iniciação de objeto	Iniciadores dinâmicos de uma classe (se existirem) são executados durante a criação do objeto, abrangendo desde o retorno da chamada ao construtor da classe pai até o retorno do primeiro construtor chamado
Pré-iniciação de objeto	Pré-iniciadores de uma classe são executados, abrangendo desde a chamada ao primeiro construtor até a chamada ao construtor da classe pai
Execução de sugestão	Qualquer parte de uma sugestão é executada

A seguir é apresentado como são tratados alguns conceitos utilizados em orientação a aspectos na linguagem AspectJ:

a) Pontos de Corte (*Pointcut*): identifica coleções de pontos de junção no fluxo do programa. O formato geral de um ponto de corte anônimo é:

```
especificação da sugestão : definição do ponto de corte
```

Caracteres coringa são utilizados para construção de pontos de corte, de forma a capturar pontos de junção com características comuns, que por sua vez, podem estar espalhados pelos diversos módulos do software. A linguagem utiliza três caracteres coringa, descritos no Quadro 2.1.

Quadro 2.1 - Caracteres Coringa do AspectJ.

Caractere	Descrição
*	Qualquer número de caracteres, excluindo o caractere ponto (.).
..	Qualquer número de caracteres, incluindo qualquer número de caracteres ponto (.).
+	Qualquer subclasse ou subinterface de um tipo específico.

AspectJ permite, também, o uso de operadores unários e binários, além de parênteses, que podem ser utilizados na construção de pontos de corte complexos a partir de expressões simples. Assim como na linguagem Java, o operador binário **!** é usado para negação e os operadores unários **||** e **&&** são usados como OU e E lógicos, respectivamente. O Quadro 2.2 exemplifica os operadores de AspectJ e exemplos de seus usos, conjuntamente com os caracteres coringas (*wildcards*).

Quadro 2.2 - Exemplo de Uso de Operadores em AspectJ.

Expressões Utilizando os Operadores	Descrição
`!Vector`	Todos os tipos exceto `Vector`
`Vector || Hashquadro`	Tipos `Vector` ou `Hashquadro`
`javax.. * Model || javax.swing.text.Document`	**Todos os tipos no pacote** `javax` **ou seus** sub-pacotes que possuam o nome terminando com `Model` ou `javax.swing.text.Document`
`java.util.RandomAccess+ && java.util.List+`	Todos os tipos que implementam as interfaces `java.util.RandomAccess` e `java.util.List`

Pontos de corte podem ser definidos por meio das assinaturas explícitas de métodos ou utilizando operadores e caracteres coringas (Kiczales e outros., 2001b). Assim, como nos métodos, expressões regulares também podem ser utilizadas em construtores e atributos para filtrar pontos de junção. No Quadro 2.3 encontram-se exemplos desses tipos de expressões.

Quadro 2.3 - Exemplos de Ponto de Corte.

Pontos de Corte Utilizando os Operadores	Descrição
`get(int Point.x)`	Leitura do atributo x do tipo inteiro da classe `Point`.
`set(!private *Point.*)`	Escrita de qualquer atributo não privado da classe `Point`.
`call(public *Figure.* (..))`	Chamada a métodos públicos da classe `Figure`.
`call(public ** (..))`	Chamada a qualquer método público de qualquer classe.
`call( *.new(int, int))`	Chamada a qualquer construtor de qualquer classe com dois argumentos do tipo inteiro.
`execution( ** (..))`	Execução de qualquer método de qualquer classe, independente do retorno e de parâmetros.
`execution(!static ** (..))`	Execução de qualquer método não-estático de qualquer classe.

O código `pointcut moves():` é um ponto de corte que identifica todas as chamadas para o método `setX` na classe `Point`, Figura 2.4. A clausula `||` (ou) em (a) e em (b) mostra que é possível fazer um ponto de corte utilizando coringas (*wildcards*), representado pelo * após a palavra set. O sufixo *(..), junto ao nome do método, indica que todos os métodos da classe `Line` que comecem com `set`, com qualquer parâmetro, serão pontos de junção.

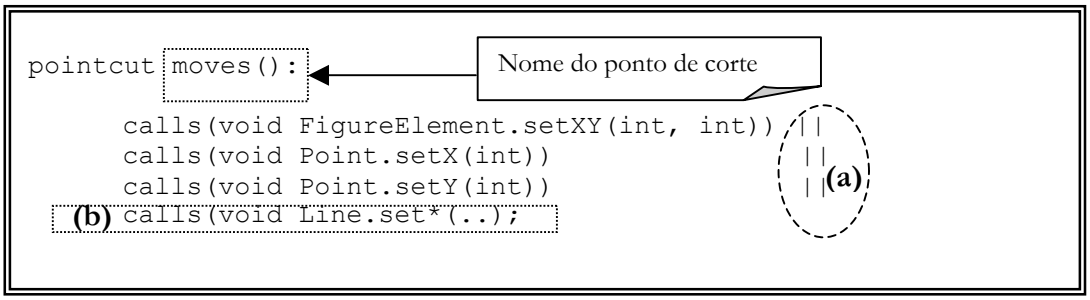


Figura 2.4 - Exemplo de Um Ponto de Corte Chamado `moves()`.

b) Sugestões (*Advices*): são construções semelhantes a métodos, que definem comportamentos adicionais aos pontos de junção. Assim, sugestões expressam as ações de entrecorte que serão executadas nos pontos de junção capturados por um ponto de corte (Kiczales e outros, 2001a), (Laddad, 2003). AspectJ tem três tipos diferentes de sugestões:

i) **Pré-sugestão (*before*):** é executado quando um ponto de junção é alcançado e antes da computação ser realizada.

ii) **Pós-susgestão (*after*):** é executado após a execução do corpo do método que foi alcançado pelo ponto de junção.

iii) **Sugestão substitutiva (*around*):** é executado quando o ponto de junção é alcançado e tem o controle explícito da computação, podendo alternar a execução com o método alcançado pelo ponto de junção.

```
class Line
{
   private  Point p1, p2;

  Point getP1() {return p1;}
  Point getP2() {return p2;}

  void setP1(Point p1) {
      this.p1 = p1;
      Display.updating();}
  void setP2(Point p2) {
      this.p2 = p2;
      Display.updating();}
}
```
(a)

```
class Point
{
  int x, int y;

 int getX() { return x};
 int getY() { return y}

 void setX(int x) {
      this.x = x;
      Display.updating();}
 void setY(int y) {
      this.y = y;
      Display.updating()}
}
```
(b)

```
aspect DisplayUpdating
{
    pointcut move():
    call(void Line.set*(Point)) || call(void Point.set*(int));

    after() returning: move() {
    Display.update();
  }
}
```
Pontos de junção

(c)

Figura 2.5 - Implementação do Mecanismo de Atualização do Display.

Considerando novamente o exemplo da Figura 2.2 da Seção 2.5, o problema da atualização do display onde o método `Display.updating();` representa o interesse não funcional que está espalhado e entrelaçado no código funcional a Figura 2.5 (a) e (b). Ainda na Figura 2.5 (c) tem-se a solução para o caso acima em que se criou um aspecto, com um ponto de corte denominado `move()` que agrupa dois pontos de junção: o primeiro, refere-se a chamadas a todos os métodos que iniciam com as letras "set" da classe `Line` e; o segundo, tem a função idêntica ao primeiro porém, referem-se a classe `Point`. Quando há uma chamada a algum método `set*`, a sugestão `after` executa o método `update()` da classe `Display` após o término da execução dos métodos chamados. Vale ressaltar que as linhas em negrito na Figura (a) e (b) devem ser retiradas quando esse aspecto é implementado. Dessa forma, as classes `Line` e `Point` contêm apenas os atributos e métodos pertinentes às suas funcionalidades, eliminando completamente chamadas ao método `update()` da classe Display.

c) Declarações inter-tipos *(Inter-type declaration*s): são construções que implementam entrecorte estático em AspectJ. Enquanto o entrecorte dinâmico modifica o comportamento de execução do programa, o entrecorte estático altera a estrutura estática dos tipos (classes, interfaces e outros aspectos) e seu comportamento em tempo de compilação. Existem diversos tipos de entrecorte estático: introdução de elementos (*introduction*), modificação de hierarquia de tipos, declaração de erros e avisos em tempo de compilação e suavização de exceção (Laddad, 2003).

O conceito de introdução (*introduction*) permite que atributos, constantes e métodos sejam adicionados às classes entrecortadas, alterando assim a sua estrutura original. Essa operação pode levar à quebra do encapsulamento, tendo em vista que a interface da classe pode vir a ser alterada.

No exemplo da Figura 2.6, o Aspecto `AspectPersistenceClient` adiciona um novo atributo (`con`) à classe `Client`, assim como o método `Save()`. Durante a composição, os elementos serão inseridos na classe original e, dessa forma, estarão disponíveis normalmente, podendo ser utilizados pelo aspecto, pela própria classe ou por classes externas (no caso dos métodos públicos).

```
public aspect AspectPersistenceClient {

      Connection Client.con;

      public void Client.Save(int cond){ ... }
}
```

Figura 2.6 - Trecho de Código Fonte do Aspecto `AspectPersistenceClient`.

AspectJ também permite a modificação da hierarquia de tipos, adicionando herança ou implementação de interfaces às classes entrecortadas. De forma, semelhante à introdução de elementos, a hierarquia é alterada durante a composição e os elementos contidos nas classes pai (ou nas interfaces) ficam disponíveis para as classes filhas. O formato geral de declaração de herança e de interface é o seguinte:

```
declare parents : [expressão] extends | implements [classe ou lista de interfaces]
```

d) Aspectos (Aspects): são unidades de implementação de entrecorte, compostas por pontos de corte, declarações inter-tipo e sugestões (Kiczales outros, 2001a). Além disso, aspectos podem conter construções Java "puras" como variáveis e métodos. Assim como classes, aspectos contém privilégio de acesso, podem herdar de outras classes (ou de aspectos abstratos) e implementar interfaces. Porém, diferentemente de objetos, aspectos não podem ser instanciados pelo programador. As instâncias são criadas automaticamente quando o programa é executado.

Os aspectos devem obrigatoriamente definir locais de entrecorte de forma a injetar interesses em objetos específicos e, assim, não podem existir sozinhos. Por esse motivo Highley e outros, (1999) não consideram os aspectos como unidades funcionais. Em geral, aspectos só possuem funcionalidade quando associados a classes. Objetos, por sua vez, constituem os componentes do AspectJ e tendem a ser unidades funcionais.

Apesar da inclusão de aspectos afetar o comportamento dos objetos no programa, eles não são exigidos para tornar os objetos unidades funcionais.

2.7. Apoio Para a Modelagem na Fase de Projeto no Paradigma Orientado a Aspecto

Embora existam linguagens de programação orientadas a aspectos disponíveis, há carência de uma técnica de notação para apoiar a modelagem do projeto de sistemas que utilizam esse paradigma. Unland e outros (2002) propõem um modelo de projeto orientado a aspectos para o desenvolvimento de sistemas em AspectJ com UML(*Unified Modelling Language*). Esse modelo facilita a percepção de interesses que entrecortam o sistema (*crosscuting*) e traz as vantagens da programação orientada a aspectos para o nível de projeto. Esse modelo de projeto estende a UML com conceitos de projeto orientado a aspectos tais como: *pontos de junção, pontos de corte, sugestões, declarações de inter-tipos, aspectos* e também reproduz o mecanismo de composição. Os aspectos são representados como classes

UML com estereótipo <<*aspect*>>, podem conter atributos e operações e, também, participar em relacionamentos de associação, generalização e dependência. Eles contêm elementos de interesses que entrecortam o sistema que podem alterar a estrutura e o comportamento de um modelo. A estrutura é alterada por meio de *templates*[1] de colaboração, que possuem o estereótipo <<*introduction*>> e o comportamento por meio de operações com estereótipo <<*advice*>>.

Stein e outros (2002) representam o comportamento de um interesse que entrecorta o sistema com diagramas de interação, os quais estão contidos em diagramas de colaboração que possuem operações com estereótipo <<*advice*>>. Essas operações estabelecem um contexto no qual o comportamento de um interesse que entrecorta o sistema é realizado. A especificação do comportamento de um aspecto que entrecorta uma classe do sistema, pode ser vista no exemplo mostrado na Figura 2.7, que contém:

a) Estereótipos tanto para pontos de corte <<pointcuts>> quanto para sugestões <<advice>>, que possui um identificador *before*, *after* ou *around*.

b) *Templates* de colaboração que são representados pelas elipses tracejadas na parte inferior do aspecto. Esses *templates* são utilizados com o estereótipo <<introduction>>

Os autores criam em sua abordagem um mecanismo de composição permitindo que o modelo de projeto orientado a aspectos se transforme em um modelo UML normal.

Pawlak e outros (2002) apresentam uma notação UML para projetar aplicações orientadas a aspectos, sua notação é independente de plataforma, é baseada na linguagem AspectJ e em trabalhos relacionados a componentes de aspectos. Devido a isso, esses autores precisaram definir o termo "métodos de aspecto" (*aspect-methods*), que são os trechos de código executados antes, após ou no lugar de algum ponto de corte, isto é, são as sugestões (*advices*) da linguagem AspectJ. A criação desse termo foi necessária porque em algumas linguagens orientadas a aspectos, como AspectS (Hirschfeld, 2003), as sugestões possuem um nome, diferentemente do que ocorrem na linguagem AspectJ.

[1] Um *template* é um elemento de modelo parametrizado utilizado para gerar outros elementos do modelo através de passagem de parâmetro.

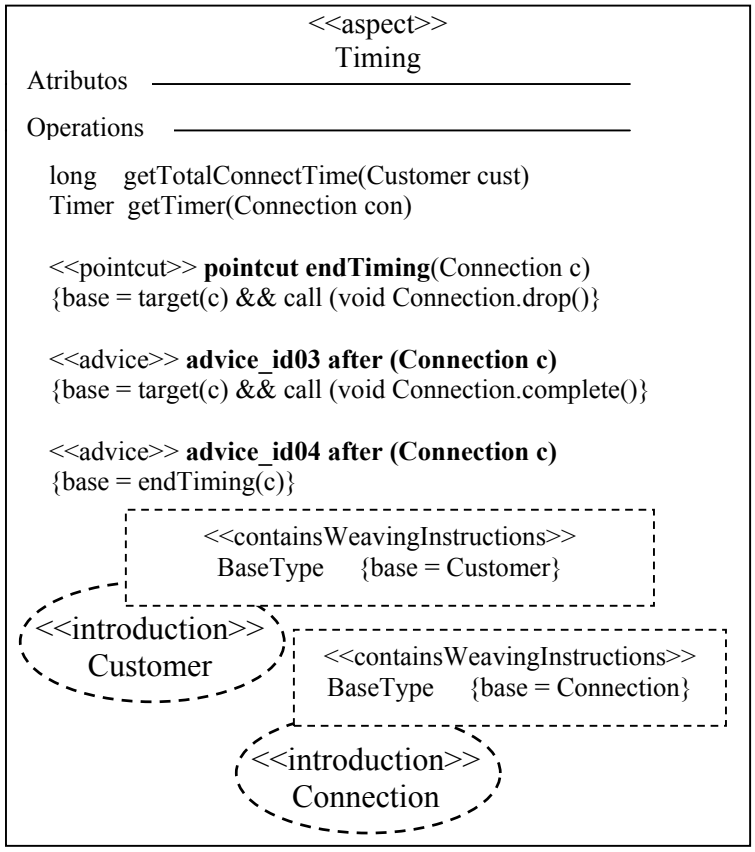


Figura 2.7 - Exemplo da Modelagem de um Aspecto de nome `Timing` com a Notação Proposta por Stein e outros (2002)

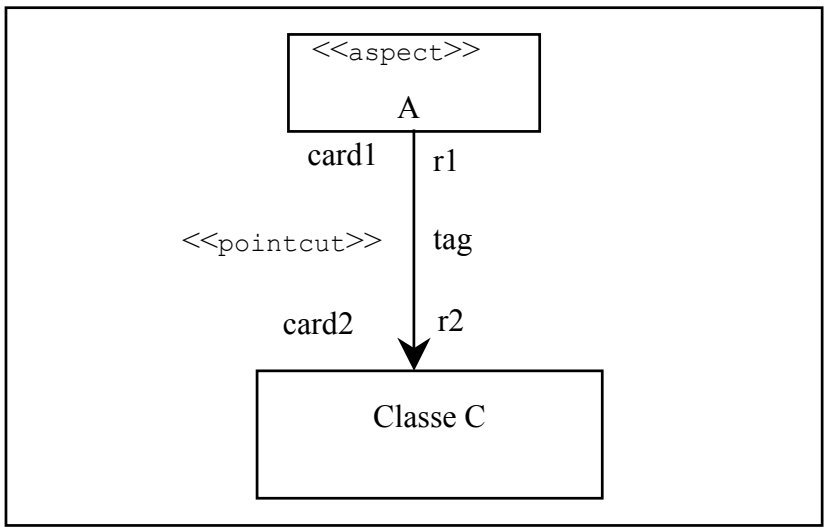


Figura 2.8 – Notação para Sistemas Orientados a Aspectos proposta por Pawlak e outros(2002)

A Figura 2.8 mostra os elementos utilizados por essa notação. O estereótipo <<`aspect`>> é utilizado para representar um aspecto A. O estereótipo <<`pointcut`>> denota uma relação unidirecional do aspecto para a Classe C, indicando que esse aspecto intercepta e interrompe a execução de um método da classe em algum ponto relevante da execução e executa um algum código antes ou depois da execução da classe. O papel "r1" é o nome de um método do aspecto (*aspect-method*) definido no aspecto A que afeta algum ponto dos componentes, definido pelo papel "r2". O rótulo "r2" é definido como um conjunto de pontos de junção que compõem o ponto de corte do Aspecto A. O formato para esse rótulo é do tipo T[expressão], onde T pertence ao conjunto de elementos descritos na Tabela 2.2 e a expressão representa os pontos de junção da classe com o aspecto. As cardinalidades "card1" e "card2" indicam o número de instâncias, tanto dos aspectos quanto das classes e o "tag" é uma etiqueta valorada para estender a semântica do relacionamento.

A Figura 2.9 mostra modelagem de um sistema que realiza consultas em um servidor. Quando o cliente efetua alguma operação que aciona um evento no servidor, a aplicação cliente exibe uma janela solicitando o nome do usuário e a senha. Após o usuário informar esses dados, o servidor invoca um serviço que decide se a requisição do cliente é aceita ou rejeitada.

Tabela 2.2 – Elementos Notacionais para Pontos de Junção.

Elemento	Exemplo	Descrição
!#[2]	!(ClassName.m())	Denota a invocação de um método chamado `m()` da classe `ClassName`.
?	?(ClassName.m())	Denota a execução de um método chamado `m()` da classe `ClassName`.
<N>	<N> (ClassName)	Denota a criação de todas as instâncias da classe `ClassName`.
<U>	<U> (ClassName)	Denota todos os pontos do programa onde uma instância da classe `ClassName` é usada pela primeira vez.
<C>	<C> (ClassName)	Denota todos os pontos do programa em que alguma instância da classe `ClassName` é clonada.
<R>	<R> (ClassName)	Denota os pontos do programa em que uma instância da classe `ClassName` foi remotamente criada.
<E>	<E> (ClassNameException)	Denota todos os pontos do programa em que a exceção `ClassNameException` foi lançada.

[2] Este símbolo foi alterado por uma cerquilha (#) na proposta apresentada por Camargo (2004.).

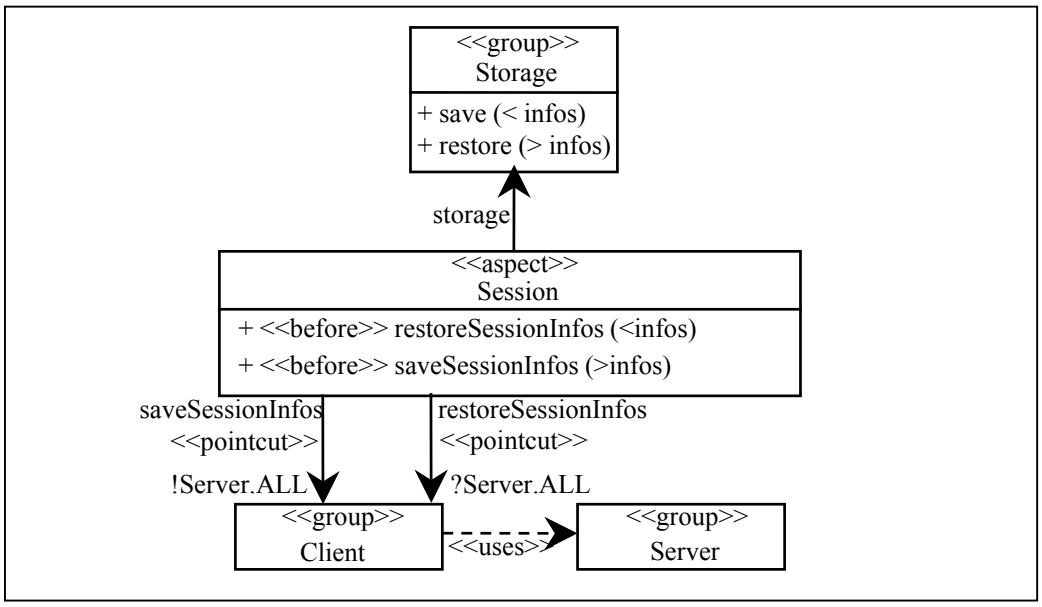


Figura 2.9 - Exemplo da Representação Gráfica de Pawlak (2002).

O aspecto `Session` possui dois métodos de aspectos que são executados antes de qualquer operação ser realizada no servidor (notação <<`before`>>). O aspecto também possui dois relacionamentos com o grupo `Cliente`: o primeiro, intercepta todas (`Server.ALL`), as invocações (!) de métodos do Servidor, e o segundo, todas as execuções (?). Dessa forma, a leitura do modelo leva ao entendimento de que quando houver uma chamada a qualquer operação do servidor, o método de aspecto `saveSessionInfos()` será executado antes dessa chamada e armazenará as informações da sessão com o método `save()` do grupo `Storage`. Da mesma forma, quando houver a execução de qualquer uma das operações do servidor, o método de aspecto `restoreSessionInfos()` será executado antes e restaurará as informações da sessão por meio do método `restore()` do grupo `Storage`.

Camargo (2004.) aponta algumas falhas e alterações na notação proposta por Pawlak (2002):

1. Onde se utilize o papel para o aspecto para descrever os pontos de corte (ou métodos de aspecto, como definido pelos autores) não é conveniente, pois a utilização de papéis possui uma semântica muito bem definida na UML e não deve ser utilizada de forma errada. O mesmo ocorre com a utilização associado à classe para descrever os pontos de junção.
2. O estereótipo <<`pointcut`>> para o relacionamento entre um aspecto e uma classe não parece ser o mais indicado, pois o termo *pointcut* denota o agrupamento de alguns

pontos de junção na linguagem AspectJ e não um relacionamento. Para esse caso, Camargo (2004) sugere utilizar o estereotipo <<croscutting>>.

3. Pawlak e outros (2002) utilizam o sinal de exclamação para denotar chamada de métodos, porém como a linguagem Java utiliza esse símbolo para representar negação, Camargo (2004) sugere substituir por (#).

Camargo (2004) ainda analisa outros trabalhos, e sugere outras notações. Neste trabalho foram utilizadas as notações propostas por Pawlak (2002) com exceção das notações de propostas para as cardinalidades e as alterações 2. e 3. propostas por Camargo (2004).

2.8. Teste Funcional

O teste funcional, caixa preta *(black box*), permite ao engenheiro de software derivar os casos de teste, conjuntos de condições de entrada, que vão exercitar completamente todos os requisitos funcionais de um programa. As categorias de erros mais evidenciadas pelo teste funcional são: erros de interface, funções incorretas ou ausentes, erros nas estruturas de dados ou no acesso a bancos de dados externos, erros de desempenho e erros de iniciação e término (Pressman, 2002). O teste funcional é geralmente aplicado quando todo ou quase todo o sistema já foi desenvolvido.

2.8.1. Teste de Software Orientado a Objetos

No teste de software OO, devido à explícita separação entre especificação e implementação da classe, casos de teste funcional podem ser gerados separadamente de casos de teste que requeiram conhecimento do código da aplicação (teste estrutural). Todas as classes podem ser testadas, mas não todos os objetos. O gerenciamento de mensagens entre objetos é similar ao de chamadas de rotinas, entretanto, a troca de mensagens ocorre mais freqüentemente. Algumas definições e declarações são reutilizadas em vários níveis da árvore de herança, interagindo com novas definições e declarações. De acordo com McGregor (1996), a especificação de sistemas de software OO, a fim de verificar sua completude, deve conter os seguintes itens,

- Especificação de todos os objetos: que enviam e/ou recebem mensagens, que são utilizados como parâmetros de entrada e/ou saída e os objetos de exceção;

- Especificação das pré-condições, pós-condições e invariantes de um método;
- Especificação de todos os métodos de uma classe e seu modelo de estados. Esse modelo fornece informações de como instâncias da classe reagem a estímulos específicos;
- Plano de teste escrito a partir das informações da especificação. O plano deve especificar a extensão do teste, ferramentas que serão utilizadas, critérios de teste, estimativa do tempo necessário para o teste, descrição dos casos de teste e dados de teste associados. Para a obtenção dos casos de teste deve-se selecionar informações com base na seguinte seqüência proposta por McGregor (1996):
    - Desenvolvimento de um conjunto de teste funcional que cubra a especificação completa da classe;
    - Desenvolvimento de casos de teste baseados em estados até que todas as transições no modelo dinâmico sejam cobertas;
    - Utilização de uma ferramenta de cobertura de teste;
    - Desenvolvimento de casos de teste estruturais adicionais para a cobertura de cada linha de código;
    - Desenvolvimento de casos de teste para testar as interações entre métodos de uma mesma classe;
    - Desenvolvimento de casos de teste para a cobertura das interações entre objetos da classe sendo testada e de outras classes.

2.8.2. Teste de Regressão

Um componente novo ou modificado pode apresentar defeitos quando agrupado a outros componentes. Quando isso ocorre, diz-se que o sistema em teste regrediu (Binder, 2000).

Teste de regressão é feito geralmente durante o desenvolvimento de interação, após a depuração de erros, durante a produção de uma nova instanciação ou de um componente reusável, como primeiro passo para integração e para apoiar a manutenção da aplicação. O principal objetivo é garantir que o programa continua a satisfazer os seus requisitos iniciais (Binder, 2000).

O processo de integração empregado no desenvolvimento orientado a objetos necessita de efetivos testes de regressão. Falhas de regressão são introduzidas quando mudanças são

feitas. A essência do desenvolvimento interativo é a mudança de código. Se o teste de regressão não for aplicado a um novo desenvolvimento que sofreu um incremento de componentes, falhas de regressão não são reveladas, pois o foco, desse teste, está no código e nas novas características. Esses erros podem dificultar ou atrasar o desenvolvimento subseqüente de novos incrementos (Binder, 2000).

Neste trabalho é utilizado o teste de regressão para verificar se o sistema apresenta a mesma funcionalidade após a sua reorganização com aspectos.

2.9. Considerações Finais

Este capítulo apresentou os trabalhos relevantes que contribuíram para a realização desta dissertação.

Diagramas de casos de uso foram analisados para apoiar a representação da funcionalidade de sistemas existentes. Neste trabalho diagramas de casos de uso são utilizados para extrair a funcionalidade de sistemas orientados a objetos que não possuem documentação.

As técnicas de separação de interesse constituem o ponto central, pois há interesse em migrar sistemas orientados a objetos para sistemas orientados a aspectos, com a finalidade de eliminar o entrelaçamento e o emaranhamento do código funcional com código não funcional. Assim, obtém-se um código melhor modularizado, que aumenta a manutenibilidade do sistema.

A linguagem AspectJ é a utilizada na migração de sistema. Dessa forma, é necessário criar diretrizes que a partir de um interesse encontrado no código fonte de um sistema, implemente-o como aspecto na linguagem AspectJ.

Para que um projeto seja manutenível é necessário que tenha documentação além do código fonte. Nesse sentido, notações de extensão da UML, para a programação orientada a aspectos, foram analisadas para que se pudesse adotar uma que seja simples e eficiente na elaboração dos diagramas de classes com aspectos.

Testes funcionais são fundamentais para garantir que a funcionalidade do sistema não é alterada com a reorganização de código. Os assuntos apresentados apoiaram a elaboração da abordagem Aspecting, apresentada no Capítulo 4 deste trabalho.

Capítulo 3

Estudos de Casos Utilizados Para a Elaboração da Abordagem Aspecting

3.1. Considerações Iniciais

A elaboração de diretrizes para conduzir a elicitação de aspectos em sistemas orientados a objetos ocorreu com a realização de três estudos de casos. Os sistemas Orientados a Objetos (OO) foram escolhidos em domínios diferentes: um caixa de banco, um sistema de estação flutuante e a versão implementada na linguagem Java do Grep do sistema Unix.

Com a realização desses três estudos de caso observou-se pontos comuns a todos eles, sendo assim, possível elaborar a abordagem *Aspecting*, que será apresentada no Capítulo 4, que tem a finalidade de auxiliar o engenheiro de software a conduzir a elicitação de aspectos a partir de sistemas OO.

Os interesses considerados para os sistemas são os principais encontrados na literatura (Kiczales, 1997), (Rashid e Chitchyan, 2003), (Lippert e Lopes, 2000), não se esgotando todos os existentes.

A Seção 3.2 refere-se ao estudo de caso realizado com um sistema de caixa de banco; a Seção 3.3 foca o estudo de caso realizado com um sistema de estação flutuante; na Seção 3.4 realiza-se o estudo de caso com o sistema GNU Grep.; a Seção 3.5 comenta a construção da Lista de Indícios. A Seção 3.6 discute o processo para a elaboração das diretrizes para a modelagem e implementação dos sistemas Orientados a Aspecto. A Seção 3.7 são apresentadas as considerações finais.

3.2. Sistema de Caixa de banco

O sistema de caixa de banco foi obtido via Internet (Portal Java, 2004), desenvolvido em linguagem Java, utiliza o banco de dados em MS-Access, cuja conexão é feita via JDBC utilizando ODBC com o *driver* que já vem embutido no próprio MS-Access. É composto por 11 classes: 3 contêm as regras de negócio e 8 tratam da interface, que é feita utilizando o pacote `javax.swing` disponível no próprio Java. Neste estudo somente serão implementados em aspectos os interesses não funcionais que estiverem espalhados e entrelaçados nas classes que contém as regras de negócio do sistema. As classes que implementam a interface também não são examinadas.

O sistema não possuía documentação disponível além dos comentários inseridos no próprio código fonte. O entendimento do sistema ocorreu a partir de sua execução gerando um documento contendo os seus requisitos funcionais. O Quadro 3.1 mostra parte do documento de requisitos elaborado para este sistema.

1 - O sistema permite que o usuário cadastre cliente com os atributos: nome de usuário, nome completo, número de identificação, senha, cpf, rg, endereço telefone e nome do pai; 2 - O sistema permite que se cadastre uma ou mais contas para o cliente, com os atributos: saldo inicial e tipo da conta (conta corrente ou poupança); ...

Quadro 3.1 - Exemplo de Parte do Documento de Requisitos Gerado.

De posse dos requisitos passou-se para a geração de um diagrama de classes em nível de implementação para verificar os relacionamentos existentes entre as classes. Esse diagrama foi gerado por meio da inspeção de cada arquivo com extensão `.java` analisando-se os

atributos e os métodos de cada classe existente. Assim, as classes foram adicionadas ao diagrama de classes juntamente com os e seus relacionamentos, Figura 3.1.

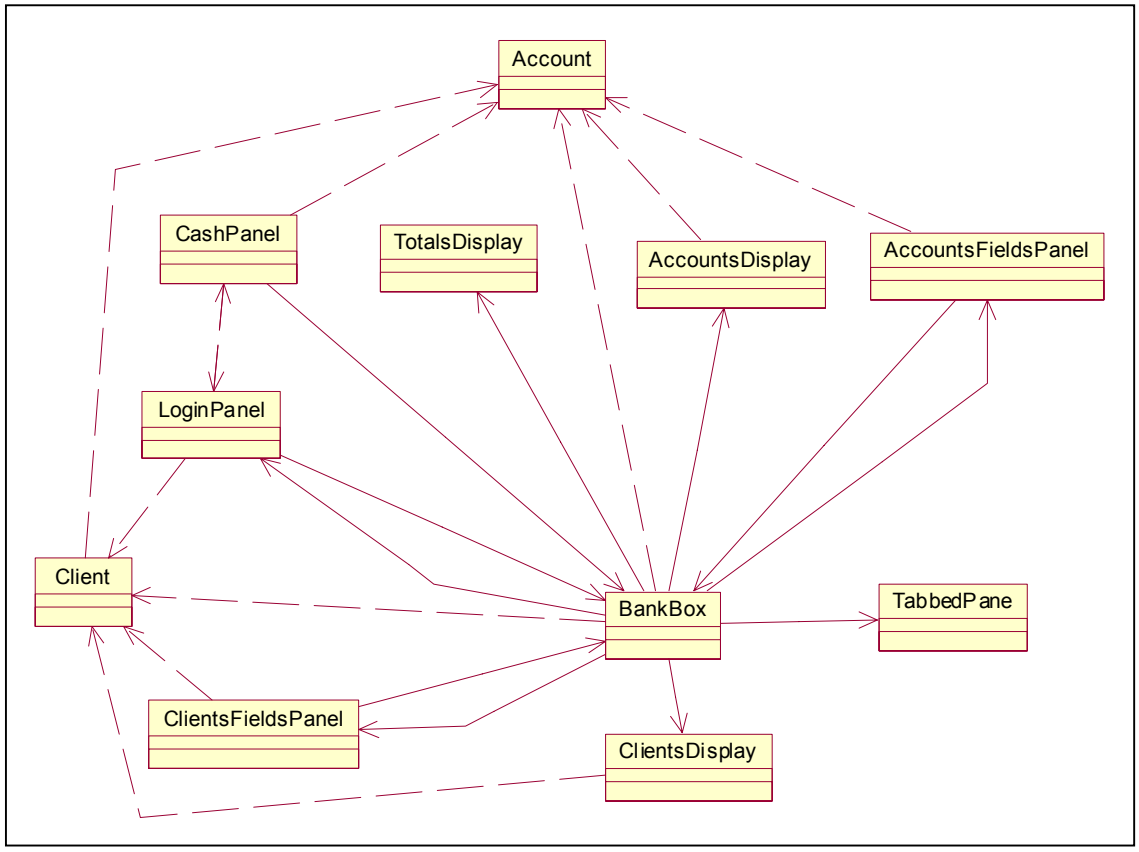


Figura 3.1 - Diagrama de Classes de Implementação do Sistema de Caixa de Banco.

O sistema de Caixa de Banco não continha um conjunto de casos de teste para que se possa aplicar o teste de funcionalidade. No entanto, esse conjunto foi criado a partir da observação da execução do sistema, documentando-se os dados de entrada e os resultados esperados. A Tabela 3.1 exibe o caso de teste funcional quando a opção Cadastrar Cliente é ativada. A primeira coluna mostra os dados de entrada, a segunda coluna os resultados esperados na interface do sistema (para o usuário) e a terceira coluna mostra os resultados esperados no console de programação (para o desenvolvedor). A primeira linha da Tabela 3.1 documenta o comportamento do sistema quando não são inseridos dados e o usuário ativa a opção adicionar cliente, a segunda linha documenta o comportamento do sistema quando aos dados de entrada são dados considerados válidos.

Tabela 3.1 - Casos de Teste Funcional para a Opção Cadastrar Cliente

Valores de Entrada	**Resultados Esperados (Interface com Usuário)**	**Resultados Esperados (Console do Desenvolvedor)**
nulo	Não foi possível adicionar o cliente: Preencha todos os campos ou tente outro Nome de Usuário.	**Exception: java.sql.SQLException: [Microsoft][Driver ODBC para Microsoft Access]**
ID = **1**, Nome de Usuário = **rar**, Nome completo = **Ricardo Argenton Ramos**, Password = **12345**, CPF = **232568974-33**, RG = **25422103-x** , Endereço = **Alameda das Papoulas, 180 - Cidade Jardim - São Carlos**, Telefone = **--** , Nome do Pai = **Paulo Meira Ramos.**	Adicionando cliente: rar... Cliente: rar salvo c/ sucesso.	`SQL do save : UPDATE CLIENTS SET FULL_NAME = 'Ricardo Argenton Ramos' , USER_NAME = 'rar' , PASSWORD = '12345' , CPF = '232568974-33' , IDENTITY = '25422103-x' , ADDRESS = 'Alameda das Papoulas, 180 - Cidade Jardim - São Carlos' , PHONE ='--', NOMEPAI= 'Paulo Meira Ramos' WHERE ID = 1` `UPDATE CLIENTS SET FULL_NAME = 'Ricardo Argenton Ramos' , USER_NAME = 'rar' , PASSWORD = '12345' , CPF = '266935918-33' , IDENTITY = '27700083-x' , ADDRESS = 'Alameda das Papoulas, 180 - Cidade Jardim - São Carlos' , PHONE = 'Paulo Meira Ramos' , NOMEPAI= '' WHERE ID = 1`

Para todas as opções do sistema foram criados casos de teste semelhantes ao mostrado na Tabela 3.1. Esses foram utilizados ao final da reorganização do código do sistema para assegurar que a funcionalidade do mesmo permanecia inalterada.

A seguir, considerando os artigos técnicos existentes na literatura especializada sobre interesses/aspectos, iniciou-se a pesquisa no código fonte por interesses que pudessem estar entrelaçados e espalhados. Dessa forma, foi possível observar indícios da presença do interesse de Persistência em Banco de Dados.

O interesse de Persistência em Banco de Dados é caracterizado por fragmentos de código que contém comandos da linguagem SQL e tipos de dados específicos para realizar a conexão com banco de dados. Para cada um desses trechos de código fonte foram adicionados comentários ao final da de cada linha, indicando o nome desse interesse. A Figura 3.2 exemplifica o código fonte do estudo de caso com interesse de Persistência. Para esse caso o indício foi a presença dos comandos da Linguagem SQL e métodos e objetos do tipo `Connection` e `Statement`.

```
...
    String sql =
      "UPDATE CLIENTS SET FULL_NAME = '"+ fullName + //Persistência
      "' , USER_NAME = '" + userName + // Persistência
      "' , PASSWORD = '" + password + // Persistência
      "' , CPF = '" + CPF + // Persistência
      "' , IDENTITY = '"+ identity + // Persistência
      "' , ADDRESS = '" + address + // Persistência
      "' , PHONE = '"+ phone  + // Persistência
      "' WHERE ID = "+ id; // Persistência
      Statement stm = connection.createStatement(); // Persistência
      System.out.println(sql); // Persistência
      stm.executeUpdate(sql); // Persistência
      stm.close();// Persistência
      }
...
```

Figura 3.2 - Trecho de Código Marcado para o Interesse de Persistência em Banco de Dados Relacional.

O prosseguimento do estudo de caso foi referente à modelagem desse interesse em aspectos. Assim, ao diagrama de classes de implementação, já criado, foram adicionados os aspectos referentes ao interesse.

A decisão de analisar e implementar os aspectos baseou-se em três caminhos sugeridos por Camargo e outros (2003): Atributos que pertencem ao interesse, métodos que pertencem totalmente ao interesse, métodos que pertencem parcialmente ao interesse.

Para as classes Client e Account, que continham atributos e métodos que pertencem ao interesse, optou-se por criar um aspecto específico para cada uma. Ressalta-se que o `AspectBankBox` não contém os métodos do interesse de persistência das classes `Client` e `Account`, pois esses métodos foram alocados nos aspectos, já elaborados, para cada classe específica. Para este estudo de caso não foram encontrados métodos que pertencem parcialmente ao interesse.

A função dos aspectos é a de modularizar os atributos e métodos do interesse e introduzi-los novamente nas classes em tempo de compilação. Os aspectos foram adicionados ao diagrama de classes com o mesmo formato de uma classe, porém com o estereotipo <<`aspect`>>. Os relacionamentos entre classes e aspectos são de associação unidirecional dos aspectos para as classes, com o estereotipo <<`introduction`>>, Figura 3.3. A notação utilizada para esse diagrama foi a proposta por Camargo e outros (2003).

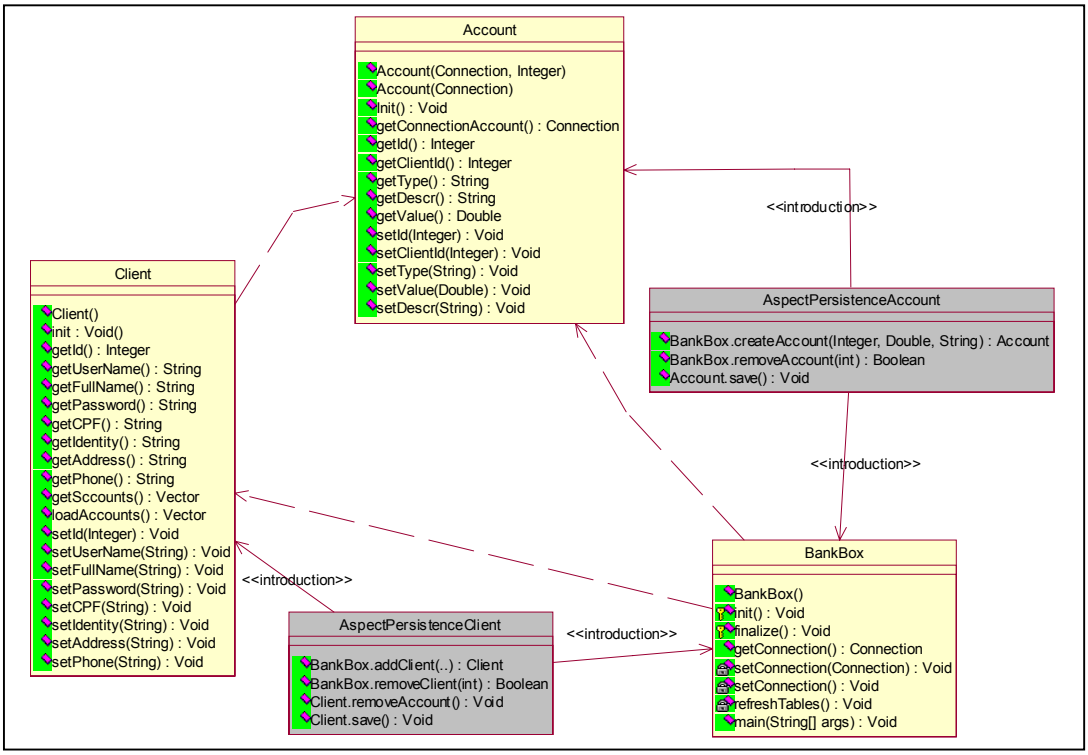


Figura 3.3 - Diagrama de Classes Parcial com o Interesse de Persistência Modelado.

Cada aspecto inserido ao diagrama de classes foi implementado como um aspecto na linguagem AspectJ (Kiczales e outros, 2001a, 2001b). Os atributos e métodos já identificados e marcados nas classes do sistema Orientado a Objetos, foram associados ao aspecto por meio do conceito de Introdução (*Introduction*). Em tempo de compilação (*Weaving*) esses atributos e métodos são re-introduzidos estaticamente na classe do sistema Orientado a Aspectos.

```
public aspect AspectPersistenceClient{

      public void Client.removeAccount() {
            try { ... }
            catch (SQLException e) { ...}}

      public void Client.save() {
            try { ... }
            catch (SQLException e) { ...}}

      public boolean BankBox.removeClient(int aId) {
            try { ... }
            catch (SQLException e) { ...}}
      . . .
}
```

Figura 3.4 - Código Fonte do Aspecto `AspectPersistenceClient`.

A Figura 3.4 mostra o trecho de código fonte do aspecto AspectPersistenceClient. A Figura 3.5 (a) e (b) exibe a classe Client implementada no sistema Orientado a Objetos, com a indicação da retirada dos métodos relativos ao interesse de Persistência em Banco de Dados, que agora passa a compor o aspecto AspectPersistenceClient criado na Figura 3.4. Observando a Figura 3.5 (b) nota-se que a classe Client contém somente os atributos e métodos funcionais.

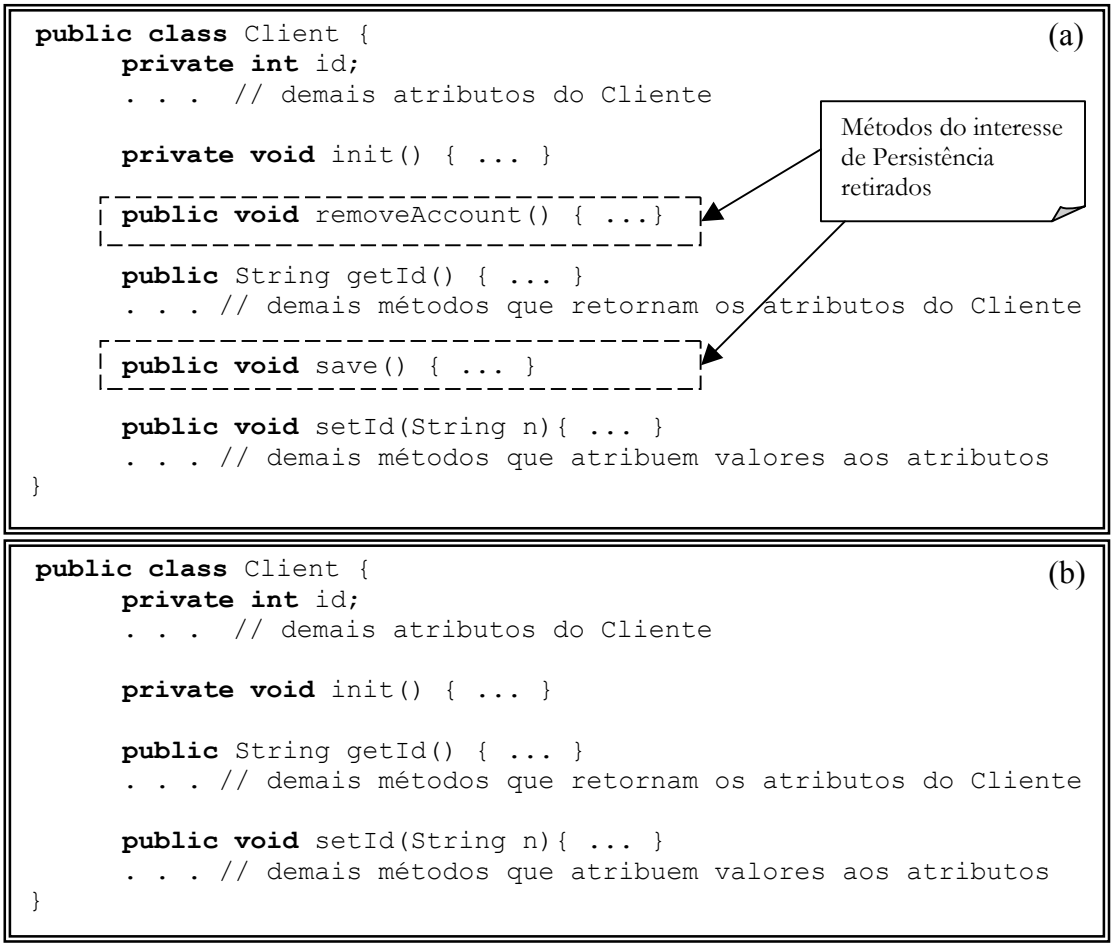


Figura 3.5 - Trechos do Código Fonte da Classe `Client` Original (a) e Sem o Interesse de Persistência (b).

Em seguida, iniciou-se a busca pelo interesse de Tratamento de Exceção que é caracterizado pela presença das palavras reservadas da linguagem Java: `try` e `catch`. A busca Poe essas palavras foi feita nas classes que representam as regras de negócio do sistema e, também, nos aspectos já implementados. Neste exemplo foram encontrados os interesses de Tratamento de Exceção nos aspectos criados anteriormente para o tratamento do interesse de Persistência. Em cada método havia o tratamento de exceção para o caso de algum erro que ocorra quando da conexão com o banco de dados é realizada.

Após essa localização foi adicionado ao diagrama de classes o aspecto que modulariza o interesse, no mesmo formato de uma classe, porém com o estereótipo <<`aspect`>>. O ponto de corte é representado por um "*like*" método com o estereótipo <<`pointcut`>>. Os relacionamentos entre classes e o aspecto foram de dependência unidirecional do aspecto para as classes, com o estereotipo <<`crosscuting`>>, Figura 3.6. Os métodos que continham os interesse de Tratamento de Exceção são definidos como pontos de junção e o interesse de Tratamento de Exceção é adicionado dinamicamente, quando ocorrer a composição (*weaving*), com o código funcional do sistema..

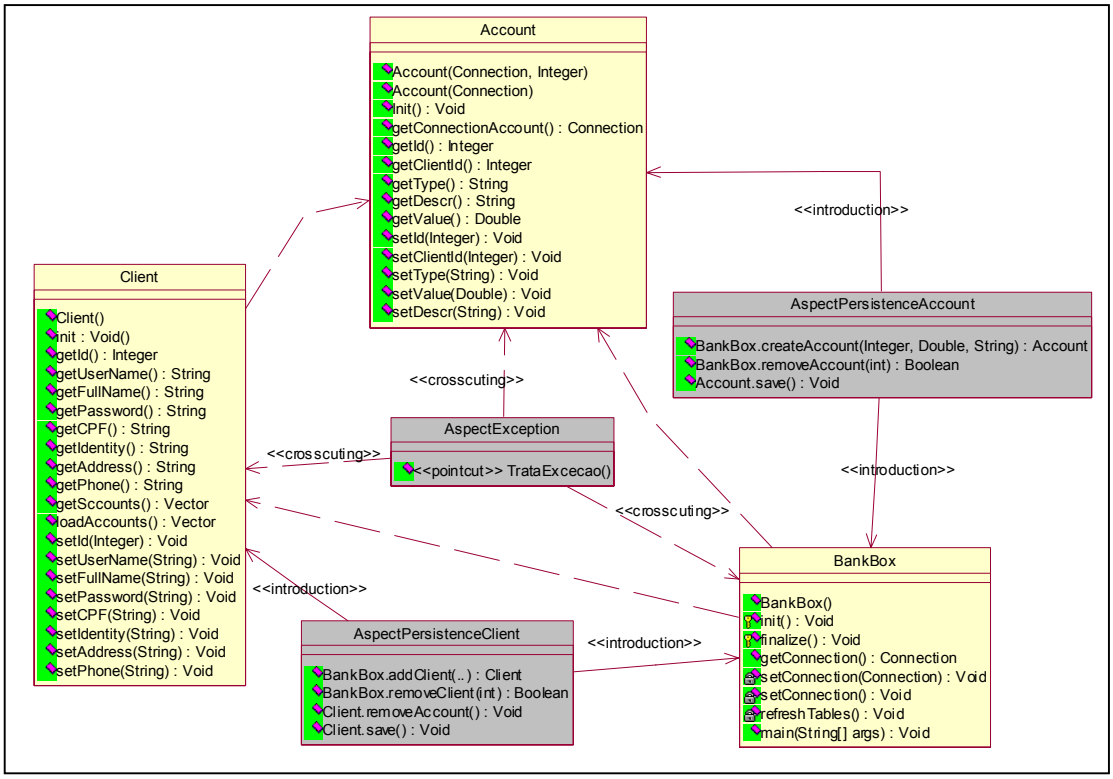


Figura 3.6 - Diagrama de Classes Parcial do Sistema de Caixa de Bancos, com os Aspectos Modelados.

O aspecto para o tratamento de exceções geradas do tipo `SQLException`, foram implementados com os pontos de junção (*join point*) que compõem o ponto de corte (*pointcut*) deste aspecto os métodos que contém em seu corpo as palavras chaves `try` e `catch`, Figura 3.7.

```
public aspect AspectException {
      pointcut TrataSQLException(): execution(* *save())
      || execution(* *createAccount(..))
      || execution(* *removeAccount(..));
   declare soft: SQLException : TrataSQLException();
   after() throwing(SQLException e): TrataSQLException(){
   try{
      throw new SQLException();
   }
   catch(SQLException e1){
     System.out.println(("Exception : " + e1.toString());
   }
  }
}
```

Figura 3.7 - Código Fonte do Aspecto de Tratamento de Exceção.

A Figura 3.8 (a) indica os trechos de códigos Orientados a Objetos retirados da classe `Client` e incorporados ao aspecto `AspectException`. O trecho de código fonte da Figura 3.8 (b) faz parte da implementação Orientada a Aspectos, após a remoção dos interesses.

(a)

```
public void Client.removeAccount() {
  try {
     String sql = "DELETE FROM ACCOUNTS WHERE CLIENT_ID = ?";
     PreparedStatement stm = this.connection.prepareStatement(sql);
     stm.setInt(1, this.getId());
     stm.executeUpdate();
     stm.close();
   }
  catch (SQLException e) {
     System.out.println("Exception : " + e.toString());
  }
   this.loadAccounts();
}
```

Trechos do Interesse de Tratamento de Exceção

(b)

```
public void Client.removeAccount() {
    String sql = "DELETE FROM ACCOUNTS WHERE CLIENT_ID = ?";
    PreparedStatement stm = this.connection.prepareStatement(sql);
    stm.setInt(1, this.getId());
    stm.executeUpdate();
    stm.close();

  this.loadAccounts();
}
```

Figura 3.8 - Código Fonte do Método `removeAccount()` Original (a) e sem o Interesse de Tratamento de Exceção (b).

A última parte do estudo de caso foi verificar se a versão gerada orientada a aspectos contemplava os requisitos funcionais do sistema original. Os casos de teste funcionais foram exercitados com o Sistema Orientado a Aspectos e pôde-se observar que as funcionalidades existentes no sistema Orientado a Objetos foram mantidas. Assim, as duas versões do sistema têm o mesmo comportamento.

3.3. Sistema de Estação Flutuante

O sistema de estação flutuante emula um mecanismo de captação de informações referente à velocidade do vento em alto mar, conforme descrito em (Weiss e Lai, 1999). Esse sistema é desenvolvido na linguagem Java e contêm 12 classes, a interface com o usuário é realizada através de um *applet*, via navegador.

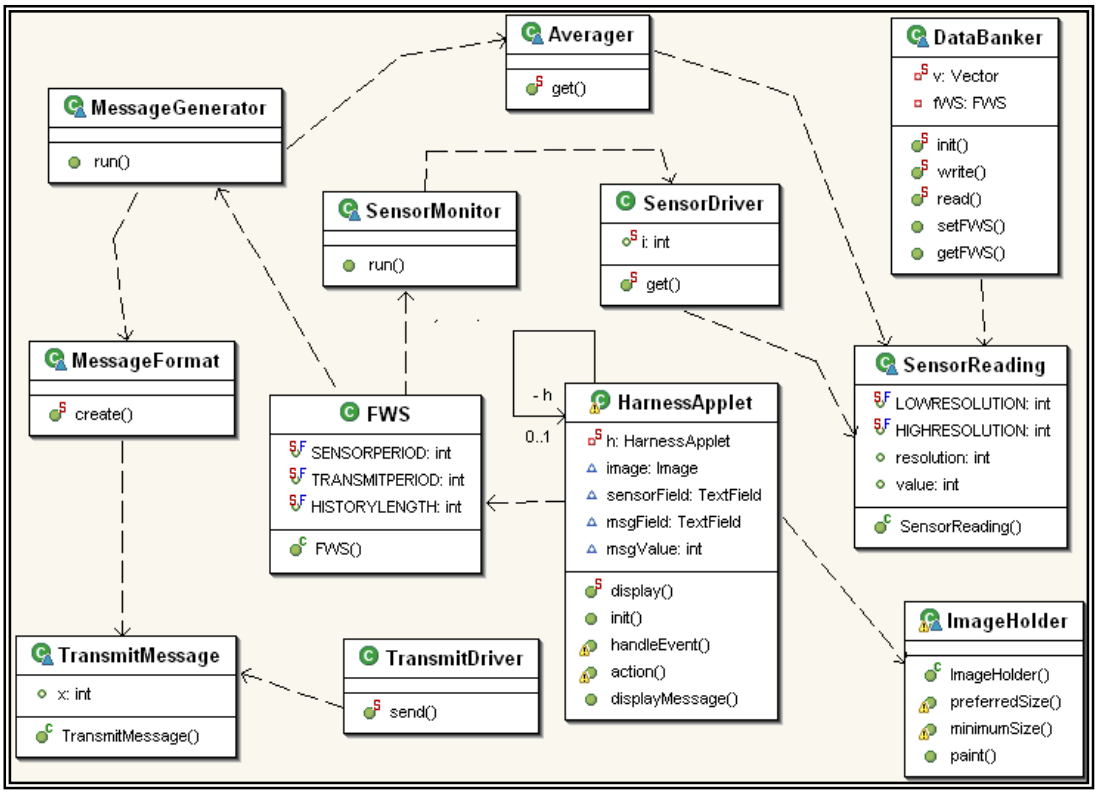


Figura 3.9 - Diagrama de Classes do Sistema de Estação Flutuante Gerado pela Ferramenta *Omondo* (Omondo, 2004).

Para gerar o diagrama de classes em nível de implementação foi utilizada a ferramenta *Omondo* (Omondo, 2004), um *plug-in* da plataforma Eclipse (IBM e outros, 2004) para engenharia reversa/avante utilizando a notação UML. A ferramenta gera o diagrama de classes a partir dos arquivos com extensão `.java` e não `.class` portanto quando existir a

implementação de mais de uma classe no mesmo arquivo, a ferramenta somente considera uma classe. Para solucionar esse problema é necessário reorganizar o código colocando em cada arquivo com extensão `.java` uma única classe. Dessa forma, o diagrama de classes com seus relacionamentos, gerado pela ferramenta *Omondo* (Omondo, 2004), é o exibido na Figura 3.9.

Assim, como o sistema de Caixa de Banco este não continha um conjunto de casos de teste de funcionalidade. Assim, os casos de teste funcionais foram elaborados a partir da execução do sistema, exibidos na Tabela 3.2, documentando-se os dados de entrada e os resultados esperados para esses dados. Esses casos serão utilizados após a reorganização do sistema em AspectJ, para verificar que sua funcionalidade foi preservada. É importante ressaltar que os resultados esperados para cada valor de entrada são produzidos após 5 segundos de espera.

Tabela 3.2 - Casos de Teste Funcional para a Opção Inserir Velocidade

Dados de Entrada	Resultados Esperados
5	5
10	10

```
class SensorMonitor extends Thread
{
  public void run()
  {
    while (true) {
      try { //Tratamento de  Exceção
            Thread.sleep(FWS.SENSORPERIOD);}
      catch (InterruptedException e) {} //Tratamento de  Exceção
      DataBanker.write(SensorDriver.get());
    }
  }
}
```

Figura 3.10 - Código fonte da Classe `SensorMonitor`.

Para a busca de interesses que estavam espalhados no código fonte do sistema foram consideradas somente as classes com métodos que implementam as regras de negócio, desconsiderando as classes relacionadas à interface com o usuário.

O primeiro interesse a ser pesquisado foi o de Tratamento de Exceção. Quando encontrado foram inseridos, no final de cada linha, comentários indicando o seu nome, como mostra a Figura 3.10. Um aspecto foi modelado e inserido no diagrama de classes (duas classes o continham) de implementação para esse interesse, e sua implementação em AspectJ seguiu os mesmos passos descritos no primeiro estudo de caso, Seção 3.2.

O segundo interesse a ser pesquisado foi o de Persistência em Banco de Dados. Para cada classe em observação, buscou-se por atributos de conexão com Banco de Dados do tipo `Connection` e comandos da linguagem SQL. Não foram encontrados esses tipos de atributos no código fonte, descartando-se assim, a presença desse interesse no sistema em estudo.

Analisando-se as classes referentes às regras de negócio e o aspecto implementado, notou-se o interesse de Programação Paralela no código fonte, que é caracterizado pela herança da classe `Thread`, o método `run()` da classe filha que sobrepõe o método `run()` da classe pai e a utilização de métodos da classe `Thread`.

Esse interesse foi encontrado nas classes `SensorMonitor` e `MessageGenerator`, sendo modelado em um aspecto e adicionado ao diagrama de classes de implementação. A função desse aspecto é a de retirar das classes a declaração de herança da classe `Thread` e os métodos a ela pertencentes.

O corpo do método `run()` contém o método `Thread.sleep(FWS.TRANSMITPERIOD)`, que depende parcialmente do interesse, pois seus argumentos são referentes a funcionalidade do sistema, indicando de quanto em quanto tempo deverá ser medida a velocidade do vento. Neste caso, para a separação dos interesses foi criado o método `setPeriod(int i)` no aspecto, que por meio do conceito de introdução (*introduction*) será adicionado à classe em tempo de compilação (*Weaving*). Esse método atribui por passagem de parâmetro o valor interiro ao método `Thread.sleep()`.

Para o aspecto criado no diagrama de classe foi implementado um aspecto que contém a declaração de parentesco das classes `SensorMonitor` e `MessageGenerator` que são filhas da classe `Thread` e o método `setPeriod(int i)` comentado anteriormente, que é inserido nas classes que continham o interesse, Figura 3.11.

```
public aspect AspectThread{
      declare parents:
      SensorMonitor extends Thread;
      declare parents:
      MessageGenerator extends Thread;
      public void MessageGenerator.setPeriod(int i)
      {     Thread.sleep(i);  }
      public void SensorMonitor.setPeriod(int i)
      {     Thread.sleep(i);  }
}
```

Figura 3.11 - Código Fonte do Aspecto `AspectThread`.

A Figura 3.12(a) mostra os trechos de código que tratam do Interesse de Programação Paralela e Tratamento de Exceção no sistema implementado em Java. A figura 3.12(b) mostra a classe SensorMonitor com a retirada desses interesses.

a

```
class SensorMonitor extends Thread
{
    public void run()
    {
       while (true) {
        try { Thread.sleep(FWS.SENSORPERIOD); }
        catch (InterruptedException e) {}
          DataBanker.write(SensorDriver.get());
       }
    }
}
```

Trechos de código do Interesse de Programação Paralela

Trechos de código do Interesse de Tratamento de Exceção

b

```
class SensorMonitor
{
   public void run()
   {
        while (true) {
            setPeriod(FWS.SENSORPERIOD);
            DataBanker.write(SensorDriver.get());
        }
   }
}
```

Figura 3.12 - Código Fonte da Classe `SensorMonitor` Original (a) e sem o Interesse de Programação Paralela e Tratamento de Exceção (b).

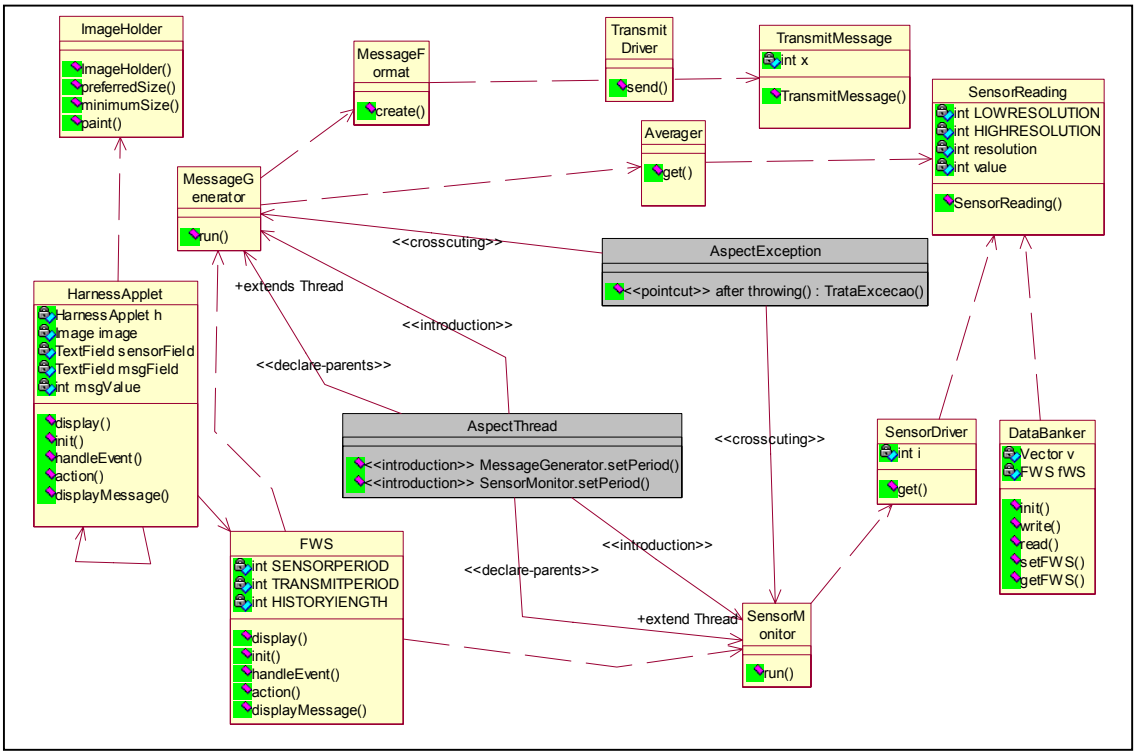


Figura 3.13 – Diagrama de Classes do Sistema de Estação Flutuante com Aspectos.

A Figura 3.13 mostra o diagrama de classes com os aspectos que foram modelados e adicionados para este estudo de caso.

O Sistema Orientado a Aspectos, utilizando os casos de teste funcionais gerados, foi exercitado, para se certificar que a sua funcionalidade permanece inalterada após a reorganização realizada. Todas as saídas previstas nos Casos de Teste foram obtidas, inferindo-se, assim, que a funcionalidade do sistema não foi alterada e que nenhum erro foi inserido com a modularização dos interesses de Tratamento de Erros e Programação Paralela. Ressalta-se que pelo fato do SEF ser um sistema de simulação, com poucas interações do usuário, os casos de teste gerados não são específicos para os interesses de Tratamento de Erros e Programação Paralela.

3.4. Sistema GNU Grep

O sistema foi obtido via Internet (GNU, 2003), sendo sua funcionalidade obtida com a partir da observação de sua execução e assim gerando um documento de requisitos, Quadro 3.2. Após esse entendimento foi gerado o diagrama de classes de implementação, Figura 14, utilizando a ferramenta Omondo (Omondo, 2004), como no segundo estudo de caso.

O Sistema GNU Grep faz parte do sistema RegExp do Unix, sua função é procurar em arquivos por linhas que contém uma correspondência a uma dada lista de padrões de expressões regulares. Quando essa correspondência é encontrada em uma linha, essa é copiada para uma saída padrão, ou é feita qualquer outra combinação de saída que o usuário designar utilizando algumas opções pré-estabelecidas.

Quadro 3.2 - Descrição Resumida da Funcionalidade do Sistema GNU Grep.

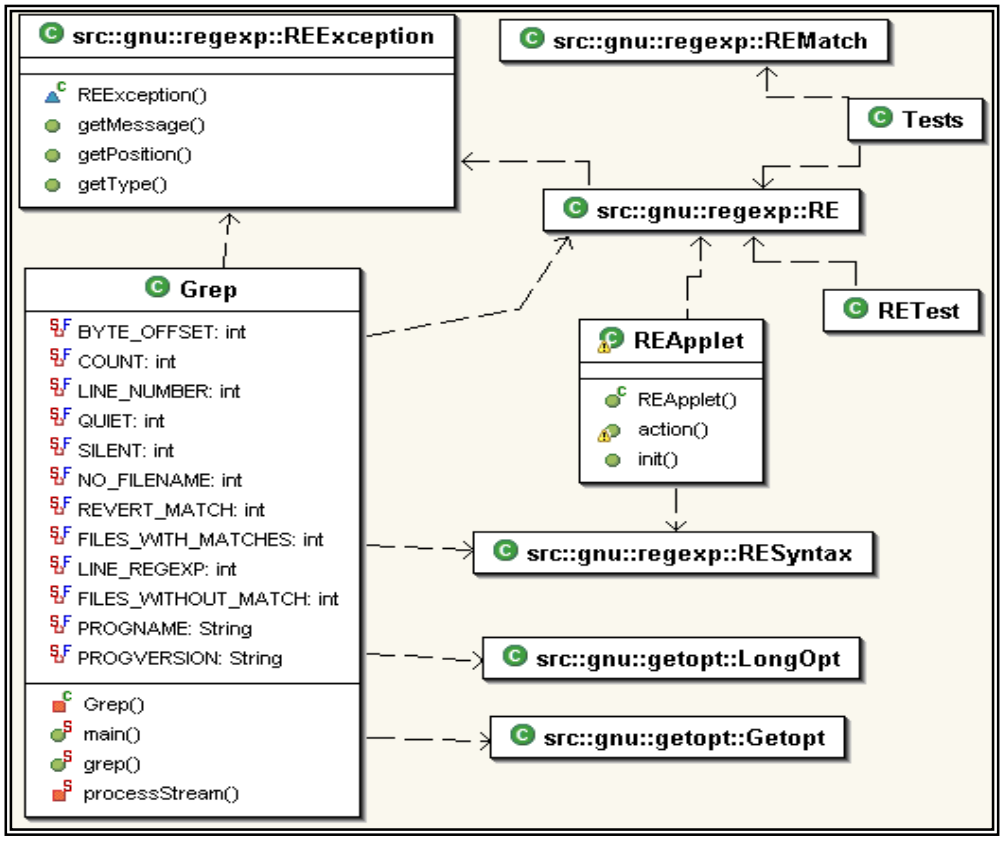


Figura 3.14 - Diagrama de Dependências de Classes (Modelo de Implementação) do Sub-Sistema GNU Grep, Gerado pela Ferramenta *Omondo* (Omondo, 2004).

As classes que contém regras de negócio são analisadas, descartando-se as outras que tratam da interface com o usuário. Para cada classe analisada, foram pesquisados os indícios de Tratamento de Exceção, Persistência em Banco de Dados Relacional e Programação Paralela, já identificados nos estudos de casos anteriores. O interesse de Tratamento de Exceção, neste exemplo, lança mais de um tipo de exceção, conduzindo à modelagem e à implementação de um aspecto para cada tipo.

Analisando-se as classes que contém as regras de negócio verificou-se o entrelaçamento do interesse de Tratamento de Erros com o código funcional. Esse interesse é caracterizado pela utilização de comandos de decisão para comparar uma variável de entrada com um valor pré-fixado. Caso essa variável não seja a esperada ou não esteja no limite previsto, uma mensagem de erro é exibida e/ou a variável é tratada para que se torne válida e/ou então o sistema é finalizado.

O interesse de Tratamento de Erros foi modelado em um aspecto e adicionado ao diagrama de classes de implementação, Figura 3.15. O ponto de junção desse interesse é o método construtor da classe, que passa a variável que está sendo comparada pelos comandos de decisão.

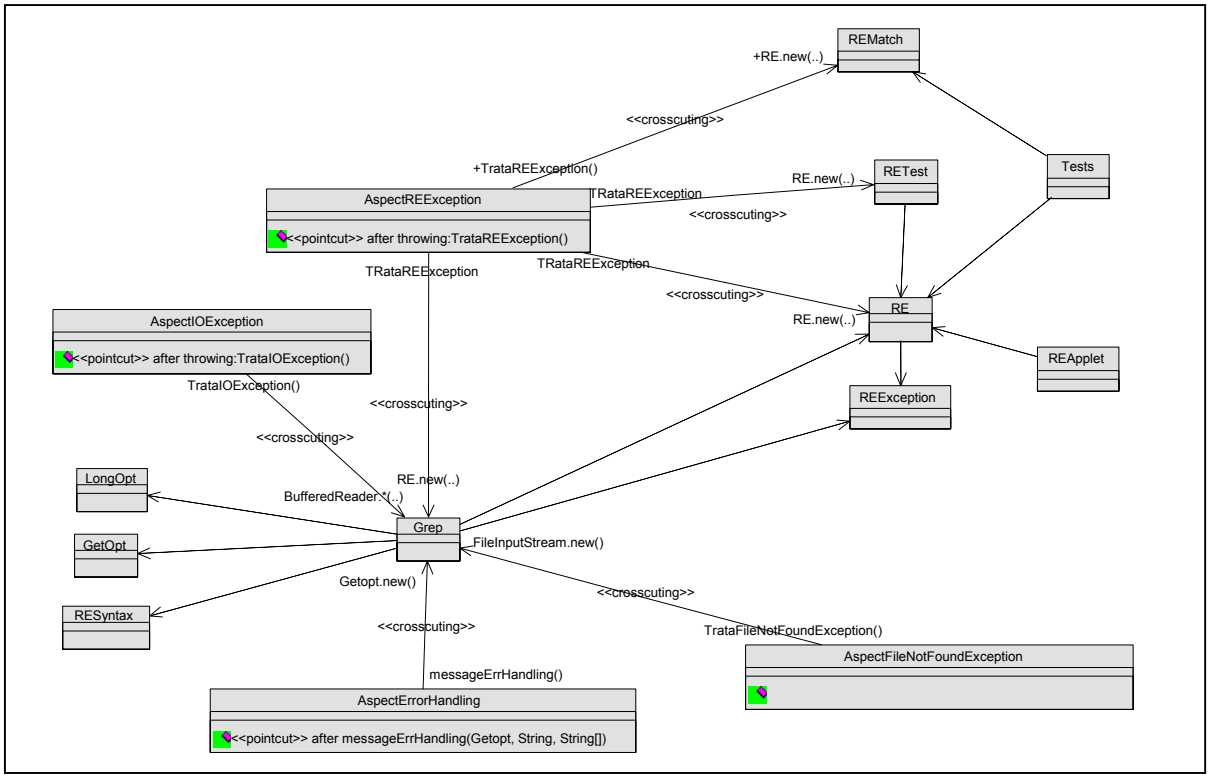


Figura 3.15 - Diagrama de Classes do Sistema GNU Grep com os Interesses de Tratamento de Exceção e de Tratamento de Erros.

Para o aspecto modelado no diagrama de classes de implementação foi implementado o aspecto `AspectErrorHandling`. O ponto de corte desse aspecto necessita obter alguns valores passados por parâmetros em tempo de execução, tanto para utilizá-los no comando de decisão, Figura 3.16 (a), quanto para imprimi-los na mensagem de erro, Figura 3.16 (b).

```
public aspect AspectErrorHandling {

pointcut messageErrHandling(Getopt gtOp, String PRNAME, String[] ar ):
target(gtOp) && args ( PRNAME, ar ) &&
call(Getopt.new(String, String[], String, LongOpt[]));

  after (Getopt gtOp, String PRNAME, String[] ar ) :
  messageErrHandling(gtOp, PRNAME, ar ) {
    if (gtOp.getOptind() >= ar.length) {
      System.err.println("Usage: java " + PRNAME + " [OPTION]...
PATTERN [FILE]...");
      System.err.println("Try `java " + PRNAME + " --help' for more
information.");
    }
  }
}
```

(b) (a) (b)

Figura 3.16 - Código Fonte do Aspecto `AspectErrorHandling`.

O próximo interesse encontrado espalhado e entrelaçado foi o de Persistência em Memória Temporária (*Buffering*), que é caracterizado pela presença dos métodos e/ou objetos que pertencem às classes `BufferredReader` e `StringBuffer`. Como nos casos anteriores a busca foi feita por todas as classes referentes às regras de negócio marcando-se esse interesse.

Para esse interesse há dois tipos de tratamento:

a) Para os atributos e métodos que pertencem totalmente ao interesse: modela-se um aspecto, adicionando-o ao diagrama de classes de implementação com esses atributos e métodos que através do recurso de introdução (*introduction*) serão re-inseridos estaticamente às classes em tempo de compilação (*Weaving*).

b) Para os métodos que pertencem parcialmente ao interesse: modela-se um aspecto, adicionando-o ao diagrama de classes de implementação e cria-se um ponto de corte que utiliza, como ponto de junção, o próprio método ou o método construtor que continha o interesse. A Figura 3.17 mostra o diagrama de classes com os aspectos modelados.

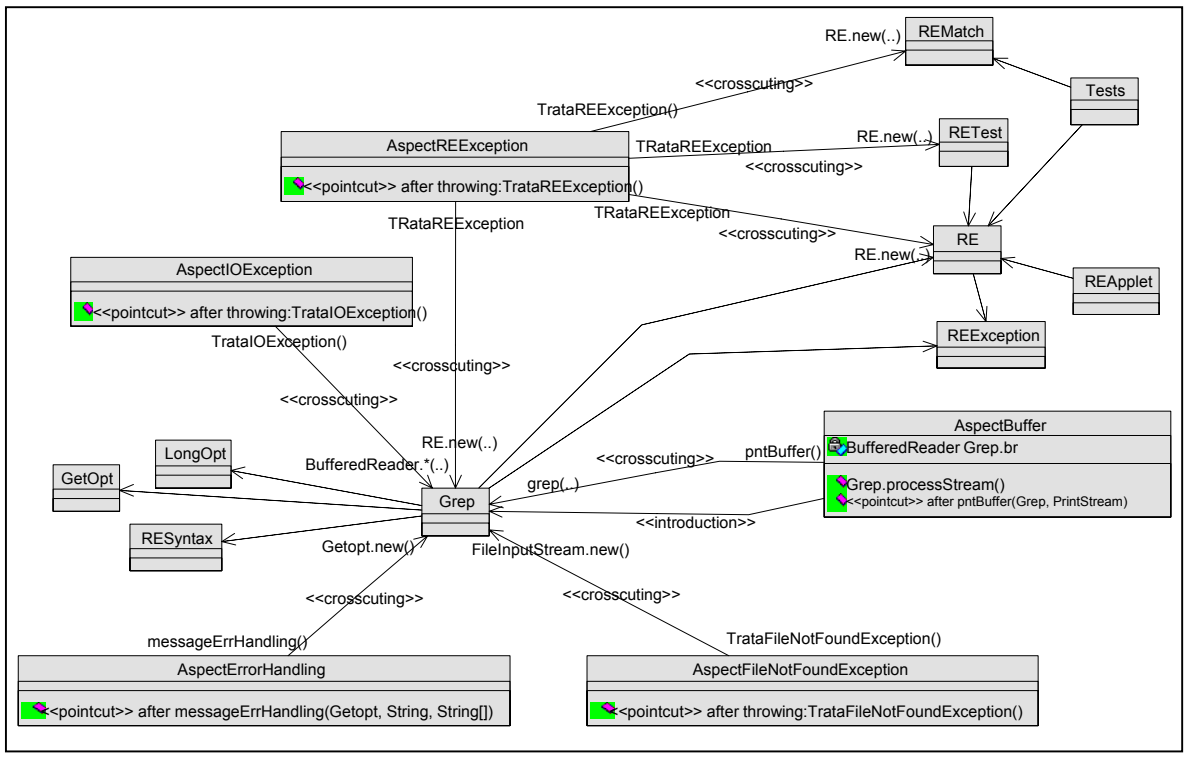


Figura 3.17 - Diagrama de Classes do Sistema GNU Grep com os Aspectos Inseridos.

Para o aspecto e Persistência em Memória Temporária modelado foi implementado o aspecto `AspectBuffer`, Figura 3.18, que contém o atributo `br` e o método `processStream()` que serão inseridos na classe `Grep` quando ocorrer o `Weaving`. O ponto de corte `pntBuffer()` contém o método `grep()` como ponto de junção.

```
public aspect AspectBuffer {

   BufferedReader Grep.br = new BufferedReader(new
InputStreamReader((Grep.class).getResourceAsStream("GrepUsage.txt")));
   String line;

   public static boolean Grep.processStream(RE pattern, InputStream
is, boolean[] options, String filename, PrintStream out)
   {   ...   }

pointcut pntBuffer(Grep g, PrintStream prOut):
   call(public static int Grep.grep(String[], RESyntax, PrintStream))
   && target(g) && args(prOut) && if (Grep.retTrue());

     after(Grep g, PrintStream prOut): pntBuffer(g, prOut){. . .}
}
```

Figura 3.18 - Trecho de código do Aspecto `AspectBuffer`.

Com esses estudos observou-se que uma lista contendo os indícios da existência de interesses auxilia o engenheiro de software na identificação desses quando de posse do código fonte. Assim, uma lista chamada de Lista de Indícios foi criada e o processo utilizado para a sua criação é descrito a seguir.

3.5. Lista de Indícios

Os interesses espalhados no código fonte Java são compostos por fragmentos de código, que podem ser: palavras chaves, atributos, métodos, classes e objetos. Observando-se essas características no código fonte, foi possível elaborar uma lista, chamada de Lista de Indícios, para auxiliar o engenheiro de software a identificar os trechos de código fonte que podem conter indícios de ser de um determinado interesse.

Essa lista auxilia o engenheiro de software a identificar sete tipos de interesses básicos, portanto ela não é completa, não atingindo a todos os interesses considerados básicos (Kiczales, 1997). É composta de dois tipos de indícios:

1 - Para interesses que são sensíveis ao contexto: que contêm fragmentos de código fonte específicos do interesse e estão em um determinado contexto. Esse tipo é utilizado quando, no

código fonte, não existem palavras reservadas que possam induzir o engenheiro de software para um determinado tipo de interesse.

2 - Para interesses não sensíveis ao contexto: quando existem palavras reservadas, no código fonte, que induzem o engenheiro de software a um determinado tipo de interesse. Nesse caso foram criadas expressões regulares que contêm essas palavras reservadas.

Além dessa divisão de tipos de indícios, a Lista de Indícios contém interesses de três níveis, (Kiczales, 1997): de Desenvolvimento, de Produção e de Tempo de Execução. As subseções 3.5.1 a 3.5.3 são dedicadas a cada um desses níveis.

3.5.1. Interesses de Desenvolvimento

São os interesses inseridos pelo desenvolvedor durante o desenvolvimento de aplicações, para facilitar a descoberta de erros, testes e desempenho (Kiczales, 1997). A Lista de Indícios para esse nível contém o seguinte tipo de interesse:

- **de Rastreamento (*tracing*):** utilizado para que o desenvolvedor de software possa acompanhar a execução do sistema. Normalmente implementado por meio de métodos que retornam tanto mensagens indicando o nome de métodos que foram invocados/executados quanto o valor de atributos ou mensagens que indicam o ponto de execução. A Figura 3.19 mostra um exemplo de como esse interesse pode ser encontrado espalhado e entrelaçado no código fonte de uma classe de um sistema.

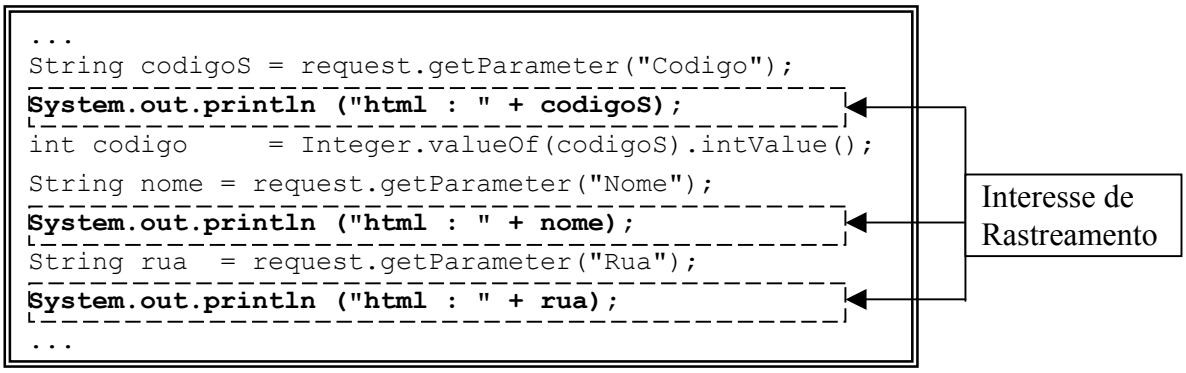


Figura 3.19 - Exemplo do Interesse de Rastreamento Espalhado e Entrelaçado no Código Fonte.

Os indícios para reconhecer o interesse de Rastreamento são dependentes do contexto, pois as mensagens impressas pelos métodos têm a intenção de informar o desenvolvedor do software e não ao usuário final do sistema. O engenheiro de software pode reconhecer indícios desse interesse, se estiver espalhado e entrelaçado no código fonte, métodos do tipo:

```
System.out.println("Mensagem>");
System.out.print("<Mensagem>");
```

<Mensagem> contém informações de nomes ou valores dos métodos, ou mensagens indicando a posição física do método no código fonte.

Outros exemplos para o nível de Interesses de Desenvolvimento são *Profiling*, *Logging*, Gerenciamento de Configuração, entre outros, porém não fazem parte da Lista de Indícios criada por não terem sido encontrados nos estudos de caso realizados.

3.5.2. Interesses de Produção

São interesses utilizados para adicionar, em uma aplicação, construções relativas a produção, ou seja, são trechos de código adicionados pelo desenvolvedor para auxiliar na descoberta de possíveis erros relacionados às classes da aplicação. Geralmente, afetam somente um pequeno número de métodos (Kiczales, 1997**)**. A Lista de Indícios para esse nível contém os seguintes tipos de interesses:

- de Tratamento de Erros: tem a função de tratar os erros que são gerados em tempo de execução quando ocorrer uma entrada inválida no sistema. Após o tratamento, a execução segue e/ou pode imprimir uma mensagem ao usuário do sistema para que uma outra entrada válida seja inserida e/ou que o sistema seja finalizado.

O interesse pode ser reconhecido por comandos de decisão que têm a função de comparar uma variável com determinado valor. Caso o resultado não seja o esperado ou não esteja no limite previsto, uma mensagem de erro é exibida e/ou a variável é tratada para que se torne válida e/ou o sistema é finalizado. Na Figura 3.20, o trecho em negrito exemplifica esse interesse espalhado e entrelaçado em classes de um sistema.

```
...
Getopt g = new Getopt(PROGNAME, argv, "bchilnqsvxyEFGLV",
longOptions);
...
int optind = g.getOptind();
if (optind >= argv.length) {
   System.err.println("Usage:java" + PROGNAME + "[OPTION]..PATTERN
   [FILE]");
   System.err.println("Try java " + PROGNAME + " --help' for more
   information.");
   return 2;}
...
```

Figura 3.20 - Exemplo do Interesse de Tratamento de Erros Espalhado e Entrelaçado no Código Fonte.

Os indícios para reconhecer o interesse de Tratamento de Erros são dependentes do contexto em que estão inseridos, pois dentro dos blocos desses comandos deverão conter métodos que tratem os erros detectados. Conforme citado anteriormente os indícios do interesse de tratamento de erros é quando no código fonte existir os comandos de decisão do tipo: `if`, `else` e/ou `switch` ; e

Dentro do bloco de decisão há:

- Impressão de mensagens de erros alertando o usuário do sistema de alguma situação e/ou;
- Tratamento da variável de entrada para que essa seja válida para aquele contexto e/ou;
- Finalização do Sistema.

- de Tratamento de Exceção: quando uma mensagem de que alguma exceção ocorreu é lançada, indicando que uma operação em execução não pode ser completada, (Lippert e Lopes, 2000). Essas exceções são implementadas por classes, para alguns tipos já estão contidas nas bibliotecas da linguagem Java, ou então é um caso específico que é desenvolvido pelo engenheiro de software. A Figura 3.21 exibe um exemplo em que esse interesse pode ser encontrado espalhado e entrelaçado em classes de um sistema.

```
...
try{
      Thread.sleep(490);
      System.out.println("Imprimindo mensagem"); }
catch(InterruptedException e){
      System.out.println("erro : "+ e);}
...
```

Figura 3.21 - Exemplo do Interesse de Tratamento de Exceção Espalhado e Entrelaçado no Código Fonte.

Os indícios para reconhecer o interesse de Tratamento de Exceção são compostos por palavras reservadas da linguagem Java, assim não são dependentes do contexto, sendo expressos por meio de uma gramática livre de contexto, mostrada na Figura 3.22.

```
<i.trat.exceção>= 'try' '{' <statements> '}' 'catch' '('<statements>
')' '{'<statements>'}' 'finally' '{' <statements> '}' |
'try' '{' <statements> '}' 'catch' '('<statements> ')'
'{'<statements>'}'
```

Figura 3.22 - Indícios para o Interesse de Tratamento de Exceção.

- de Persistência em Banco de Dados Relacional: referem-se a comandos da linguagem SQL e a objetos específicos da linguagem Java que fazem a conexão com o banco de dados relacional e que executam esses comandos.

A Figura 3.23 exibe um exemplo em que esse interesse é encontrado espalhado e entrelaçado em classes de um sistema.

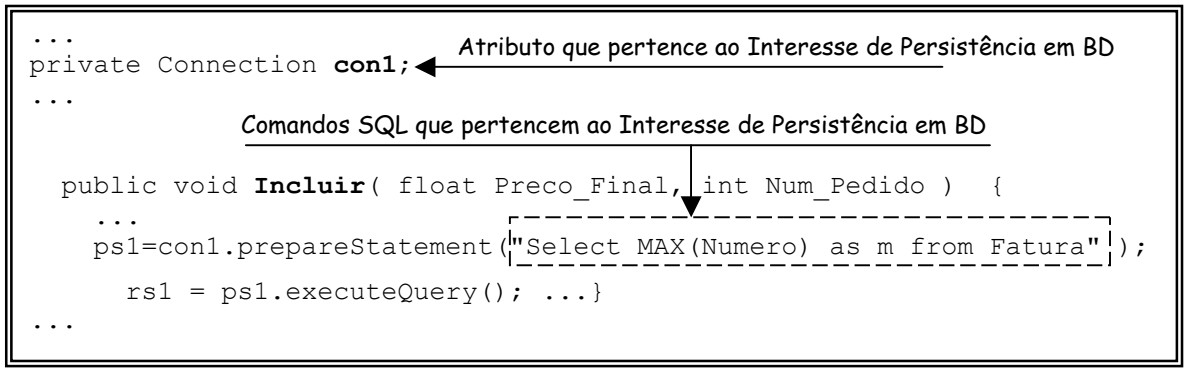


Figura 3.23 - Exemplo do Interesse de Persistência em BD Relacional, Espalhado e Entrelaçado no Código Fonte.

Os indícios para reconhecer o interesse de Persistência em Banco de Dados Relacional não são dependentes do contexto, porém são dependentes de tipos de objetos existentes na linguagem Java, como o objeto `Connection` que pertence ao pacote SQL. Caso esse pacote por algum motivo não esteja presente no sistema em observação, ou esteja alterado, esses indícios não serão válidos. Todo trecho de código fonte, encontrado com as características da gramática livre de contexto apresentada na Figura 3.24, indica que o interesse de Persistência em Banco de Dados Relacional deve ser tratado.

```
<i.Persistencia.BD> = 'Connection' <statements> |
 'Connection' <statements> <SQL>|
 <SQL> <statements> 'Connection'|
'PreparedStatement' <statements>|
'PreparedStatement' <statements> <SQL>|
 <SQL> <statements> 'PreparedStatement'|
'ResultSet' <statements>|
'ResultSet' <statements> <SQL>|
 <SQL> <statements> 'ResultSet'
```

Figura 3.24 - Indícios para o Interesse de Persistência em Banco de Dados Relacional.

Outros exemplos para o nível de Interesses de Produção são, Monitoramento de Mudanças, Sincronização, Passagem de Contexto, porém não fazem parte da Lista de Indícios criada por não terem sido encontrados nos estudos de caso realizados.

3.5.3. Interesses de Tempo de Execução

São interesses utilizados para melhorar o desempenho em tempo de execução. Adicionam ao sistema requisitos não funcionais relativos à otimização do tempo de resposta quando há acesso a dados na memória principal. A Lista de Indícios para esse nível contém os seguintes tipos de interesses:

- de Programação Paralela (*Thread*): refere-se à execução de dois ou mais métodos ao mesmo tempo para atender determinada funcionalidade do sistema (Deitel, 2000). O trecho de código da Figura 3.25 exibe um exemplo de como esse interesse é implementado de forma espalhada e entrelaçada em classes de um sistema.

```
Class SensorMonitor extends Thread
{   public void run()
   {    while (true) {
            try { Thread.sleep(FWS.SENSORPERIOD); }
            catch (InterruptedException e) {}
            DataBanker.write(SensorDriver.get()); }
   }
}
```

Figura 3.25 - Exemplo do Interesse de Programação Paralela, Espalhado e Entrelaçado no Código Fonte de uma Classe.

Os indícios para reconhecer o interesse de Programação Paralela não são dependentes do contexto, porém são dependentes da classe `Thread` e da interface `Runnable` que fazem parte de bibliotecas na linguagem Java. Caso essa classe ou essa interface, por algum motivo, não esteja presente ou esteja modificada, esses indícios não serão válidos. Todos os trechos de código fonte que forem reconhecidos pela gramática livre de contexto da Figura 3.26, indicam ao engenheiro de software a presença do interesse de Programação Paralela.

```
<i.Programacao.Paralela> = 'extends' 'Thread' <statements> 'run()'
'{'<statements> '}' |  'extends' 'Thread' <statements> 'run()' '{'
'Thread' '.' <statements> <statements> '}' | 'extends' 'Thread'
<statements> 'run()' '{' <statements> 'Thread' '.' <statements>
<statements> '}' | 'extends' 'Thread' <statements> 'run()' '{'
<statements> 'Thread' '.' <statements> '}' | 'implements' 'Runnable'
<statements> 'run()' '{'<statements> '}' | 'implements' 'Runnable'
<statements> 'run()' '{' 'Thread' '.' <statements> <statements> '}' |
'implements' 'Runnable' <statements> 'run()' '{' <statements>
'Thread' '.' <statements> <statements> '}' | 'implements' 'Runnable'
<statements> 'run()' '{' <statements> 'Thread' '.' <statements> '}' |
'new' 'Thread' '(' <statements> ')'
```

Figura 3.26 - Indícios para o Interesse de Programação Paralela.

- de Persistência em Memória Temporária (*Buffering*): trata do armazenamento e manipulação de dados em memória temporária. É composto por classes específicas da biblioteca Java que têm a função de manipular cadeias de caracteres em memória temporária (Java, 2004). A Figura 3.27 exibe um exemplo de como esse interesse pode ser encontrado espalhado e entrelaçado em classes de um sistema.

```
...
BufferedReader br = new BufferedReader(new
InputStreamReader((Grep.class).getResourceAsStream("GrepUsage.txt")));
      String line;
      try {
          while ((line = br.readLine()) != null)
          out.println(line);
      } catch (IOException ie) { }
...
```

Figura 3.27 - Exemplo do Interesse de Persistência em Memória Temporária Espalhado e Entrelaçado no Código Fonte.

Os indícios para o reconhecimento do interesse de Persistência em Memória Temporária (*Buffering*) não são dependentes de um contexto, porém são dependentes das classes `BufferredReader` e `StringBuffer` e de seus métodos, que estão presentes nas bibliotecas padrão da linguagem Java. Caso essas classes, por algum motivo, não estejam presentes ou estejam modificadas, esses indícios não serão válidos. Todos os trechos de código fonte que forem reconhecidos pela gramática livre de contexto da Figura 3.28 indicam ao engenheiro de software a presença do interesse de Programação Paralela.

```
<i.persistência.mem.temp> = 'BufferredReader'
<statements> |
'new' 'BufferredReader' '(' <statements> ')'  |
'StringBuffer' <statements> |
'new' 'StringBuffer' '(' <statements> ')'  |
'BufferedInputStream' <statements> |
'new' 'BufferedInputStream' '(' <statements> ')'
```

Figura 3.28 - Indícios para o Interesse de Programação Paralela.

Existem outros exemplos para o nível de Interesses de Tempo de Execução que não fazem parte da lista de indícios criada por não terem sido encontrados nos estudos de caso realizados. A Lista de Indícios completa é apresentada no Apêndice 1.

3.6. Diretrizes para Modelar e para Implementar Aspectos

As diretrizes para modelar foram elaboradas com base em trabalhos de extensão da notação UML para a programação orientada a aspectos (Unland e outros, 2002), (Stein e outros, 2002), (Pawlak e outros, 2002), (Camargo, 2004), e na experiência obtida com a realização de estudos de casos. As diretrizes guiam o engenheiro de software para a elaboração do diagrama de classes de implementação com aspectos que modularizam os interesses reconhecidos do código fonte.

Para cada interesse da Lista de Indícios existe um conjunto de diretrizes específicas de modo a auxiliar o engenheiro de software no processo de identificação dos aspectos que serão modelados e posteriormente implementados. Essas diretrizes são apresentadas no próximo Capítulo.

Hanenberg e Unland (2003) propõem em seu trabalho idiomas que facilitam o desenvolvedor a trabalhar com as novas características inseridas pela linguagem AspectJ (Kiczales e outros 2001a e 2001b). Neste trabalho são fornecidas diretrizes para a implementação de aspectos a partir de experiências adquiridas em alguns trabalhos (Camargo e outros, 2003) e de estudos de caso realizados.

As diretrizes de implementação, como as de modelagem, são específicas de cada interesse e têm como objetivo auxiliar o engenheiro de software a implementar os aspectos na linguagem AspectJ, a partir do código fonte marcado do sistema OO e com base no diagrama de classes de implementação.

3.7. Considerações Finais

Os estudos de caso realizados foram a base para: a elaboração da Lista de Indícios, dos passos realizados na modelagem dos aspectos e para a implementação dos aspectos. Assim, os estudos de caso cuidaram da elicitação de aspectos em sistemas orientados a objetos favorecendo sua implementação para sistemas orientados a aspectos. Analisando o processo realizado em cada um dos estudos foi possível perceber duas fases distintas: a de entendimento do sistema e a responsável por tratar os interesses e se possível implementa-los em aspectos.

O sistema de Caixa de Banco teve dois interesses implementados em aspectos: o de Persistência em Banco de Dados Relacional e o de Tratamento de Exceção. A partir das

experiências obtidas com ele, foram considerados os indícios que ajudam a reconhecer os interesses em outros sistemas, as Diretrizes que auxiliam na Modelagem dos interesses encontrados e as Diretrizes que auxiliam na implementação de aspectos.

O Sistema de Estação Flutuante teve dois interesses implementados em aspectos: o de Tratamento de exceção e o de Programação Paralela, sendo que o primeiro interesse já havia sido analisado no primeiro estudo de caso. Para o segundo, foram abstraídos os indícios que ajudaram a reconhecê-lo quando se encontra espalhado e entrelaçado no código fonte de um outro sistema. Os passos utilizados na modelagem e na implementação serviram como base para gera as Diretrizes de modelagem e de implementação.

O último estudo de caso realizado, Sistema GNU Grep, teve implementado em aspectos os interesses de: Tratamento de Erros, Persistência em Memória Temporária e de Tratamento de Exceção. O mesmo processo de abstração realizado nos outros estudos de caso foi realizado nesse para se obter os indícios e as diretrizes para modelar e implementar os aspectos.

O único interesse que não foi encontrado nos estudos de casos apresentados neste trabalho, foi o interesse de Rastreamento (*tracing*), que foi aproveitado do estudo realizado por Chavez (Chavez e Lucena, 2001). As diretrizes criadas para esse interesse foram elaboradas da mesma forma das demais que fizeram parte dos estudos de caso.

Neste Capítulo não foram tratados sistemas que utilizam padrões de projeto cuja implementação podem ter interesses entrelaçados. Camargo e outros (2003) propõem uma solução para o caso em que o padrão, Camada de Persistência (Yoder e outros, 1998), é utilizado. Noda e Kishi (2001) apresentam a implementação dos vinte e três padrões de projeto propostos por Gamma e outros (1995) em AspectJ e em HyperJ (Tarr e Ossher, 2003) e avaliam os resultados obtidos.

O Capítulo seguinte apresenta a abordagem Aspecting para a migração de sistemas Orientados a Objetos para sistemas Orientados a Aspectos, utilizando a Lista de Indícios aqui elaborada e detalhando os passos para modelagem e implementação de aspectos.

Capítulo 4

Aspecting – Abordagem para Migrar Sistemas Orientados a Objetos para Sistemas Orientados a Aspectos.

4.1. Considerações Iniciais

A necessidade de migrar sistemas Orientados a Objetos para sistemas Orientados a Aspectos de modo ordenado motivou a elaboração da abordagem *Aspecting*, a partir das diretrizes criadas para modelagem e implementação de aspectos efetuados nos estudos de caso relatados no Capítulo anterior.

A abordagem *Aspecting* é apresentada sendo formada por: Diretrizes para extrair a funcionalidade do sistema a partir da sua execução e a representação dessa funcionalidade em casos de uso; Diretrizes criadas para modelar o diagrama de classes, em nível de Implementação, a partir do código fonte do sistema; da utilização da Lista de Indícios para a modelagem e a implementação dos aspectos segundo as diretrizes já descritas no Capítulo 3.

Este Capítulo apresenta na Seção 4.2 a Abordagem *Aspecting*; na Seção 4.3 a Etapa de Entendimento da Funcionalidade; na Seção 4.4 a etapa Tratar Interesses e na Seção 4.5 são apresentadas as considerações finais.

4.2. Abordagem Aspecting

Os sistemas orientados a objetos podem migrar para sistemas orientados a aspectos, com a realização de algumas atividades.

A abordagem *Aspecting* surge como abstração do processo de migração realizado nos estudos de casos apresentados no capítulo 3. Como descrito, cada um dos estudos de casos continha diferentes interesses em sua implementação, o que possibilitou a observação, em diferentes tipos de código fonte, de como se comportavam os requisitos funcionais e não funcionais.

A Abordagem *Aspecting* é composta de duas etapas, de um conjunto de diretrizes que apóia a elicitação de aspectos em sistemas Orientados a Objetos (OO) e a modelagem e a re-implementação desse sistema utilizando conceitos da programação Orientada a Aspectos (AO).

A Figura 4.1 exibe as etapas da abordagem *Aspecting*, representadas por retângulos, utilizando a notação SADT (Ross, 1977). Cada etapa é composta de passos realizados de forma iterativa visando o aperfeiçoamento do produto final. Cada etapa utiliza/produz artefatos que são representados pelas suas entradas e saídas. Os controles são as diretrizes elaboradas para a realização da etapa, enquanto que os mecanismos referem-se às ferramentas que podem auxiliar na sua realização. Na etapa I, por exemplo, tem-se como entrada o Sistema Legado e a possível documentação, se existir, e como saída os Casos de Teste gerados, Diagramas de Casos de Uso e de classes que auxiliaram para o entendimento do sistema original OO. As Diretrizes para extrair os Diagramas de Casos de Uso OO e Diagrama de Classe de Implementação, a partir do código fonte existente, são os controles dessa etapa. As possíveis ferramentas Case que podem auxiliar a execução dessa etapa são representadas pelos mecanismos.

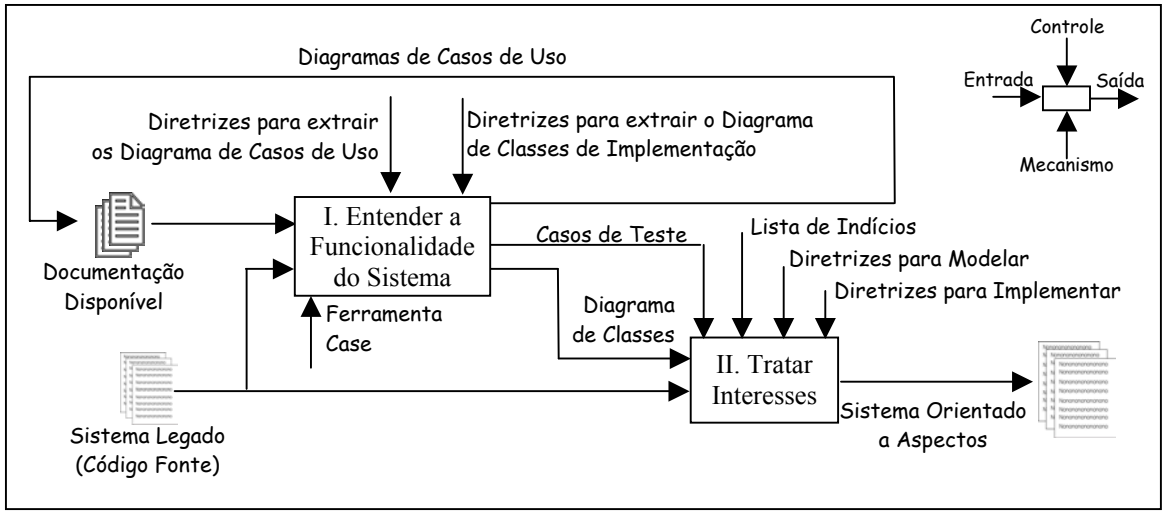


Figura 4.1 - Abordagem Aspecting.

Na etapa II os interesses que se encontram entrelaçados no código fonte com os requisitos funcionais são identificados, modelados e incorporados juntamente com as classes OO da implementação original. Ainda nessa etapa, os Casos de Teste são executados para assegurar que a funcionalidade não foi alterada. O diagrama de classes é a base para a implementação utilizando a linguagem AspectJ. Após a implementação de cada interesse são efetuados os testes de regressão utilizando os Casos de Teste gerados na etapa 1. Diretrizes foram elaboradas em todas as etapas para que o processo possa ser bem conduzido pelo engenheiro de software, mesmo que ele não seja um especialista em programação orientada a aspectos.

4.3. Etapa I: Entender a Funcionalidade do Sistema

Tem o objetivo de extrair informações sobre a funcionalidade do sistema, o relacionamento existente entre as classes e os casos de teste funcional. É composta por três passos: Gerar Diagramas de Casos de Uso, Gerar Casos de Teste e Gerar Diagramas de Classes de Implementação. A entrada para essa etapa é o código fonte do sistema orientado a objetos e a documentação disponível, se existir, e três artefatos são gerados como saída: o diagrama de casos de uso, diagrama de classes de Implementação e os casos de teste funcional.

4.3.1. Passo I.1: Gerar Diagramas de Casos de Uso

Este passo tem o objetivo de gerar os diagramas de casos de uso, se não existirem. Esses diagramas serão utilizados na terceira etapa da abordagem para comparar a funcionalidade do sistema orientado a aspectos gerado com o original, orientado a objetos. A entrada para esse passo é o código fonte do sistema e o artefato gerado como saída é o diagrama de casos de uso.

As diretrizes para gerar diagramas de casos de uso são apresentadas utilizando-se o formato das descrições de casos de uso.

Descrição do Caso de Uso: Extrair os Diagramas de Casos de Uso a partir de um Sistema Orientado a Objetos Implementado na Linguagem Java.

Curso Normal

1. Criar um ator usuário.
2. Identificar os menus existentes no sistema OO como as opções desse sistema.
3. Criar um caso de uso para cada opção do menu.
4. Verificar se toda a funcionalidade foi representada.
5. Encerrar caso de uso.

Cursos Alternativos

2. Não existem menus no sistema.
    - 2.1. Considerar as interações existentes com o usuário como opções de menu.
    - 2.2. Encerrar caso de uso.
4. A funcionalidade não foi verificada completamente
    - 4.1. Identificar as funcionalidades presentes nos sub-menus.
    - 4.2. Criar um caso de uso para cada uma dessas funcionalidades.
    - 4.3. Encerrar caso de uso.

Descrição do Caso de Uso: Criar Relacionamento entre Casos de Uso.

Curso Normal

1. Fixar um caso de uso, identificando se esse chama outro caso de uso, até que todos os casos de uso sejam verificados.
2. Colocar o relacionamento <<*include*>> com origem no caso de uso que chama e com destino no chamado.
3. Encerrar caso de uso.

Cursos alternativos.

2. Caso de uso é chamado opcionalmente.
   2.1. Colocar o relacionamento <<*extends*>> com origem no caso de uso chamado e com destino no que chamou.

4.3.2. Passo I.2: Gerar Casos de Teste

Este passo tem por objetivo gerar casos de teste funcionais, se não existirem, com a finalidade de se certificar que após a reorganização de código o sistema permanece com a funcionalidade inalterada.

A entrada para este passo é a execução do sistema e os casos de uso gerados no passo anterior. O engenheiro de software deve exercitar os Casos de Uso considerando os dados de entrada e os resultados esperados como casos de teste. Uma ferramenta Case como a Jtest (Jtest, 2004) pode ser utilizada para gerar automaticamente os casos de teste funcionais.

4.3.3. Passo I.3: Gerar Diagrama de Classes de Implementação

Este passo tem o objetivo de gerar um diagrama de classes de Implementação, caso esse não exista, expressando as classes e os relacionamentos existentes. A esse diagrama serão acrescidos os aspectos identificados no Passo II.2 da Etapa Tratar Interesse.

A entrada para este passo é o código fonte do sistema e o artefato gerado como saída é o diagrama de classes de Implementação. Uma ferramenta Case[3] que gere automaticamente o diagrama de classes de Implementação pode ser usada, ou o engenheiro de software poderá seguir manualmente as diretrizes criadas para extrair o diagrama de classes a partir do código

[3] A Ferramenta Omondo(Omondo, 2004) foi utilizada em alguns estudos de caso para agilizar esta etapa.

fonte. A seguir são apresentadas as diretrizes para elaborar o diagrama de classes a partir do código fonte Orientado a Objetos implementado em linguagem Java, considerando as classes, as interfaces e os relacionamentos.

Para adicionar as classes do sistema no diagrama de classes.

1. Para cada classe do sistema (`public class` ou `public abstract class`).
    1.1. Criar uma classe correspondente a essa, com o mesmo nome no diagrama.
        1.1.1. Considerar como atributos da classe criada no diagrama de classes cada um dos atributos de tipo primitivo[4] que existirem na classe no código fonte Orientado a Objetos.
        1.1.2. Associar à classe criada no diagrama de classes os métodos, que estão implementados associados à ela no código fonte Orientado a Objeto.

Para adicionar as interfaces do sistema no diagrama de classes.

1. Para cada interface existente no sistema.
    1.1. Criar uma interface correspondente à essa com o mesmo nome no diagrama de classes.
        1.1.1. Associar a interface criada no diagrama de classes os métodos, que estão implementados no código fonte Orientado a Objeto. Associados à interface

Para adicionar os relacionamentos de associação, dependência e de generalização entre as classes criadas no diagrama de classes.

1. Para cada classe do sistema.
    1.1. Se existir atributo de tipo não primitivo da linguagem, para cada atributo.
        1.1. Criar um relacionamento direcional de associação da classe que implementa esse tipo, com a classe que contém esse atributo.

[4] Tipos que são próprios da linguagem Java, como `int, float, char, boolean` entre outros.

1.2. Se existir dentro do método uma instanciação de um objeto, para cada objeto.

1.2.1. Criar um relacionamento direcional de dependência da classe que contém esse método, para a classe que implementa o objeto.

1.3. Se existir na assinatura da classe a palavra reservada `extends` que indica herança.

1.3.1. Criar um relacionamento direcional de generalização dessa classe para a classe pai indicada.

1.4. Se existir na assinatura da classe a palavra reservada `implements` que indica a implementação de uma interface.

1.4.1. Criar um relacionamento direcional de generalização dessa classe para a interface indicada.

4.4. Etapa II: Tratar Interesses

Auxilia o engenheiro de software a marcar, no código fonte Orientado a Objetos (OO), trechos de código que contêm indícios de um determinado interesse que é modelado em aspectos e adicionado ao diagrama de classes de Implementação gerado na primeira etapa dessa abordagem. Após essa modelagem o sistema é implementado na linguagem AspectJ (Kiczales e outros 2001a e 2001b). Como pode ser observado na Figura 4.2, essa etapa é evolutiva, composta por quatro passos que são repetidos até que todos os interesses elencados na Lista de Indícios sejam pesquisados ou até que o engenheiro de software decida finalizar o processo.

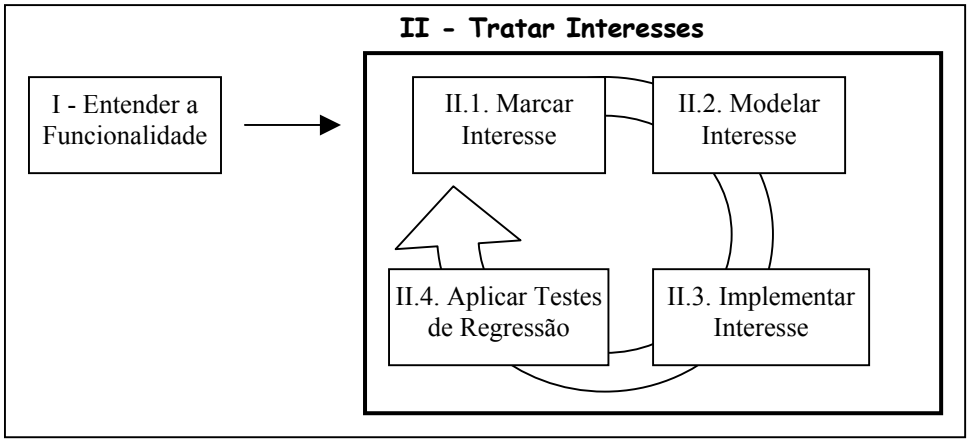


Figura 4.2 - Seqüência das Etapas da Abordagem *Aspecting*.

Os passos da Etapa II, com as respectivas diretrizes, são apresentados nas seções 4.4.1 a 4.4.4.

4.4.1. Passo II.1: Marcar o Interesse

Tem o objetivo de inspecionar o código fonte do sistema a procura de interesses que estejam espalhados e entrelaçados, a partir da Lista de Indícios[5], exibida no Apêndice 1.

Para marcar o código fonte o engenheiro de software deverá escolher um interesse da Lista de Indícios e procurar em cada classe do sistema trechos de código semelhantes aos desses indícios. Ao encontrá-los, todas as linhas desse trecho devem ser marcadas, ou seja, adicionam-se comentários ao final da linha, contendo o nome do interesse ali encontrado seguido de um número seqüencial, que indica a ordem em que o trecho aparece na classe.

O engenheiro de software poderá optar pela utilização de uma ferramenta que o auxilie na busca dos trechos de código que contêm indícios de algum interesse, ou então poderá fazer essa busca manualmente, pesquisando classe por classe os trechos de código. Com a utilização de uma ferramenta é necessário que o engenheiro de software também analise o trecho de código encontrado para confirmar se realmente é um trecho de código válido para a lista de indícios daquele interesse.

A entrada para esse passo é o código fonte Orientado a Objetos (OO) do sistema em análise e o artefato gerado como saída é o mesmo código fonte OO, porém com um interesse marcado.

4.4.2. Passo II.2: Modelar o Interesse

Tem o objetivo de modelar o diagrama de classes com aspectos relativos aos interesses que foram marcados no passo anterior. A entrada para este passo é o diagrama de classes de Implementação, gerado no passo I.2, e trechos do código fonte Orientados a Objetos marcados no passo II.1. O artefato gerado como saída é o diagrama de classes de Implementação, gerado no passo I.2, acrescido da modelagem com aspectos relativo ao interesse.

A Tabela 4.1 apresenta os estereótipos utilizados no diagrama de classes quando aspectos são inseridos. A Tabela 4.2 apresenta a terminologia adotada para a apresentação das diretrizes para modelagem dos diversos tipos de interesse.

[5] Contém indícios para 6 interesses básicos segundo a literatura (Kiczales, 1997).

Tabela 4.1 - Estereótipos Utilizados nas Diretrizes para Modelar.

Estereótipo	Descrição
<<aspect>>	Indica que a estrutura representa um Aspecto.
<<crosscuting>>	Indica que no Aspecto existem pontos de corte que entrecortam as classes associadas ao relacionamento.
<<introduction>>	Indica que o Aspecto introduz atributos e/ou métodos às classes associadas ao relacionamento.
<<declare-parents>>	Indica que no Aspecto estão relacionados às declarações de parentesco das classes associadas ao relacionamento.
<<before>>	Indica que a execução da sugestão (*advice*) deve ocorrer antes da execução ou chamada do ponto de junção.
<<after>>	Indica que a execução da sugestão (*advice*) deve ocorrer após a execução ou chamada do ponto de junção.
<<around>>	Indica que a execução da sugestão (*advice*) deve ocorrer ao mesmo tempo em que a execução ou chamada do ponto de junção ocorrer.
<<target>>	Indica a captura do objeto que recebe a chamada do método (ponto de junção).
<<this>>	Indica a captura do objeto corrente que está associado no ponto de corte.
<<args>>	Indica os argumentos que devem ser capturados pelo ponto de corte.

Tabela 4.2 - Terminologias Utilizados nas Diretrizes para Modelar.

Terminologia	Descrição do Significado
Classe A	Refere-se à classe do diagrama de classes de Implementação para a qual se identificou um interesse.
Atributos da Classe A	Referem-se aos atributos da classe, do diagrama de classes de Implementação, para a qual se identificou um interesse.
Métodos da Classe A	Referem-se aos métodos associados à classe, do diagrama de classes de Implementação, para a qual se identificou um interesse.
Aspect-class	Refere-se à modelagem do interesse ou parte do interesse em aspecto.
Aspect-methods	Referem-se aos pontos de corte do aspecto e a ordem em que esse é executado.
Nome do aspecto	Refere-se ao nome dado ao *Aspect-class* adicionado ao modelo de classes de Implementação. Sugere-se compor o nome do aspecto com o seguinte formato: Aspect+[Nome do interesse que o aspecto modulariza] para aspectos genéricos e; Aspect+[Nome do interesse que o aspecto modulariza]+[Nome da Classe A] para aspectos específicos da Classe A.
Papel do *Aspect-class*	Refere-se ao papel do relacionamento do lado do *Aspect-class*.
Papel da Classe A	Refere-se ao papel do relacionamento do lado da Classe A.
?	Refere-se à captura da execução do método.
#	Refere-se à captura da chamada do método.

Para cada tipo de interesse encontrado há um conjunto de diretrizes para realizar a sua modelagem utilizando um diagrama de classes de Implementação. Essas diretrizes são apresentadas em três partes: a) um resumo, contendo uma explicação sucinta de como a modelagem será tratada, b) um exemplo de diagrama de classes que inclui o interesse que será implementado em aspecto e c) descrição, passo a passo, para que o diagrama de classes da Implementação seja construído.

➲ **Diretrizes para Modelar o Interesse de Rastreamento.**

a) Resumo:

Adicionar ao diagrama de classes um aspecto específico para a modularização do interesse de Rastreamento
Adicionar nesse Aspecto um ponto de corte para cada trecho de código que foi marcado, verificar se o ponto de corte é executado antes, depois ou enquanto o ponto de junção é chamado ou executado

b) Exemplo de Diagrama de Classes de Implementação, Figura 4.3: do interesse de Rastreamento, gerado a partir das diretrizes de modelagem, apresentadas a seguir, e com base no código fonte legado de um sistema OO.

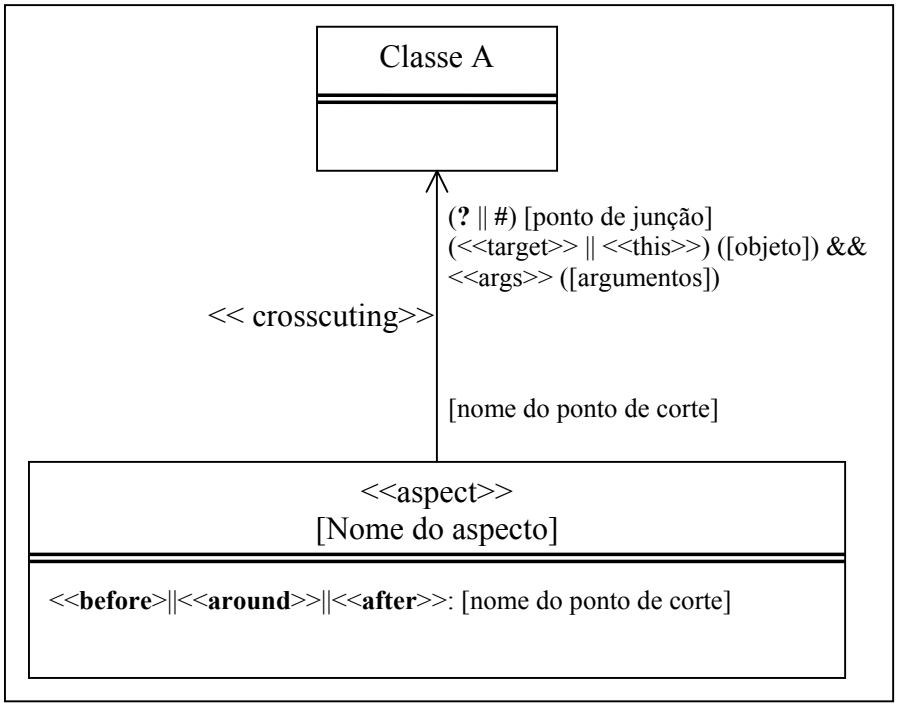


Figura 4.3 - Modelagem de um Aspecto para o Interesse de Rastreamento.

c) Descrição - para modelar o Interesse de Rastreamento**:**

1. Para cada trecho de código fonte marcado e numerado como sendo do interesse de Rastreamento.

1.1. Adicionar ao diagrama de classes de Implementação um *Aspect-class* com o estereotipo <<aspect>>, seguido do Nome do aspecto.

1.1.1. Criar um "*aspect-method*" e adiciona-lo ao *Aspect-class*, utilizando o modelo:

<<before>||<<around>>||<<after>>: [nome do ponto de corte]

Sendo que:

<<before>||<<around>>||<<after>>: escolhido dependendo da ordem em que deseja ser executar o corpo da sugestão (*advice*).

[nome do ponto de corte]: indica o nome do ponto de corte.

1.2. Criar um relacionamento de associação com origem no Aspect-class e destino na Classe A, utilizando o estereótipo <<crosscuting>> .

1.2.1. Criar uma declaração do tipo: [nome do ponto de corte] como papel do *Aspect-class*.

1.2.2. Criar uma declaração do tipo: (**?** || #) [ponto de junção] (<<target>> || <<this>>) ([objeto]) && <<args>> ([argumentos]) como papel da Classe A.

Sendo que:

[nome do ponto de junção]: indica a instanciação do objeto ou a assinatura do método que contém os parâmetros que se deseja capturar.

Caso seja necessário capturar o objeto que esta no ponto de junção:

Sendo que:

(<<target>> || <<this>>) [objeto]: é escolhido dependendo do contexto.

[objeto] indica o nome do objeto que se deseja capturar.

Caso seja necessário capturar os argumento do ponto de junção:

[argumentos] indica os argumentos que se deseja capturar.

➲ Diretrizes para Modelar o Interesse de Tratamento de Erros

a) Resumo:

Adicionar ao diagrama de classes um aspecto especifico para a modularização do interesse de Tratamento de Erros.
Adicionar nesse Aspecto um ponto de corte para cada trecho de código que foi marcado, capturando o método ou objeto que passa o valor que esta sendo utilizado no comando de decisão.

b) Exemplo de Diagrama de Classes de Implementação, Figura 4.4: do interesse de Tratamento de Erros, gerado a partir das diretrizes de modelagem, apresentadas a seguir, e com base no código fonte legado de um sistema OO.

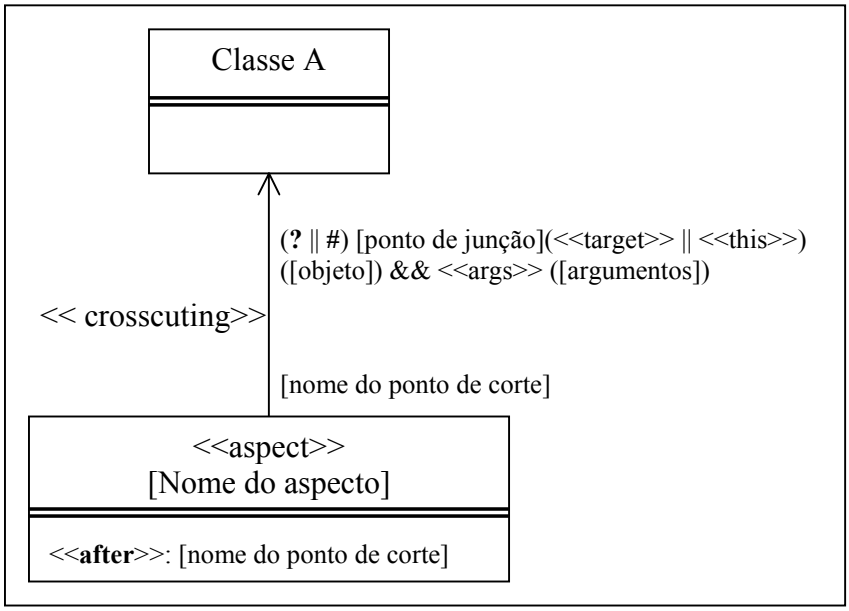


Figura 4.4 - Modelagem de um Aspecto para o Interesse de Tratamento de Erros.

c) Descrição - para modelar o Interesse de Tratamento de Erros**:**

1. Para cada trecho de código fonte marcado e numerado como sendo do interesse de Tratamento de Erros.

1.1. Adicionar ao diagrama de classes de Implementação um *Aspect-class* com o estereotipo <<aspect>>, seguido do Nome do aspecto.

1.1.1. Criar um *aspect-method* e adicioná-lo ao *Aspect-class*, utilizando o modelo: <<**after**>>: [nome do ponto de corte]

Sendo que:

[nome do ponto de corte]: indica o nome do ponto de corte.

1.2. Criar um relacionamento de associação com origem no *Aspect-class* e destino na Classe A, utilizando o estereótipo <<crosscuting>> .

1.2.1. Criar uma declaração do tipo: [nome do ponto de corte] como papel do *Aspect-class*.

1.2.2. Criar uma declaração do tipo: (**?** || #) [ponto de junção] (<<target>> || <<this>>) ([objeto]) && <<args>> ([argumentos]) como papel da Classe A.

Sendo que:

[nome do ponto de junção]: indica a instanciação do objeto ou a assinatura do método que contém os parâmetros utilizados nos comandos de decisão.

(<<target>> || <<this>>): deve-se optar dependendo do contexto.

([objeto]): indica o nome do objeto que se deseja capturar.

&& : conjunção aditiva.

([argumentos]): indica os argumentos, sem indicar seus tipos.

➲ **Diretrizes para Modelar o Interesse de Tratamento de Exceção**

a) Resumo:

Adicionar ao diagrama de classes um aspecto especifico para cada tipo de exceção encontrada nos trechos marcados.
Adicionar nesse Aspecto um ponto de corte para cada trecho de código que foi marcado, capturando o método que lança a exceção e declarando esse como ponto de junção.

b) Exemplo de Diagrama de Classes de Implementação, Figura 4.5: do interesse de Tratamento de Exceção, gerado a partir das diretrizes de modelagem, apresentadas a seguir, e com base no código fonte legado de um sistema OO.

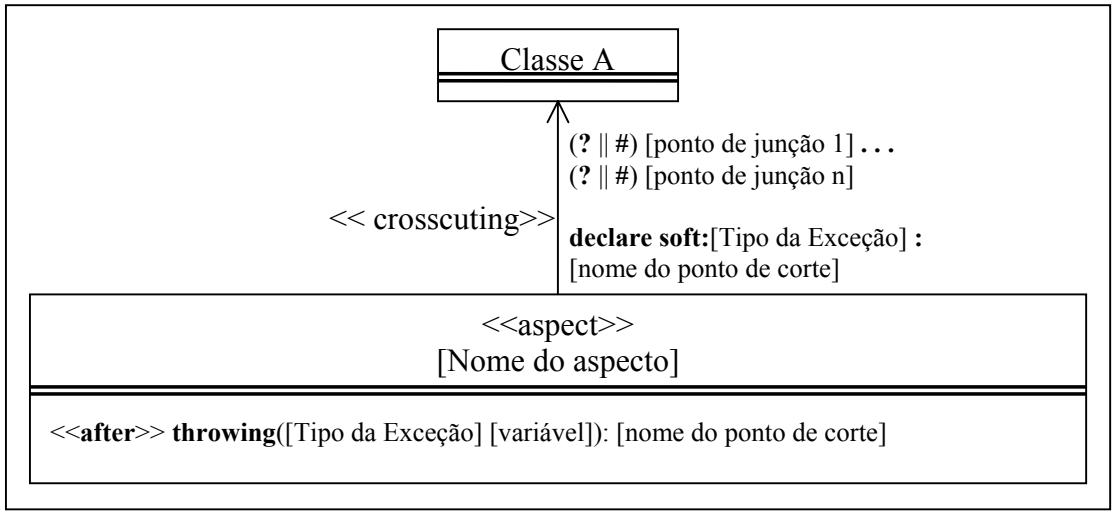


Figura 4.5 - Modelagem de um Aspecto para o Interesse de Tratamento de Exceção.

c) Descrição - para modelar o Interesse de Tratamento de Exceção:

1. Verificar no trecho de código fonte marcado como sendo do interesse de Tratamento de Exceção, qual o tipo de exceção que está sendo tratada. Caso não exista um aspecto para o tipo utilize o Caso I se não utilize o Caso II.

Caso I

I.1.1. Adicionar ao diagrama de classes de Implementação um *Aspect-class* com o estereotipo <<aspect>>, seguido do Nome do aspecto.

I.1.1.1. Criar um "*aspect-method*" e adicioná-lo ao *Aspect-class*, utilizando o modelo:

<<after>> throwing ([Tipo da Exceção] [variável]): [nome do ponto de corte]

Sendo que:

[Tipo da Exceção]: indica o tipo de exceção que é tratado no bloco `try{...} catch(..){...}`.

[variável]: indica nome da variável que armazena a mensagem de erro que foi gerada.

[nome do ponto de corte]: indica o nome do ponto de corte.

I.1.2 Criar um relacionamento de associação com origem no *Aspect-class* e destino na Classe A, utilizando o estereótipo <<crosscuting>> .

I.1.2.1 Criar uma declaração do tipo: declare soft:[Tipo do Interesse] : [nome do ponto de corte] como papel do *Aspect-class*.

I.1.2.2. Criar um a declaração do tipo: (**?** || **#**) [ponto de junção], como papel da Classe A.

Sendo que:

[nome do ponto de junção]: indica a assinatura do método que contém o bloco `try{...} catch(..){...}`.

Caso II (quando já existe um aspecto para o tipo de exceção tratada).

II.1.1. Adicionar ao relacionamento já criado como papel da Classe A, a captura da execução(?) ou da chamada(#) do ponto de junção, que corresponde a assinatura do método que contém o bloco `try{...} catch(..){...}`.

➲ **Diretrizes para modelar o interesse de Persistência em Banco de Dados Relacional.**

a) Resumo:

Adicionar ao diagrama de classes um aspecto especifico para cada classe que faça a Persistência em Banco de Dados Relacional.
Adicionar nesse Aspecto utilizando o conceito de introduções os atributos e métodos da classe que forem específicos do interesse de persistência. Caso o método seja parcialmente do interesse, criar um ponto de corte que capture como ponto de junção esse método.

b) Exemplo de Diagrama de Classes de Implementação, Figura 4.6: do interesse de Persistência em Banco de Dados Relacional, gerado a partir das diretrizes de modelagem, apresentadas a seguir, e com base no código fonte legado de um sistema OO.

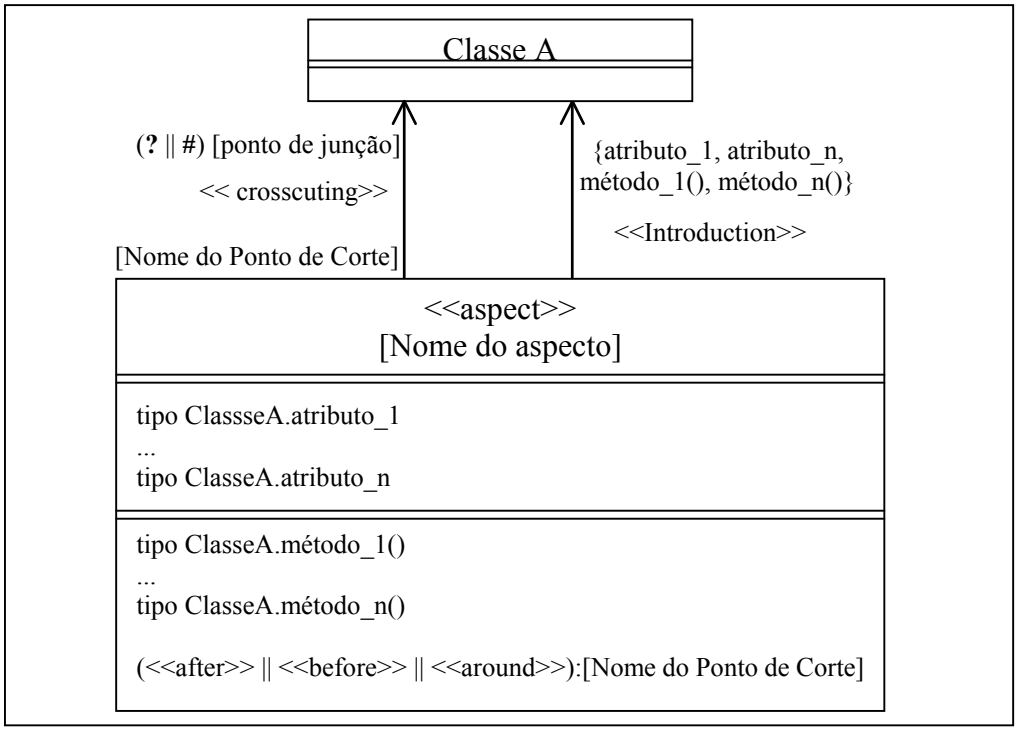


Figura 4.6 – Modelagem de um Aspecto para o Interesse de Persistência em Banco de Dados Relacional.

c) Descrição - para modelar o Interesse de Persistência em Banco de Dados Relacional:

1. Para cada classe que contenha trechos marcados e numerados como sendo do interesse de Persistência em Banco de Dados Relacional.

1.1. Adicionar, ao diagrama de classes de Implementação, um *Aspect-class* com o estereotipo <<aspect>>, seguido do Nome do aspecto.

1.1.1. Adicionar ao *Aspect-class* os atributos do interesse, utilizando o modelo:

[Tipo] [Nome da classe].[Atributo];

Sendo que:

[Tipo]: indica o tipo do atributo que foi retirado da Classe A.

[Nome da classe]: indica o nome da Classe A.

[Atributo]: indica-se o nome do atributo que foi retirado da Classe A.

1.1.2. Adicionar ao *Aspect-class* os métodos que pertencem totalmente ao interesse, utilizando o modelo:

[Tipo] [Nome da classe].[Método];

Sendo que:

[Tipo]: indica o tipo do retorno do método retirado da Classe A.

[Nome da classe]: indica o nome da Classe A.

[Método]: indica o nome do método que retirado da Classe A.

1.1.3. Caso existam métodos que dependam de outros interesses:

1.1.3.1 Criar um "*aspect-method*" e adicioná-lo ao *Aspect-class*, utilizando o modelo:

(<<after>> || <<before>> || <<around>>):[Nome do Ponto de Corte]

Sendo que:

(<<after>> || <<before>> || <<around>>): deve-se optar pela opção dependendo da ordem em que deseja ser executar o corpo da sugestão (*advice*).

[Nome do Ponto de Corte]: Deve-se indicar o nome do ponto de corte.

1.2. Criar um relacionamento de associação com origem no *Aspect-class* e destino na Classe A, utilizando o estereótipo <<Introduction>>, se existirem métodos e/ou atributos relacionados ao interesse.

1.2.1. Criar uma declaração do tipo: {atributo_1, atributo_n, método_1(), método_n()}, como papel da Classe A.

1.3. Criar um relacionamento de associação com origem no *Aspect-class* e destino na Classe A, utilizando o estereótipo <<crosscuting>>, se existirem métodos que dependem de outros interesses.

1.3.1. Criar uma declaração do tipo: [nome do ponto de corte], como papel do *Aspect-class*.

Sendo que:

[nome do ponto de corte]: indica o nome do ponto de corte.

1.3.2.Criar um a declaração do tipo: (**?** || #) [ponto de junção], como papel da Classe A.

Sendo que:

[nome do ponto de junção]: indica a assinatura do método que contém o trecho de código marcado com interesse.

➲ **Diretrizes para Modelar o Interesse de Programação Paralela**

a) Resumo:

Adicionar ao diagrama de classes um aspecto especifico para cada classe que contenha o interesse de Programação Paralela.
Adicionar nesse Aspecto utilizando o conceito de introduções os atributos e métodos da classe que forem específicos do interesse.
Caso o método seja parcialmente do interesse, criar um ponto de corte que capture como ponto de junção esse método.
Criar no Aspecto uma declaração de parentesco para a classe Thread ou para a Interface Runnable.

b) Exemplo de Diagrama de Classes de Implementação, Figura 4.7: do interesse de Programação Paralela, gerado a partir das diretrizes de modelagem, apresentadas a seguir, e com base no código fonte legado de um sistema OO.

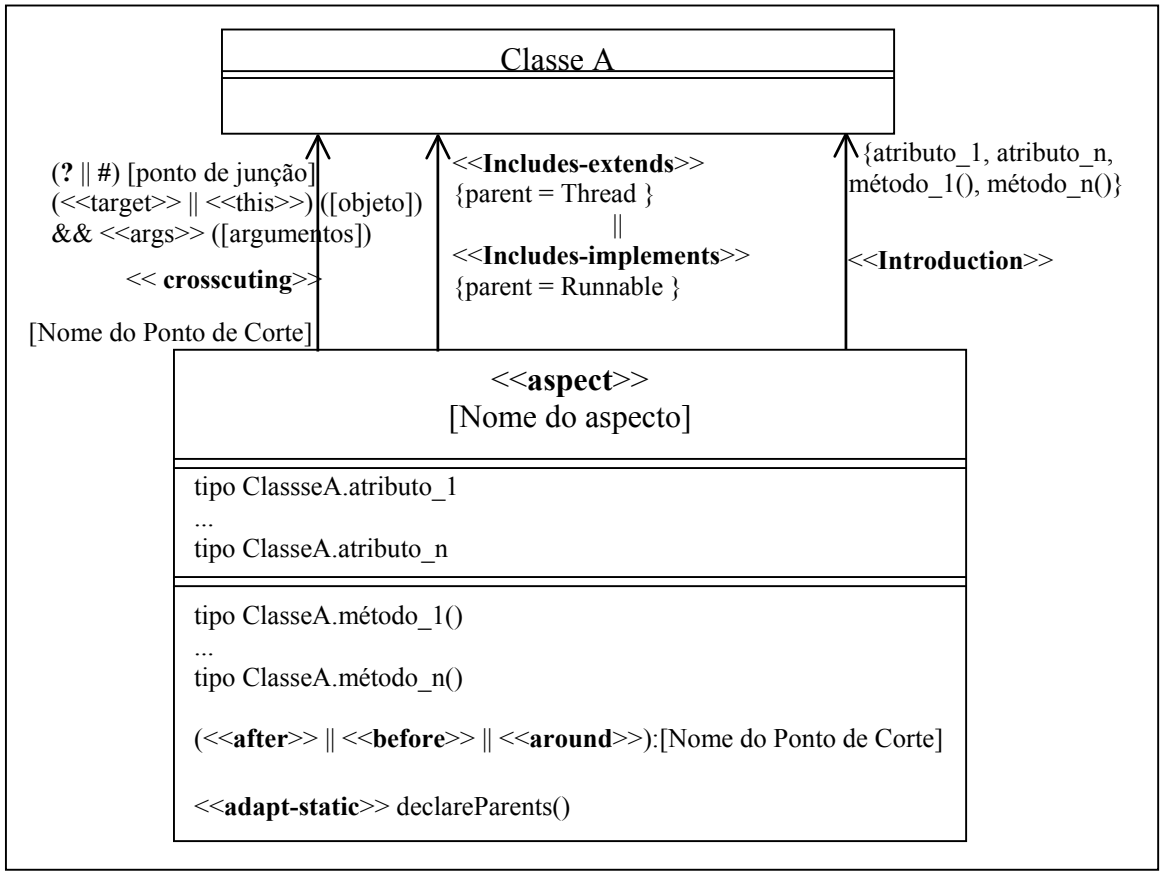


Figura 4.7 - Modelagem de um Aspecto para o Interesse de Programação Paralela.

c) Descrição - para modelar o Interesse de Programação Paralela**:**

1. Para cada classe que contenha trechos marcados e numerados como sendo do interesse de Programação Paralela.

1.1. Adicionar ao diagrama de classes de Implementação um *Aspect-class* com o estereotipo <<aspect>>, seguido do Nome do aspecto.

1.1.1. Adicionar ao *Aspect-class* os atributos do interesse, utilizando o modelo:
[Tipo] [Nome da classe].[Atributo];
Sendo que:
[Tipo]: indica o tipo do atributo que foi retirado da Classe A.
[Nome da classe]: indica o nome da Classe A.
[Atributo]: indica-se o nome do atributo que foi retirado da Classe A.

1.1.2. Adicionar ao *Aspect-class* os métodos que pertencem totalmente ao interesse, utilizando o modelo:
[Tipo] [Nome da classe].[Método];
Sendo que:
[Tipo]: indica o tipo do retorno do método retirado da Classe A.
[Nome da classe]: indica o nome da Classe A.
[Método]: indica o nome do método que retirado da Classe A.

1.1.3. Caso existam métodos que dependam de outros interesses:

1.1.3.1 Se o trecho de código marcado como interesse não estiver dentro de um bloco de *loop*, então:

1.1.3.1.1. Criar um "*aspect-method*" e adicioná-lo ao *Aspect-class*, utilizando o modelo:
(<<after>> || <<before>> || <<around>>):[Nome do Ponto de Corte]
Sendo que (<<after>> || <<before>> || <<around>>): deve-se optar pela opção dependendo da ordem em que deseja ser executar o corpo da sugestão (*advice*).
[Nome do Ponto de Corte]: Deve-se indicar o nome do ponto de corte.

1.1.3.2. Se não:

1.1.3.2.1. Criar um método que contenha esse trecho de código e inseri-lo como no passo 1.1.2, acima.

1.1.4. Criar um "*aspect-method*" que indica a declaração de parentesco para a declaração de herança da classe `Thread` ou para a implementação da interface `Runnable`, utilizando o modelo:

<<adapt-static>> `declareParents( )`

1.2. Criar um relacionamento de associação, indicando o tipo de parentesco, que pode ser *implements* ou *extends*, utilizando o modelo:

<<Includes-extends>> {parent = Thread } ||

<<Includes-implements>>{parent = Runnable }

1.3. Criar um relacionamento de associação com origem no *Aspect-class* e destino na Classe A, utilizando o estereótipo <<Introduction>>, se existirem métodos e/ou atributos relacionados ao interesse.

1.3.1. Criar uma declaração do tipo: {atributo_1, atributo_n, método_1(), método_n()}, como papel da Classe A.

1.4. Criar um relacionamento de associação com origem no *Aspect-class* e destino na Classe A, utilizando o estereótipo <<crosscuting>> .

1.4.1. Criar uma declaração do tipo: [nome do ponto de corte] como papel do *Aspect-class*.

1.4.2. Criar uma declaração do tipo: (**?** || #) [ponto de junção] (<<target>> || <<this>>) ([objeto]) && <<args>> ([argumentos]) como papel da Classe A.

Sendo que [nome do ponto de junção]: indica a instanciação do objeto ou a assinatura do método que contém os parâmetros utilizados nos comandos de decisão.

(<<target>> || <<this>>): deve-se optar dependendo do contexto.

([objeto]): indica o nome do objeto que se deseja capturar.

&& : conjunção aditiva.

([argumentos]): indica os argumentos, sem indicar seus tipos.

➲ **Diretrizes para Modelar o Interesse de Persistência em Memória Intermediária** (*Buffering*)

a) Resumo:

Adicionar ao diagrama de classes um aspecto especifico para cada classe que faça a Persistência em Memória Intermediária.
Adicionar nesse Aspecto utilizando o conceito de introduções os atributos e métodos da classe que forem específicos do interesse.
Caso o método seja parcialmente do interesse, criar um ponto de corte que capture como ponto de junção esse método.

b) Exemplo de Diagrama de Classes de Implementação, Figura 4.8: do interesse de Persistência em Memória Temporária, gerado a partir das diretrizes de modelagem, apresentadas a seguir, e com base no código fonte legado de um sistema OO.

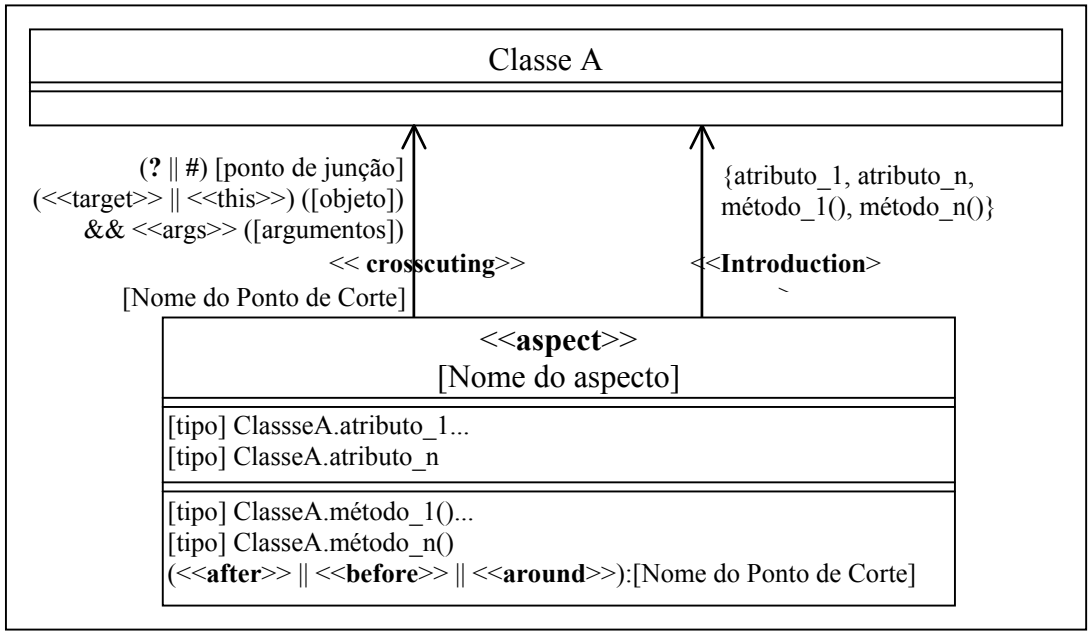


Figura 4.8 - Modelagem de um Aspecto para o Interesse de Persistência em Memória Temporária.

c) Descrição - para modelar o Interesse de Persistência em Memória Temporária**:**

1. Para cada classe que contenha trechos marcados e numerados como sendo do interesse de Persistência em Memória Temporária (*Buffering*).

1.1. Adicionar ao diagrama de classes de Implementação um *Aspect-class* com o estereotipo <<aspect>>, seguido do Nome do aspecto.

1.1.1. Adicionar ao *Aspect-class* os atributos do interesse, utilizando o modelo:

[Tipo] [Nome da classe].[Atributo];

Sendo que:

[Tipo]: indica o tipo do atributo que foi retirado da Classe A.

[Nome da classe]: indica o nome da Classe A.

[Atributo]: indica-se o nome do atributo que foi retirado da Classe A.

1.1.2. Adicionar ao *Aspect-class* os métodos que pertencem totalmente ao interesse, utilizando o modelo:

[Tipo] [Nome da classe].[Método];

Sendo que:

[Tipo]: indica o tipo do retorno do método retirado da Classe A.

[Nome da classe]: indica o nome da Classe A.

[Método]: indica o nome do método que retirado da Classe A.

1.1.3. Caso existam métodos que dependam de outros interesses:

1.1.3.1 Se o trecho de código marcado como interesse não estiver dentro de um bloco de *loop*, então:

1.1.3.1.1. Criar um "*aspect-method*" e adicioná-lo ao *Aspect-class*, utilizando o modelo:

(<<after>> || <<before>> || <<around>>):[Nome do Ponto de Corte]

Sendo que:

(<<after>> || <<before>> || <<around>>): é escolhido dependendo da ordem em que deseja ser executar o corpo da sugestão (*advice*).

[Nome do Ponto de Corte]: Deve-se indicar o nome do ponto de corte.

1.1.3.2. Se não:

1.1.3.2.1. Criar um método que contenha esse trecho de código e inseri-lo como no passo 1.1.2, acima.

1.2. Criar um relacionamento de associação com origem no *Aspect-class* e destino na Classe A, utilizando o estereótipo <<Introduction>>, se existirem métodos e/ou atributos relacionados ao interesse.

1.2.1. Criar uma declaração do tipo: {atributo_1, atributo_n, método_1(), método_n()}, como papel da Classe A.

1.3. Criar um relacionamento de associação com origem no *Aspect-class* e destino na Classe A, utilizando o estereótipo <<crosscuting>> .

1.3.1. Criar uma declaração do tipo: [nome do ponto de corte] como papel do *Aspect-class*.

1.3.2. Criar uma declaração do tipo: (**?** || #) [ponto de junção] (<<target>> || <<this>>) ([objeto]) && <<args>> ([argumentos]) como papel da Classe A.

Sendo que:

[nome do ponto de junção]: indica a instanciação do objeto ou a assinatura do método que contém os parâmetros utilizados nos comandos de decisão.

(<<target>> || <<this>>): deve-se optar dependendo do contexto.

([objeto]): indica o nome do objeto que se deseja capturar.

&& : conjunção aditiva.

([argumentos]): indica os argumentos, sem indicar seus tipos.

4.4.3. Passo II.3: Implementar o Interesse

Trata da implementação do sistema orientado a aspectos utilizando o diagrama de classes gerado no passo II.2 e as diretrizes criadas para esse fim. A entrada desse passo é o código fonte, com o interesse marcado, juntamente com o diagrama de classes com aspectos e o artefato de saída é o sistema orientado a aspectos implementado na linguagem AspectJ.

Como comentado anteriormente, esta etapa é evolutiva, sendo necessário que o engenheiro de software retorne aos passos “Marcar Interesse”, “Modelar Interesse”, “Implementar Interesse” e “Aplicar Testes de Regressão”, até que todos os interesses da lista de indícios sejam pesquisados ou até que o engenheiro de software decida parar o processo, por considerar suficiente a implementação de apenas alguns aspectos.

Para cada tipo de interesse encontrado há um conjunto de diretrizes para realizar a implementação em AspectJ que é apresentado com três partes: a) um resumo, contendo uma explicação sucinta de como a implementação será realizada, b) um exemplo de código fonte do interesse que será implementado em aspecto e c) descrição, passo a passo, para que a implementação de cada aspecto seja conduzida.

➲ **Diretrizes para Implementar o Aspecto de Rastreamento**

a) Resumo:

Para cada Aspecto criado no diagrama de Classes de Implementação para o interesse, implementar um na linguagem AspectJ.
Adicionar na sugestão (*Advice*) o código referente ao Rastreamento para o ponto de corte correspondente

b) Exemplo de Código Fonte da Implementação do aspecto para o interesse de Rastreamento, Figura 4.9, gerado a partir das diretrizes de implementação, apresentadas a seguir.

```
public aspect AspectRastreamento {

      pointcut [nome do ponto de corte] ([objeto]):
      [execution || call]([ponto de junção])
      && [target || this]([objeto])
      && args([argumentos]);

      [after || before || around] ([argumentos]): [nome do ponto
      de corte] ([argumentos]) { [corpo da sugestão] }
}
```

Figura 4.9 - Exemplo de um Aspecto para o Interesse de Rastreamento.

c) Descrição - para implementar o Interesse de Rastreamento**:**

Para cada *Aspect-class* criado no diagrama de classes de Implementação para o interesse de Rastreamento.

1. Criar um aspecto em AspectJ, utilizando o modelo:

`public aspect [nome do aspecto]`

Sendo que `[nome do aspecto]`: é o nome indicado no *Aspect-class*.

1.1. Criar um ponto de corte de acordo com as informações inseridas no diagrama de classes de Implementação, utilizando o modelo:

`pointcut [nome do ponto de corte] ([objeto]): [execution || call]([ponto de junção]) && [target || this]([objeto]) && args([argumentos]);`

Sendo que:

`[nome do ponto de corte]`: refere-se ao nome do ponto de corte, indicado no papel do *Aspect-class* no relacionamento de entrecorte do diagrama de classes de Implementação.

`([objeto])`: refere-se ao objeto que se deseja capturar.

`[execution || call]`: refere-se ao (? || #) indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

`[target || this]`: refere-se ao indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

`([argumentos])`: refere-se aos argumentos indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

1.2. Criar uma sugestão (*advice*), utilizando o modelo:

```
[after || before || around] ([argumentos]): [nome do ponto de corte] ([argumentos]) {[corpo da sugestão]}
```

Sendo que:

`[after || before || around]`: refere-se ao indicado no *aspect-method* do *Aspect-class* do diagrama de classes de Implementação.

`[argumentos]`: são os mesmos referenciados no passo 1.1.

`[nome do ponto de corte]`: é o mesmo referenciado no passo 1.1.

`[corpo da sugestão]`: insere-se o trecho de código fonte referente ao interesse marcado no método da Classe A.

2. Utilizar comentários para marcar na Classe A todo o trecho de código que foi implementado nesse aspecto.

➲ **Diretrizes para Implementar os Aspectos de Tratamento de Erros.**

a) Resumo:

Para cada Aspecto criado no diagrama de Classes de Implementação para o interesse, implementar um na linguagem AspectJ.
Adicionar na sugestão (*Advice*) o código referente ao Tratamento de Erros para o ponto de corte correspondente.

b) Exemplo de Código Fonte da Implementação do aspecto para o interesse de Tratamento de Erros, Figura 4.10, gerado a partir das diretrizes de implementação, apresentadas a seguir.

```
public aspect AspectTratamentoErros {

      pointcut [nome do ponto de corte] ([objeto]):
      [execution || call]([ponto de junção])
      && [target || this]([objeto])
      && args([argumentos]);
      after([argumentos]): [nome do ponto de corte] ([argumentos])
      { [corpo da sugestão] }
}
```

Figura 4.10 - Exemplo de um Aspecto para o Interesse de Tratamento de Erros.

c) Descrição - para implementar o Interesse de interesse de Tratamento de Erros**:**

Para cada *Aspect-class* criado no diagrama de classes de Implementação para o interesse de Tratamento de Erros.

1. Criar um aspecto em AspectJ, utilizando o modelo:

`public aspect [nome do aspecto]`

Sendo que :

`nome do aspecto]`: é o nome indicado no *Aspect-class*.

1.1. Criar um ponto de corte de acordo com as informações inseridas no diagrama de classes de Implementação, utilizando o modelo:

`pointcut [nome do ponto de corte] ([objeto]): [execution || call]([ponto de junção]) && [target || this]([objeto]) && args([argumentos]);`

Sendo que:

`[nome do ponto de corte]`: refere-se ao nome do ponto de corte, indicado no papel do *Aspect-class* no relacionamento de entrecorte do diagrama de classes de Implementação.

`([objeto])`: refere-se ao objeto que se deseja capturar.

`[execution || call]`: refere-se ao (? || #) indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

`[ponto de junção]`: refere-se ao(s) ponto(s) de junção(ões) indicados no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

`[target || this]`: refere-se ao indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

`([argumentos])`: refere-se aos argumentos indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

1.2. Criar uma sugestão (*advice*), utilizando o modelo:

```
after ([argumentos]): [nome do ponto de corte] ([argumentos])
{[corpo da sugestão]}
```

`[argumentos]`: são os mesmos referenciados no passo 1.1.

`[nome do ponto de corte]`: é o mesmo referenciado no passo 1.1.

`[corpo da sugestão]`: insere-se o trecho de código fonte referente ao interesse marcado no método da Classe A.

2. Utilizar comentários para marcar na Classe A todo o trecho de código que foi implementado neste aspecto.

➲ **Diretrizes para Implementar os Aspectos de Tratamento de Exceção.**

a) Resumo:

Para cada Aspecto criado no diagrama de Classes de Implementação para o interesse, implementar um na linguagem AspectJ.
Adicionar ao Aspecto a declaração do tipo de exceção que será tratada.
Adicionar na sugestão (*Advice*) o código referente ao Tratamento de Exceção para o ponto de corte correspondente.

b) Exemplo de Código Fonte da Implementação do aspecto para o interesse de Tratamento de Exceção, Figura 4.11, gerado a partir das diretrizes de implementação, apresentadas a seguir.

```
public aspect AspectTratamentoExcecaoTipo {

      pointcut [nome do ponto de corte]:
      [execution || call]([ponto de junção I]) ...
      [execution || call]([ponto de junção n]); // Caso II
      declare soft: [TipoI da Exceção] : [nome do ponto de corte];
      after throwing([Tipo da Exceção] [variável]): [nome do ponto de
      corte] {
            try{ throw new [Tipo da Exceção] }
            catch([Tipo da Exceção] [variável 2])
            {
                  [Corpo do bloco catch para [ponto de junção I]]
                  if(thisJoinPoint.getStaticPart().equals([ponto     de
                  junção n]) { [conteúdo do bloco catch] }
             }
      }
}
```

Figura 4.11 - Exemplo de um Aspecto para o Interesse de Tratamento de Exceção.

c) Descrição - para implementar o Interesse de Tratamento de Exceção**:**

Para cada *Aspect-class* criado no diagrama de classes de Implementação para o interesse de Tratamento de Exceção.

Para o Caso I (Existe Aspecto para o Tipo).

1. Criar um aspecto em AspectJ, utilizando o modelo:

`public aspect [nome do aspecto]`

Sendo que: `[nome do aspecto]`: é o nome indicado no *Aspect-class*.

1.1. Criar um ponto de corte de acordo com as informações inseridas no diagrama de classes de Implementação, utilizando o modelo:

`pointcut [nome do ponto de corte]: execution || call([ponto de junção]);`

Sendo que:

`[nome do ponto de corte]`: refere-se ao nome do ponto de corte, indicado no papel do *Aspect-class* no relacionamento de entrecorte do diagrama de classes de Implementação.

`[execution || call]`: refere-se ao (? || #) indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

`[ponto de junção]`: refere-se ao ponto de junção indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

1.2. Criar a declaração "*declare soft*", utilizando o modelo:

```
declare soft: [Tipo da Exceção] : [nome do ponto de corte];
```

Sendo que `[Tipo da Exceção]`: refere-se ao indicado no papel do *Aspect-class* no relacionamento de entrecorte do diagrama de classes de Implementação.

`[nome do ponto de corte]`: refere-se ao indicado no papel do *Aspect-class* no relacionamento de entrecorte do diagrama de classes de Implementação.

1.3. Criar uma sugestão (*advice*), utilizando o modelo:

```
after throwing([Tipo da Exceção] [variável]): [nome do ponto de corte] { [corpo do aspect-method] }
```

Sendo que:

`[Tipo da Exceção]`: é o mesmo referenciado no passo 1.2.

`[variável]`: refere-se a variável indicada no *aspect-method* do *Aspect-class*.

`[nome do ponto de corte]`: refere-se ao nome indicado no *aspect-method* do *Aspect-class*.

`[corpo do aspect-method]`: insere-se o código fonte para tratar a exceção, utilizando o modelo:

```
try{ throw new [Tipo da Exceção] } catch([Tipo da Exceção] [variável 2]) {[Corpo do bloco catch]}
```

Sendo que:

`[Tipo da Exceção]`: é o mesmo referenciado no passo 1.2.

`[variável 2]`: refere-se a uma variável que deverá ser criada para tratar a exceção no bloco `catch`.

`[Corpo do bloco catch]`: insere-se o conteúdo do bloco `catch` que está marcado em um trecho de código fonte da Classe A correspondente ao trecho de código que deu origem a este aspecto.

2. Utilizar comentários para marcar na classe todo o trecho de código que foi implementado.

Para o Caso II (Não Existe Aspecto para o Tipo).

1. Adicionar ao ponto de corte já criado no aspecto, o pontos de junção, utilizando o modelo:

```
pointcut [ponto de corte já criado]: [execution || call]([ponto de junção I]) || ... || [execution || call] ([ponto de junção n]);
```

Sendo que:

`[execution || call]`: refere-se ao (? || #) indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

`[ponto de junção n]`: refere-se ao ponto de junção indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

2. Se o conteúdo do bloco `catch` da Classe A, contido no trecho de código marcado como interesse, já estiver contido no bloco catch do aspecto, então:

2.1. Utilizar comentários para marcar na Classe A todo o trecho de código que foi implementado.

3. Senão:

3.1. Para cada bloco `catch` da Classe A com o conteúdo diferente do bloco `catch` existente no Aspecto:

3.1.1. Adicionar um comando de decisão dentro do bloco `catch` do aspecto, que faça a comparação em tempo de execução do ponto de junção que foi alcançado, utilizando o modelo:

```
if(thisJoinPoint.getStaticPart().equals([ponto de junção])
{[conteúdo do bloco catch]}
```

Sendo que:

`[ponto de junção]`: refere-se a cada ponto de junção que tenha o conteúdo do bloco catch diferente do já existente no aspecto.

`[conteúdo do bloco catch]`: insere-se o código fonte referente ao tratamento da exceção para o `[ponto de junção]`.

3.1.2. Utilizar comentários para marcar na classe todo o trecho de código que foi implementado.

➲ **Diretrizes para Implementar os Aspectos de Persistência em Banco de Dados Relacional.**

a) Resumo:

Para cada Aspecto criado no diagrama de Classes de Implementação para o interesse, implementar um na linguagem AspectJ.
Adicionar ao Aspecto utilizando o conceito de introduções os atributos e métodos correspondentes, relacionados no diagrama.
Adicionar na sugestão (*Advice*) o código referente ao interesse de Persistência em Banco de Dados Relacional para o ponto de corte correspondente.

b) Exemplo de Código Fonte da Implementação do aspecto para o interesse de Persistência em Banco de Dados Relacional, Figura 4.12, gerado a partir das diretrizes de implementação, apresentadas a seguir.

```
public aspect AspectPersistenceClasseA {

      [tipo] [Classe A].[nome do atributo];
      ...

      [tipo] [Classe A].[nome do método] {[corpo do método]}
      ...

      pointcut [nome do ponto de corte] ([objeto]):
      [execution || call]([ponto de junção])
      && [target || this]([objeto])
      && args([argumentos]);

      [after || before || around] ([argumentos]): [nome do ponto
      de corte] ([argumentos]) { [corpo da sugestão] }
}
```

Figura 4.12 - Exemplo de um Aspecto para o Interesse de Persistência em Banco de Dados Relacional.

c) Descrição - para implementar o Interesse de Persistência em Banco de Dados Relacional**:**

Para cada *Aspect-class* criado no diagrama de classes de Implementação para o interesse de Persistência em Banco de Dados Relacional.

1. Criar um aspecto em AspectJ, utilizando o modelo:

`public aspect [nome do aspecto]`

Sendo que:

`[nome do aspecto]`: é o nome indicado no *Aspect-class*.

1.1. Inserir ao aspecto os atributos, se existirem, utilizando o modelo:

`[tipo] [nome da classe].[nome do atributo];`

Sendo que:

`[tipo]`: refere-se ao tipo do atributo indicado no *Aspect-classes* do diagrama de classes de Implementação.

`[nome da classe]`: refere-se ao nome da Classe A.

`[nome do atributo]`: refere-se ao nome do atributo indicado no *Aspect-classes* do diagrama de classes de Implementação.

1.1.1. Utilizar comentários para marcar na classe todo o trecho de código que foi implementado.

1.2. Inserir ao aspecto os métodos que pertencem ao interesse, se existirem, utilizando o modelo:

```
[tipo] [nome da classe].[nome do método] {[corpo do método]}
```

Sendo que:

`[tipo]`: refere-se ao tipo do método indicado no *Aspect-classes* do diagrama de classes de Implementação.

`[nome do método]`: refere-se ao nome do método indicado no *Aspect-classes* do diagrama de classes de Implementação.

`[corpo do método]`: Insere-se o corpo do método da Classe A referente a esse método que esta sendo implementado no aspecto.

1.2.1. Utilizar comentários para marcar na classe todo o trecho de código que foi implementado.

1.3. Criar um ponto de corte, se existir, de acordo com as informações inseridas no diagrama de classes de Implementação, utilizando o modelo:

```
pointcut [nome do ponto de corte] ([objeto]): [execution || call]([ponto de junção]) && [target || this]([objeto]) && args([argumentos]);
```

Sendo que `[nome do ponto de corte]`: refere-se ao nome do ponto de corte, indicado no papel do *Aspect-class* no relacionamento de entrecorte do diagrama de classes de Implementação.

`([objeto])`: refere-se ao objeto que se deseja capturar.

`[execution || call]`: refere-se ao (? || #) indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

`[target || this]`: refere-se ao indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

`([argumentos])`: refere-se aos argumentos indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

1.4. Criar uma sugestão (*advice*), utilizando o modelo:

```
[after || before || around] ([argumentos]): [nome do ponto de corte] ([argumentos]) {[corpo da sugestão]}
```

Sendo que: `[after || before || around]`: refere-se ao indicado no *aspect-method* do *Aspect-class* do diagrama de classes de Implementação.

`[argumentos]`: são os mesmos referenciados no passo 1.3.

`[nome do ponto de corte]`: é o mesmo referenciado no passo 1.3.

`[corpo da sugestão]`: insere-se o trecho de código fonte referente ao interesse marcado no método da Classe A.

1.4.1. Utilizar comentários para marcar na classe todo o trecho de código que foi implementado.

➲ **Diretrizes para Implementar os Aspectos de Programação Paralela.**

a) Resumo:

Para cada Aspecto criado no diagrama de Classes de Implementação para o interesse, implementar um na linguagem AspectJ.
Adicionar ao Aspecto as declarações de parentesco conforme descritas no diagrama.
Adicionar ao Aspecto utilizando o conceito de introduções os atributos e métodos correspondentes, relacionados no diagrama.
Adicionar na sugestão (*Advice*) o código referente ao interesse de Programação Paralela para o ponto de corte correspondente.

b) Exemplo de Código Fonte da Implementação do aspecto para o interesse de Programação Paralela, Figura 4.13, gerado a partir das diretrizes de implementação, apresentadas a seguir.

```
public aspect AspectProgramacaoParalela {
      declare parents: [nome da classe] extends Thread; ou
      declare parents: [nome da classe] implements Runnable;

      [tipo] [Classe A].[nome do atributo];
      ...

      [tipo] [Classe A].[nome do método] {[corpo do método]}
      ...

      pointcut [nome do ponto de corte] ([objeto]):
      [execution || call]([ponto de junção])
      && [target || this]([objeto])
      && args([argumentos]);

      [after || before || around] ([argumentos]): [nome do ponto
      de corte] ([argumentos]) { [corpo da sugestão] }
}
```

Figura 4.13 - Exemplo de um Aspecto para o Interesse de Programação Paralela.

c) Descrição - para implementar o Interesse de Programação Paralela**:**

Para cada *Aspect-class* criado no diagrama de classes de Implementação para o interesse de Programação Paralela.

1. Criar um aspecto em AspectJ, utilizando o modelo:

```
public aspect [nome do aspecto]
```

Sendo que `[nome do aspecto]`: é o nome indicado no *Aspect-class*.

1.1. Inserir ao aspecto a declaração de parentesco da Classe A, referindo-se ao papel da Classe A, utilizando o modelo:

```
declare parents: [nome da classe] extends Thread; ou
declare parents: [nome da classe] implements Runnable;
```

Sendo que:

`[nome da classe]`: refere-se ao nome da Classe A.

1.2. Inserir ao aspecto os atributos, se existirem, utilizando o modelo:

```
[tipo] [nome da classe].[nome do atributo];
```

Sendo que:

`[tipo]`: refere-se ao tipo do atributo indicado no *Aspect-classes* do diagrama de classes de Implementação.

`[nome da classe]`: refere-se ao nome da Classe A.

`[nome do atributo]`: refere-se ao nome do atributo indicado no *Aspect-classes* do diagrama de classes de Implementação.

1.2.1. Utilizar comentários para marcar na classe todo o trecho de código que foi implementado.

1.3. Inserir ao aspecto os métodos que pertencem ao interesse, se existirem, utilizando o modelo:

```
[tipo] [nome da classe].[nome do método] {[corpo do método]}
```

Sendo que:

`[tipo]`: refere-se ao tipo do método indicado no *Aspect-classes* do diagrama de classes de Implementação.

`[nome do método]`: refere-se ao nome do método indicado no *Aspect-classes* do diagrama de classes de Implementação.

`[corpo do método]`: Insere-se o corpo do método da Classe A referente a esse método que esta sendo implementado no aspecto.

1.3.1. Utilizar comentários para marcar na classe todo o trecho de código que foi implementado.

1.4. Criar um ponto de corte, se existir, de acordo com as informações inseridas no diagrama de classes de Implementação, utilizando o modelo:

```
pointcut [nome do ponto de corte] ([objeto]): [execution || call]([ponto de junção]) && [target || this]([objeto]) && args([argumentos]);
```

Sendo que:

`[nome do ponto de corte]`: refere-se ao nome do ponto de corte, indicado no papel do *Aspect-class* no relacionamento de entrecorte do diagrama de classes de Implementação.

`([objeto])`: refere-se ao objeto que se deseja capturar.

`[execution || call]`: refere-se ao (? || #) indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

`[target || this]`: refere-se ao indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

`([argumentos])`: refere-se aos argumentos indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

1.5. Criar uma sugestão (*advice*), utilizando o modelo:

```
[after || before || around] ([argumentos]): [nome do ponto de corte] ([argumentos]) {[corpo da sugestão]}
```

Sendo que:

`[after || before || around]`: refere-se ao indicado no *aspect-method* do *Aspect-class* do diagrama de classes de Implementação.

`[argumentos]`: são os mesmos referenciados no passo 1.4.

`[nome do ponto de corte]`: é o mesmo referenciado no passo 1.4.

`[corpo da sugestão]`: insere-se o trecho de código fonte referente ao interesse marcado no método da Classe A.

1.5.1. Utilizar comentários para marcar na classe todo o trecho de código que foi implementado.

➲ **Diretrizes para Implementar os Aspectos de Persistência em Memória Intermediária (*Buffering*).**

a) Resumo:

Para cada Aspecto criado no diagrama de Classes de Implementação para o interesse, implementar um na linguagem AspectJ.
Adicionar ao Aspecto utilizando o conceito de introduções os atributos e métodos correspondentes, relacionados no diagrama.
Adicionar na sugestão (*Advice*) o código referente ao interesse de Persistência em Memória Temporária para o ponto de corte correspondente.

b) Exemplo de Código Fonte da Implementação do aspecto para o interesse de Persistência em Memória Temporária, Figura 4.14, gerado a partir das diretrizes de implementação, apresentadas a seguir.

```
public aspect AspectBuffering {

      [tipo] [Classe A].[nome do atributo];
      ...

      [tipo] [Classe A].[nome do método] {[corpo do método]}
      ...

      pointcut [nome do ponto de corte] ([objeto]):
      [execution || call]([ponto de junção])
      && [target || this]([objeto])
      && args([argumentos]);


      [after || before || around] ([argumentos]): [nome do ponto
      de corte] ([argumentos]) { [corpo da sugestão] }
}
```

Figura 4.14 - Exemplo de um Aspecto para o Interesse de Persistência em Memória Temporária (*Buffering*).

c) Descrição - para implementar o Interesse de Persistência em Memória Temporária**:**

Para cada *Aspect-class* criado no diagrama de classes de Implementação para o interesse de Persistência em Memória Temporária (*Buffering*).

1. Criar um aspecto em AspectJ, utilizando o modelo:

`public aspect [nome do aspecto]`

Sendo que `[nome do aspecto]`: é o nome indicado no *Aspect-class*.

1.1. Inserir ao aspecto os atributos, se existirem, utilizando o modelo:

```
[tipo] [nome da classe].[nome do atributo];
```

Sendo que:

`[tipo]`: refere-se ao tipo do atributo indicado no *Aspect-classes* do diagrama de classes de Implementação.

`[nome da classe]`: refere-se ao nome da Classe A.

`[nome do atributo]`: refere-se ao nome do atributo indicado no *Aspect-classes* do diagrama de classes de Implementação.

1.1.1. Utilizar comentários para marcar na classe todo o trecho de código que foi implementado.

1.2. Inserir ao aspecto os métodos que pertencem ao interesse, se existirem, utilizando o modelo:

```
[tipo] [nome da classe].[nome do método] {[corpo do método]}
```

Sendo que:

`[tipo]`: refere-se ao tipo do método indicado no *Aspect-classes* do diagrama de classes de Implementação.

`[nome do método]`: refere-se ao nome do método indicado no *Aspect-classes* do diagrama de classes de Implementação.

`[corpo do método]`: Insere-se o corpo do método da Classe A referente a esse método que esta sendo implementado no aspecto.

1.2.1. Utilizar comentários para marcar na classe todo o trecho de código que foi implementado.

1.3. Criar um ponto de corte, se existir, de acordo com as informações inseridas no diagrama de classes de Implementação, utilizando o modelo:

```
pointcut [nome do ponto de corte] ([objeto]): [execution || call]([ponto de junção]) && [target || this]([objeto]) && args([argumentos]);
```

Sendo que:

`[nome do ponto de corte]`: refere-se ao nome do ponto de corte, indicado no papel do *Aspect-class* no relacionamento de entrecorte do diagrama de classes de Implementação.

`([objeto])`: refere-se ao objeto que se deseja capturar.

`[execution || call]`: refere-se ao (? || #) indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

`[target || this]`: refere-se ao indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

`([argumentos])`: refere-se aos argumentos indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de Implementação.

1.4. Criar uma sugestão (*advice*), utilizando o modelo:

```
[after || before || around] ([argumentos]): [nome do ponto de corte] ([argumentos]) {[corpo da sugestão]}
```

Sendo que:

`[after || before || around]`: refere-se ao indicado no *aspect-method* do *Aspect-class* do diagrama de classes de Implementação.

`[argumentos]`: são os mesmos referenciados no passo 1.3.

`[nome do ponto de corte]`: é o mesmo referenciado no passo 1.3.

`[corpo da sugestão]`: insere-se o trecho de código fonte referente ao interesse marcado no método da Classe A.

1.4.1. Utilizar comentários para marcar na classe todo o trecho de código que foi implementado.

4.4.4. Passo II.4: Aplicar Testes de Regressão

O principal objetivo deste passo é verificar que a funcionalidade do sistema que foi gerado com aspectos, após cada iteração, foi preservada em relação ao sistema original.
A partir dos casos de teste funcionais gerados na primeira etapa, o engenheiro de software executa o sistema utilizando os dados de entrada para verificar que os resultados obtidos são os mesmo documentados nos casos de teste. Após a verificação de cada caso de teste, infere-se se a funcionalidade total do sistema implementado com aspectos continua ou não como a original.

4.5. Considerações Finais

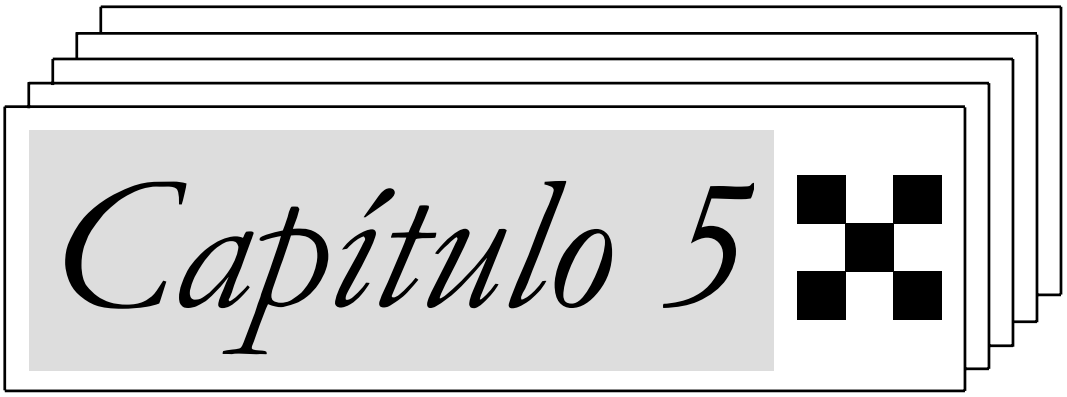
Este Capítulo apresentou a abordagem de migração de sistemas orientados a objetos para sistemas orientados a aspectos, com o objetivo de torná-los mais manuteníveis, separando os interesses mais comuns que estavam espalhados e entrelaçados no código fonte do sistema.

A Abordagem *Aspecting* tem duas etapas distintas: Entender a Funcionalidade do Sistema e Tratar o Interesse. Cada etapa foi desenvolvida cuidando dos detalhes que devem ser seguidos pelo engenheiro de software para que a migração dos sistemas Orientados a Objetos seja realizada de forma segura e obtenha-se um produto confiável. Essa abordagem foi criada a partir de estudos de caso realizados, expostos no Capítulo 3.

O ponto fraco da *Aspecting* é que novos indícios podem surgir a qualquer momento, portanto todas as possibilidades de migrar o sistema completamente para a orientação a aspectos, não foram aqui esgotadas. Periodicamente, pode-se reavaliar a Lista de Indícios e completá-la a fim de atender outros interesses.

Embora nesta dissertação não se tenha tratado do desenvolvimento de sistemas orientados a aspectos a partir do levantamento de requisitos, a *Aspecting* também pode ser utilizada para essa finalidade, elaborando-se os diagramas de casos de uso com as suas descrições, diagramas de classes, incluindo-se os aspectos nesse diagrama de classes para posterior implementação.

O Capítulo 5 apresenta os resultados obtidos na aplicação da abordagem *Aspecting* em um sistema de Controle de Bebidas, desenvolvido com o paradigma orientado a objetos e implementado na linguagem Java.



Capítulo 5

Estudo de Caso

5.1.Considerações Iniciais

A aplicação da abordagem Aspecting em um estudo de caso diferente dos que foram utilizados para a sua elaboração é o foco neste capítulo. Assim, um sistema desenvolvido na linguagem Java para atender os pedidos de uma distribuidora de bebidas foi utilizado.

Esse sistema, assim como os outros, foram implementados por diferentes pessoas e não possuíam documentação além do código fonte. O sistema desse estudo de caso tem 1433 linhas de código, de complexidade simples, utiliza o banco de dados em *MS-Access*, cuja conexão é feita via JDBC utilizando ODBC com o *driver* que já vem embutido no próprio *MS-Access*. É composto por 19 arquivos com extensão `.java` que implementam 69 classes, sendo que dessas 6 contêm as regras de negócio e 63 tratam da interface, que é feita utilizando o pacote `javax.swing` disponível no próprio Java. Neste estudo somente serão implementados em aspectos os interesses não funcionais que estiverem espalhados e entrelaçados nas classes que contém as regras de negócio do sistema.

A fim de facilitar a aplicação da abordagem aspecting, a Tabela 5.1 apresenta resumindo seus passos.

Tabela 5.1 - Etapas e Passos da Abordagem *Aspecting*.

Etapas	Passos	Descrição
I. Entender a Funcionalidade do Sistema **Descrição:** Extrair as informações sobre a funcionalidade do sistema.	**I.1**. Gerar Diagramas de Casos de Uso. (caso não existam)	Executar o sistema para criar casos de uso correspondentes às funcionalidades nele existente. Utilizar as diretrizes criadas para esse fim.
	I.2. Gerar Casos de Teste (caso não existam)	Exercitar os Casos de Uso considerando os dados de entradas e saídas como casos de teste. **Opção**: Utilizar uma Ferramenta que gere os Casos de Teste, por exemplo, Jtest (2004)
	I.3. Gerar Diagrama de Classes de Implementação. (caso não exista)	Criar um diagrama de classes a partir do código fonte.Utilizar as diretrizes criadas para esse fim. **Opção**: Utilizar uma Ferramenta Case, por exemplo, Omondo (2004).
II. Tratar Interesses **Descrição:** Esta etapa é evolutiva sendo seus passos repetidos até que todos os interesses da Lista de Indícios sejam pesquisados ou até que o engenheiro de software decida finalizar o processo.	**II.1**. Marcar o Interesse. (caso exista)	Para um dos interesses da lista de indícios: Para cada classe implementada no código fonte, se existir tal indício Marcar o trecho do código fonte, adicionando comentários no final de cada linha, indicando o nome do interesse, seguido de um número seqüencial que deverá indicar a ordem em que esse trecho aparece na classe.
	II.2. Modelar o Interesse.	Adicionar ao diagrama de classe de implementação o aspecto. Utilizar as diretrizes específicas de cada interesse criadas para esse fim.
	II.3. Implementar o Interesse.	Implementar o aspecto que foi adicionado ao diagrama de classes no passo anterior. Utilizar as diretrizes específicas de cada interesse criadas para esse fim.Retornar ao passo II.1.
	II.4. Aplicar Testes de	Utilizar os Casos de Teste gerados no passo I.2 para verificar se a funcionalidade do

A organização deste Capítulo é da seguinte forma: a Seção 5.2 apresenta a aplicação da Etapa I - Entender a Funcionalidade, a Seção 5.3 foca a aplicação da Etapa II - Tratar Interesses; a Seção 5.4 discute a Avaliação dos Resultados e na Seção 5.5 são apresentadas as considerações finais.

5.2. Etapa I: Entender a Funcionalidade

Esta etapa é composta por três passos discutidos nas Seções 5.2.1, 5.2.2. e 5.2.3.

5.2.1. Passo I.1: Gerar Diagramas de Casos de Uso.

Neste passo deve-se identificar a funcionalidade do sistema em observação que será expressa por meio de diagramas de casos de uso. Para gerar esse diagrama de casos de uso utilizaram-se as diretrizes criadas para esse fim.

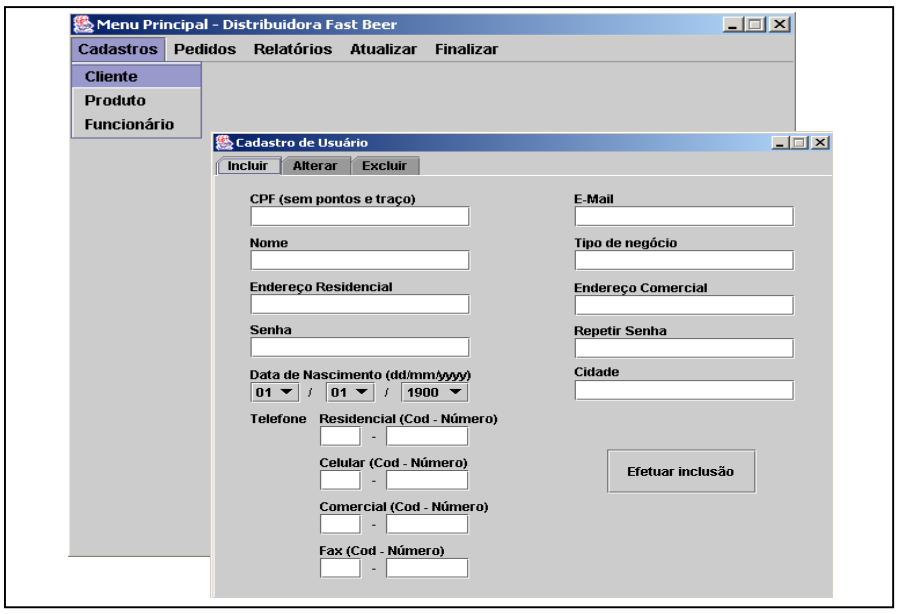


Figura 5.1 - Interface do Sistema de Distribuidora de Bebidas, em Destaque o Item Cliente do Menu Cadastros.

O primeiro passo é criar o ator usuário e identificar todos os menus como opção do sistema e para cada item do menu criar um caso de uso. Em seguida, verificar se toda a funcionalidade para aquele caso de uso foi realmente expressa. A Figura 5.1 apresenta a interface principal do sistema Orientado a Objetos ao fundo e a de "Cadastro de Usuário", à frente. Observando-se do menu "Cadastro de Usuário" têm-se três opções: Incluir, Alterar e Excluir. Assim, foi gerado o diagrama de Casos de Uso, Figura 5.2, e a especificação do curso normal e curso alternativo de cada caso de uso, Quadro 5.1.

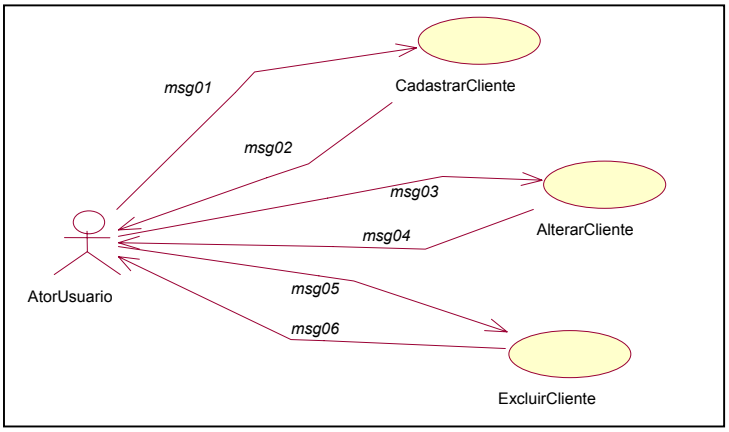


Figura 5.2 - Diagrama de Casos de Uso, Elaborado a Partir da Execução do Item Cliente do Menu Cadastros.

Quadro 5.1 - Descrição do Curso Normal e Curso Alternativo dos Casos de Uso Gerado na Figura 5.2.

Cadastrar Cliente	
Curso Normal	1. Obter msg01 = Nro do CPF do Cliente 2. Verificar que Cliente não existe. 3. Obter mais inf. msg01 = Nome, Endereço Residencial, E-Mail, Tipo de Negocio, Endereço Comercial, Senha, Repetir Senha, Cidade, Data de Nascimento, Telefone Residencial, Telefone Celular, telefone Comercial, Nro do Fax. 4. Efetuar Inclusão. 5. Emitir msg02 = "Cliente cadastrado".
Curso Alternativo	2. Cliente já cadastrado. 2.1. Emitir msg02 = "Cpf já cadastrado". 2.2. Abandonar caso de uso.
Alterar Cliente	
Curso Normal	1. Obter msg03 = Nro do CPF do Cliente 2. Verificar que Cliente existe. 3. Mostrar dados do Cliente. 4. Efetuar alteração. 5. Emitir msg04 = "Dados alterados".
Curso Alternativo	2. Cliente não existe 2.1. Emitir msg04 = "Cliente Inexistente". 2.2. Abandonar caso de uso.
Excluir Cliente	
Curso Normal	1. Obter msg05 = Nro do CPF do Cliente 2. Verificar que Cliente existe. 3. Excluir Cliente. 4. Emitir msg06 = "Cliente excluído"
Curso Alternativo	2. Cliente não existe 2.1. Emitir msg06 = "Cliente Inexistente". 2.2. Abandonar caso de uso.

5.2.2. Passo I.2: Gerar Casos de Teste

Exercitando os casos de uso, gerados no passo anterior, elaboraram-se os casos de teste para o sistema de distribuidora de bebidas. A Tabela 5.2 mostra os casos de teste para a opção Cadastrar Cliente.

Tabela 5.2 - Casos de Teste para a Opção Cadastrar Cliente.

Dados de Entrada	Resultados Esperados (Interface com o Usuário)	Resultados Esperados (Console com o Desenvolvedor)
nulo	Preencha os Campos	`java.lang.NullPointerException Funcionario.Localizar(Cliente.java:242)`
Nro do CPF = '**255985988-33**', Nome = '**José da Silva**', Endereço Residencial = '**Alameda das Papoulas**', E-Mail = '**js@zipmail.br**', Tipo de Negocio = '**bar**', Endereço Comercial = '**Rua dos coqueiros**', Senha = '**12345**', Repetir Senha = '**12345**', Cidade = '**São Carlos**', Data de Nascimento = '**12/01/1970**', Telefone Residencial = '**--**', Telefone Celular = '**9701 1111**', telefone Comercial = '**1224455**' , Nro do Fax = '**232325656**'	O cliente José da Silva foi Cadastrado com sucesso.	`SQL : UPDATE CLIENTE SET NOME = 'José da Silva', ENDERECO = 'Alameda das Papoulas', EMAIL = 'js@zipmail.br', TIPO = 'bar', ENDERECOCOMER = 'Rua dos coqueiros', SENHA = '12345', REPSENHA = '12345', CIDADE = 'São Carlos', DATANASC = '12/01/1970', TELRES = '--', TELCELULAR = '9701 1111', TELCOMER = '1224455' , FAX = '232325656'`

5.2.3. Passo I.3: Gerar Diagramas de Classes de Implementação

Para gerar o diagrama de classes de implementação a partir do código fonte, foi utilizada a ferramenta CASE Omondo (Omondo, 2004), um *Plug-in* da plataforma Eclipse (IBM e outros, 2004) para trabalhar com a notação UML (UML, 2004). O sistema é composto por 19 arquivos `.java` que implementam 69 classes, sendo que 63 classes tratam da interface com o usuário utilizando o pacote `javax.swing`, e 6 classes que tratam das regras de negocio do sistema. Essa diferenciação das classes é possível, pois todas as classes que tratam da interface com o usuário herdam da classe `JFrame`.

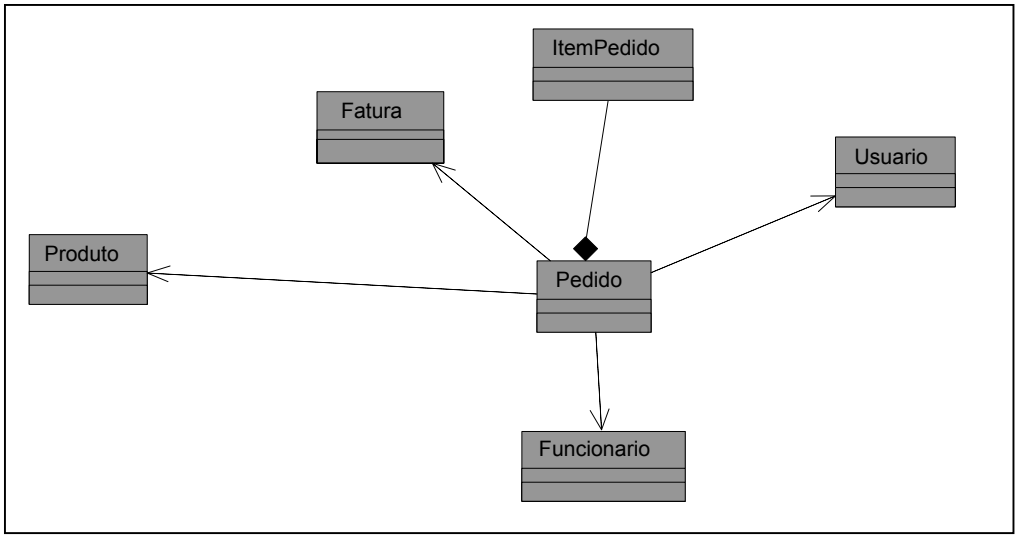


Figura 5.3 - Diagrama de Classes de Implementação Gerado pela Ferramenta *Omondo* (Omondo, 2004).

O diagrama de classes foi gerado pela ferramenta Omondo (Omondo, 2004), somente para as 6 classes que tratam das regras de negócio do sistema, Figura 5.3.

5.3. Etapa II: Tratar Interesses

Esta etapa é composta por quatro passos que devem ser realizados para cada um dos indícios de interesses que compõem a Lista de Indícios (Apêndice 1).

5.3.1. Passo II.1: Marcar o Interesse (primeira iteração)

O objetivo deste passo é encontrar os interesses que estão espalhados e entrelaçados no código fonte do sistema e marcá-los adicionando comentários no final da linha que contenha o interesse. Para isso utiliza-se a Lista de Indícios. Seguindo a ordem da Lista, o primeiro interesse a ser pesquisado no código fonte é o de Rastreamento.

Manualmente , o engenheiro de software busca nas classes implementadas, se algum dos indícios referentes a esse interesse existe. Em caso afirmativo, adiciona-se um comentário ao final da linha de código em que foi encontrado como exemplifica a Figura 5.4. Como o trecho de código referente ao interesse é utilizado somente uma única vez,ele não se encontra espalhado e entrelaçado na classe, não comprometendo, portanto, o entendimento do sistema. Assim, optou-se por não tratá-lo como aspecto e a implementação original foi preservada.

```
public void Excluir( int Num_Pedido )  {
    . . .
    ps1.clearParameters();
    ps1.setInt(1,Num_Pedido);
    ps1.executeUpdate();
    ps1.close();
    System.out.println("Dentro de Fatura.Excluir, após o método
    ps1.close()"); // Rastreamento, 01
  }
```

Figura 5.4 - Trecho de Código da Classe `Fatura`, que Contém a Única Linha do Sistema com o Interesse de Rastreamento.

O segundo indício pesquisado foi o referente ao Tratamento de Erros e não foi encontrado nesse sistema. O próximo indício que foi pesquisado, no código fonte do sistema, refere-se ao interesse de Tratamento de Exceção.

A Figura 5.5 mostra um trecho do método `Incluir()` da classe `Fatura` que continha o interesse e que foi marcado.

```
public void Incluir( float Preco_Final, int Num_Pedido )  {
  try{ // Tratamento de exceção, 01
     int cod ;
       ...
      ps1.close();
  } catch(SQLException e){ // tratamento de exceção, 01
    JOptionPane.showMessageDialog(null,"Erro ...");
    }
}
```

Figura 5.5 - Trecho de Código do Método `Incluir()` que contém o Interesse de Tratamento de Exceção.

Quando os indícios de algum interesse são pesquisados por todas as classes e é encontrado, o engenheiro de software deve decidir por implementá-lo ou não. Em caso afirmativo, deve-se então, encerrar a pesquisa por indícios de outros interesses e passar para o passo II.2. A pesquisa por indícios de outros interesses da Lista de Indícios será continuada na segunda interação do passo II.1.

5.3.2. Passo II.2: Modelar o Interesse (primeira iteração)

O segundo passo trata da modelagem do interesse encontrado no passo II.1 no diagrama de classes de implementação iniciado na etapa I.

O interesse de Tratamento de Exceção foi modelado em aspectos e adicionados ao diagrama de classes, segundo as diretrizes criadas e específicas para ele.

```
. . .
public void Incluir( float Preco_Final, int Num_Pedido )  {
    try{ // tratamento de exceção, 01
      int cod ;
      . . .
      ps1.close();}
    catch(SQLException e){ // tratamento de exceção, 01
      JOptionPane.showMessageDialog(null,"Erro ao acessar Banco de
      Dados", "SQL Exception", JOptionPane.INFORMATION_MESSAGE);}
}
. . .
```

Figura 5.6 - Trecho de Código da Classe `Produto` que Contém o Primeiro Trecho de Código do Interesse de Tratamento de Exceção.

O primeiro trecho de código com o interesse que foi modelado estava contido na classe `Produto`, no método `Incluir()`, Figura 5.6. Seguindo as diretrizes de modelagem para o interesse, primeiramente deve-se examinar o tipo de exceção que está sendo tratada, neste caso o tipo é `SQLException`. Como não havia aspecto criado para esse tipo de exceção, criou-se o aspecto `AspectSQLException`, adicionando-o ao diagrama de classes, Figura 5.7.

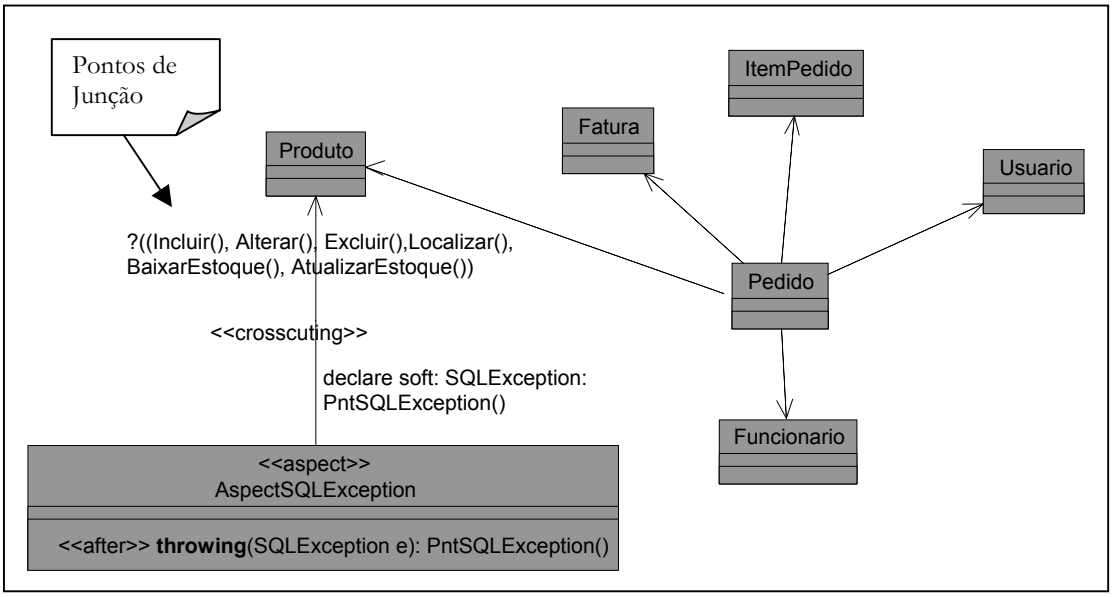


Figura 5.7 - Diagrama de Classes com o Aspecto `AspectSQLException`.

O próximo trecho de código marcado com sendo do interesse na classe Produto também tratava do mesmo tipo de exceção. Portanto, apenas o nome do método que continha esse interesse foi adicionado como ponto de junção no aspecto para esse tipo de exceção. O processo foi repetido por toda a classe adicionando assim como ponto de junção todos os métodos que continham trechos do interesse de Tratamento de Exceção e, posteriormente,

para todas as classes do sistema. Foram criados relacionamentos indicando que o aspecto `AspectSQLException` entrecorta cada classe que contém trechos do interesse.

Na classe `Funcionario` um outro tipo de exceção foi encontrado (o tipo `Exception`). Seguindo as diretrizes de modelagem criadas para o interesse, foi criado um novo aspecto, no diagrama de classes, Figura 5.8, com o nome `AspectException`. Os métodos que continham trechos do interesse com esse tipo de exceção, foram associados aos pontos de corte.

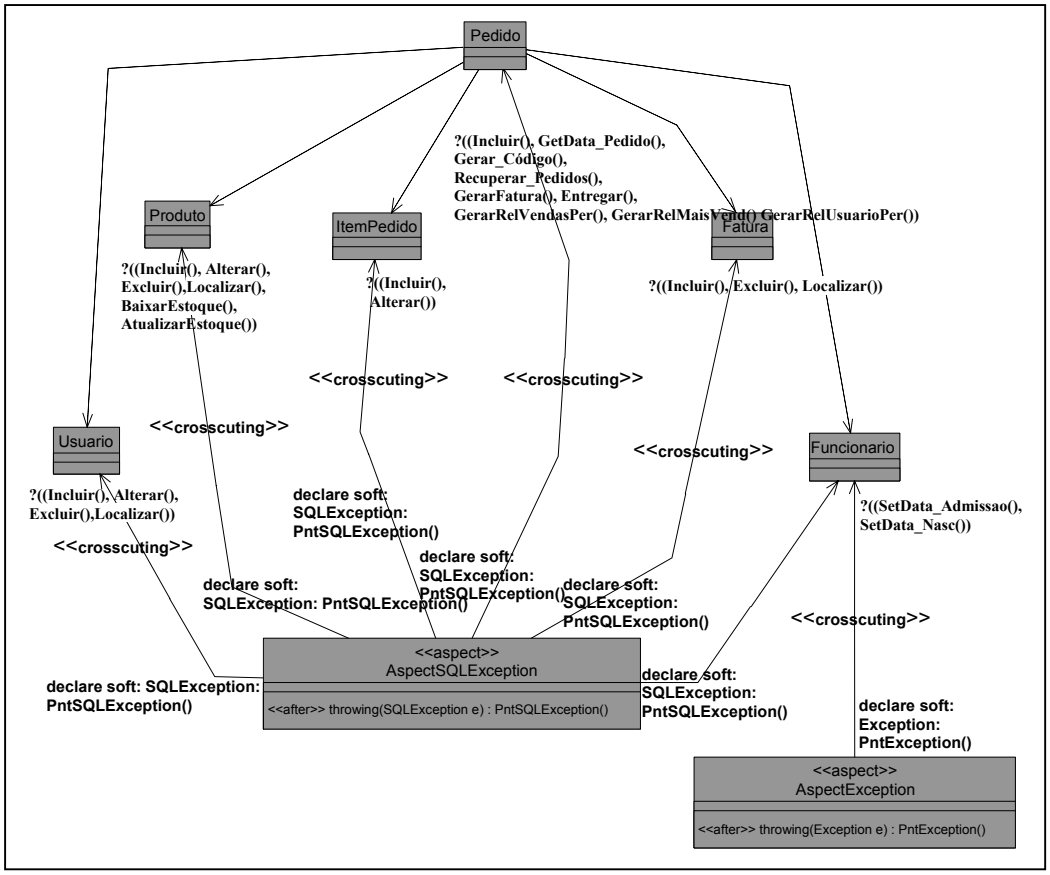


Figura 5.8 - Diagrama de Classes de Implementação do Sistema com o Interesse de Tratamento de Exceção Modelado em Aspectos.

Somente esses dois tipos de interesses de Tratamento de Exceção foram encontrados para esse sistema, passando-se assim para a implementação desses.

5.3.3. Passo II.3: Implementar o Interesse (primeira iteração)

Após a identificação e a modelagem dos interesses nos diagramas de classe, deve-se tratar da implementação utilizando a linguagem AspectJ. Para cada aspecto modelado no diagrama de classes, Figura 5.8, foi implementado um aspecto na linguagem AspectJ (Kiczales e outros 2001a e 2001b), seguindo as diretrizes para implementar cada um deles. O

aspecto `AspectSQLException` foi implementado adicionando ao ponto de corte `PntSQLException` todos os pontos de junção identificados e comentados no passo anterior modelar interesse. Com a existência de alguns pontos de junção com o mesmo nome, Como se observa na Figura 5.9, decidiu-se refinar o ponto de corte, adicionando-se caracteres coringas (*wildcards*) com o objetivo de generalizar alguns pontos de junção, Figura 5.10.

```
public aspect AspectSQLPersistence {
      pointcut PntSQLException():
            execution(public void Fatura.Incluir(..))
            || execution(public void ItemPedido.Incluir(..))
            || execution(public boolean Funcionario.Incluir(..))
            || execution(public boolean Usuario.Incluir(..))
            || execution(public boolean Produto.Incluir(..))
            || execution(public void Pedido.Incluir(..))...
```

Figura 5.9 - Trecho do Ponto de Corte `PntSQLException` com os Pontos de Junção que Contém o Mesmo Nome.

```
execution( * *.Incluir(..))
```

Figura 5.10 - Ponto de Junção Refinado com os Caracteres Coringa.

O corpo dos aspectos implementados incluindo as sugestões (*advices*), seguiram o modelo proposto nas diretrizes para implementar o interesse, diferenciando-se somente o tipo da exceção que está sendo tratada, Figura 5.11 e Figura 5.12.

```
public aspect AspectException {
   pointcut PntException():
      execution(public boolean setData_Admissao(..))
      || execution(public boolean setData_Nasc(..));
   declare soft : Exception : PntException();
   after()throwing (SQLException e): PntException() {
      try{
         throw new SQLException();}
      catch (SQLException e1){
         JOptionPane.showMessageDialog(null,"Erro ao acessar Banco
de Dados", "SQL Exception", JOptionPane.INFORMATION_MESSAGE);}
      }
}
```


Figura 5.11 - Corpo do Aspecto `AspectException`.

```
public aspect AspectSQLException {

   pointcut PntSQLException():
     execution( * *.Incluir(..))
     || execution(* *.Alterar())
     || execution(* *.Excluir(..))
     || execution(* *.Localizar(..))
      || execution(public Hashtable Produto.Recuperar_Produtos())
      || execution(public int Produto.Gerar_Codigo())
      || execution(public void Produto.baixarEstoque(..))
      || execution(public void Produto.atualizarEstoque(..))
      || execution(public String Pedido.getData_Pedido(..))
      || execution(public Vector Pedido.Recuperar_Pedidos(..))
      || execution(public void Pedido.entregar())
      || execution(public void Pedido.gerarFatura())
      || execution(public Vector Pedido.gerarRelVendasPer(..))
      || execution(public Vector Pedido.gerarRelMaisVend(..))
      || execution(public Vector Pedido.gerarRelUsuarioPer(..));

      declare soft : SQLException : PntSQLException ();

      after()throwing (SQLException e): PntSQLException () {
            try{
                  throw new SQLException();
            }
            catch (SQLException e1){
                  JOptionPane.showMessageDialog(null,"Erro ao
acessar Banco de Dados", "SQL Exception",
JOptionPane.INFORMATION_MESSAGE);
            }
      }
}
```

Pontos de Junção Refinados

Figura 5.12 - Corpo do Aspecto `AspectSQLException`.

Como a abordagem Aspecting segue um modelo de processo evolutivo o engenheiro de software deve retornar ao passo Marcar Interesse para que outros indícios de interesses que compõem a Lista de Indícios.

5.3.4. Passo II.4: Aplicar Testes de Regressão (primeira iteração)

Neste passo todos os casos de teste elaborados no passo I.2, que geram algum tipo de exceção como resultado esperado no console do desenvolvedor, foram executados após a reorganização do sistema. Como os resultados esperados, Tabela 5.2 – coluna 2, foram os mesmos documentados nos casos de teste, pode-se inferir que a funcionalidade permaneceu inalterada.

5.3.5. Passo II.1: Marcar o Interesse (segunda iteração)

Na segunda iteração os indícios do Interesse de Persistência em Banco de Dados Relacional, foram procurados. A inspeção foi realizada em todas as classes da aplicação, em todos os aspectos já inseridos anteriormente no sistema e todos os trechos de código reconhecidos com a gramática livre de contexto (Apêndice 1) foram marcados com comentários sendo adicionados ao final de cada uma das linhas que continha esse código.

Para exemplificar um caso em que o interesse foi marcado, apresenta-se o trecho de código da Figura 5.13 que se refere à classe `Fatura`. Os comentários que servem para identificar indício do interesse em questão podem ser observados também nessa Figura.

```
. . .
public void Incluir( float Preco_Final, int Num_Pedido )  {
      int cod ;
      ps1 = con1.prepareStatement( "Select MAX(Numero) as m from
      Fatura" );// persistencia BD relacional 01
      rs1 = ps1.executeQuery();// persistencia BD relacional 01
      if ( rs1.next() )// persistencia BD relacional 01
        cod = rs1.getInt("m") + 1;// persistencia BD relacional 01
      else
        cod = 1;
      ps1 = con1.prepareStatement( "Insert into Fatura values
      (?,?,?,?,?)" );// persistencia BD relacional 01
      ps1.clearParameters();// persistencia BD relacional 01
      ps1.setInt(1,cod);// persistencia BD relacional 01
      ps1.setDate(2,Data_Emissao);// persistencia BD relacional 01
      ps1.setDate(3,Data_Pagto);// persistencia BD relacional 01
      ps1.setFloat(4,Preco_Final);// persistencia BD relacional 01
      ps1.setInt(5,Num_Pedido);// persistencia BD relacional 01
      ps1.executeUpdate();// persistencia BD relacional 01
      ps1.close();// persistencia BD relacional 01
   }
. . .
```

Figura 5.13 - Trecho de Código da Classe `Fatura`, com a Presença de Indícios do Interesse de Persistência em Banco de Dados Relacional.

5.3.6. Passo II.2: Modelar o Interesse (segunda iteração)

O interesse de Persistência em Banco de Dados Relacional foi modelado em aspectos, Figura 5.14, seguindo as diretrizes para modelar criadas para esse interesse. Para cada classe que continha o interesse foi criado um aspecto específico que retira o código correspondente ao interesse, das classes, e o re-introduz às classes em tempo de compilação utilizando o conceito de introdução (*introduction*).

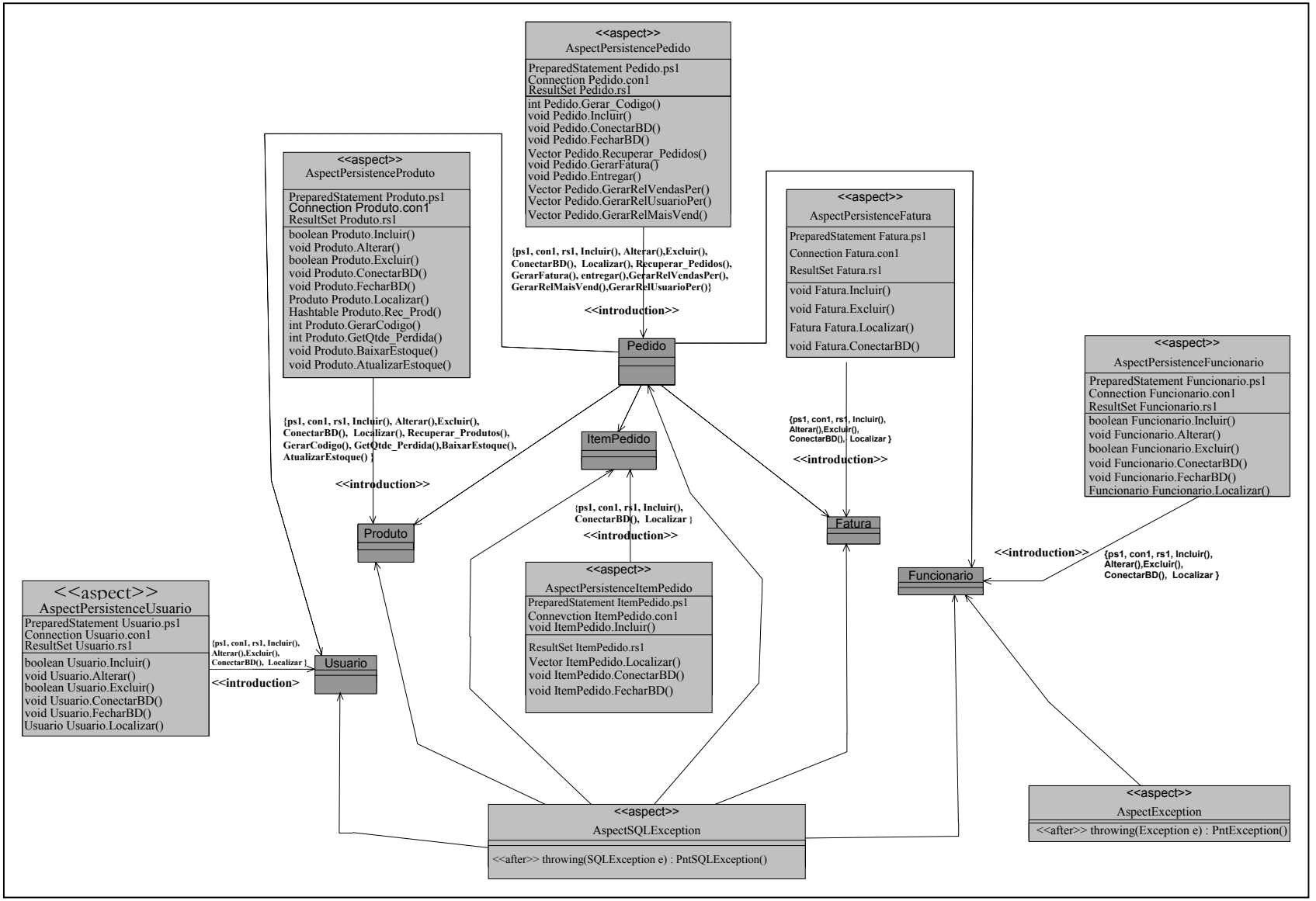


Figura 5.14 - Diagrama de Classes com os Interesses de Tratamento de Exceção (sem descrição nos relacionamentos) e de Persistência em Banco de Dados Relacional Modelados em Aspectos.

5.3.7. Passo II.3: Implementar o Interesse (segunda iteração)

O diagrama de classes com aspectos, gerado no passo anterior, e as diretrizes para implementar o interesse apóiam a implementação do aspecto de Persistência em Banco de Dados Relacional identificado.

O conceito de introdução (*introduction*) disponível na linguagem AspectJ (Kiczales e outros, 2001a e 2001b) permite que as linhas marcadas como interesse sejam colocadas nos aspectos re-colocadas nas classes que continham esse interesse, estaticamente em tempo de compilação. A Figura 5.15 exibe o trecho de código do aspecto que trata do interesse de Persistência em Banco de Dados Relacional na classe `Fatura`. O retângulo tracejado (a) corresponde ao método mostrado na Figura 5.13.

```
public AspectPersistenceFatura {

      PreparedStatement Fatura.ps1;
      Connection Fatura.con1;
      ResultSet Fatura.rs1;

(a)→  public void Fatura.Incluir( float Preco_Final, int Num_Pedido )
      { ... }
      public void Fatura.Excluir( int Num_Pedido ) { ... }
      public Fatura Fatura.Localizar( int Num_Pedido ) { ... }
      public void Fatura.ConectarBD(){ ... }
      public void Fatura.FecharBD(){ ... }
}
```

Figura 5.15 - Trecho de Código do Aspecto `AspectPersistenceFatura`.

5.3.8. Passo II.4: Aplicar Testes de Regressão (segunda iteração)

Neste passo todos os casos de teste elaborados no passo I.2, que geram algum tipo de persistência como resultado esperado no console do desenvolvedor, foram executados após a reorganização do sistema. Como os resultados esperados foram os mesmos documentados nos casos de teste, pode-se inferir que a funcionalidade do sistema permaneceu inalterada.

5.3.9. Passo II.1: Marcar o Interesse (terceira iteração)

Na terceira iteração da etapa II da abordagem Aspecting os indícios do Interesse de Persistência em Memória Temporária e indícios do Interesse de Programação Paralela foram considerados. Todas as classes obtidas na etapa I e os aspectos inseridos anteriormente foram

inspecionados para verificar se possuíam indícios relacionados a esses interesses, mas não foram encontrados.

A Lista de Indícios foi esgotada sendo que o sistema Orientado a Objetos original agora se encontra implementado com aspectos na AspectJ.

5.4. Avaliação dos Resultados

A proposta principal de se usar a programação Orientada a Aspectos é produzir sistemas em que os interesses estão separados da funcionalidade. A medição de software é importante para garantir a qualidade do produto, avaliar a produtividade das pessoas que o produzem e formar uma linha base para estimativas, (Presmman, 2002). Assim, esta seção tem o objetivo de avaliar a aplicação da Abordagem *Aspecting* no sistema de Distribuidora de Bebidas, Orientado a Objetos, comparando o código fonte do sistema antes e após a realização dessa. Para isso foram utilizadas métricas específicas para a Programação Orientada a Aspectos propostas por Sant´Anna e outros (2003).

O tamanho do código fonte pode indicar a quantidade de esforço necessária para entender os componentes do sistema. A separação de interesses propicia o aumento do entendimento do sistema, pois o código fica bem organizado e possibilitando a identificação de cada um dos seus componentes (aspectos e classes).

As métricas de tamanhos utilizadas neste estudo de caso são divididas em três tipos:

a) Tamanho do Vocabulário (*Vocabulary Size*): Conta o número de componentes do sistema. Cada nome de componente é contado como parte do vocabulário do sistema e as suas instâncias não são contadas.

b) Linhas de Código (*Lines of Code*): Conta o número de linhas de código, como tradicionalmente ocorre com medida de tamanho. Documentação e comentários de implementação, bem como linhas em branco, não são interpretadas como código.

c) Número de Atributos (*Number of Attributos*): Conta o vocabulário interno de cada componente, o número de atributos de cada classe ou aspecto. Atributos herdados não são incluídos na conta.

A Tabela 5.3 apresenta os resultados obtidos com a métrica Vocabulário do Sistema. Pode-se observar que o sistema Orientado a Objetos possui seis componentes (classes) enquanto que o Orientado a Aspectos possui nove, pois foram acrescidos os aspectos. Desse modo, nota-se um aumento de 120% no número de componentes do sistema. Sob essa avaliação infere-se que a versão OA seja menos eficiente que a OO. Portanto, cabe ao

engenheiro de software avaliar se ocorrendo um aumento de módulos do sistema na versão AO, ela é mais manutenível do que a versão OO.

Tabela 5.3 - Número de Componentes dos Sistemas.

Vocabulário do Sistema			
Sistema Orientado a Objetos		Sistema Orientado a Aspectos	
	Número de Componentes		Número de Componentes
Classes	6	Classes	6
		Aspectos	9
Total	**6**	**Total**	**15**

A Tabela 5.4 apresenta os resultados obtidos com a métrica Linhas de Código. As seis classes existentes no sistema Orientado a Objetos totalizam 1433 linhas de código. Já o sistema Orientado a Aspectos para essas mesmas classes apresentam redução de aproximadamente 60% para número de linhas de código. Contabilizando a introdução dos aspectos pode-se observar redução de 10% do número total de linhas de código do sistema Orientado a Objetos para o Orientado a Aspectos.

Tabela 5.4 – Número de Linhas dos Sistemas.

Linhas de Código			
Sistema Orientado a Objetos		**Sistema Orientado a Aspectos**	
Classes	LOC	Classes	LOC
Fatura	181	Fatura	100
Produto	282	Produto	85
ItemPedido	96	ItemPedido	43
Funcionario	246	Funcionario	128
Usuário	227	Usuario	103
Pedido	401	Pedido	126
		Sub-Total	**585**
		Aspectos	
		AspectPersistenceFatura	75
		AspectPersistenceProduto	175
		AspectPersistenceItemPedido	48
		AspectPersistenceFuncionario	105
		AspectPersistenceUsuario	110
		AspectPersistencePedido	275
		AspectSQLException	27
		AspectException	13
		Sub-Total	**810**
Total	**1433**	**Total**	**1395**

É importante notar que a métrica Número de Atributos, Tabela 5.5, apresenta o mesmo resultado tanto para o sistema Orientado a Objetos quanto para o Sistema Orientado a Aspectos. Porém, as classes que contêm as regras de negócio do sistema Orientado a Aspectos tem 52 atributos enquanto que o sistema Orientado a Objetos tem 70. Portanto, pode-se inferir que pelo tamanho dos sistemas a versão Orientada a Aspectos é mais fácil de entender do que a versão Orientada a Objetos.

Tabela 5.5 - Número de Atributos dos Sistemas.

Número de Atributos			
Sistema Orientado a Objetos		**Sistema Orientado a Aspectos**	
Classes	Número de Atributos	Classes	Número de Atributos
Fatura	8	Fatura	5
Funcionario	15	Funcionario	12
ItemPedido	7	ItemPedido	4
Pedido	12	Pedido	9
Produto	12	Produto	9
Usuário	16	Usuário	13
		Sub-Total	**52**
		Aspectos	Número de Atributos
		AspectPersistenceFatura	3
		AspectPersistenceFuncionario	3
		AspectPersistenceItemPedido	3
		AspectPersistencePedido	3
		AspectPersistenceProduto	3
		AspectPersistenceUsuario	3
		AspectSQLException	0
		AspectException	0
Total	**70**	**Total**	**70**

5.5. Considerações Finais

Este Capítulo apresentou a aplicação da Abordagem *Aspecting* em um sistema orientado a objetos, implementado na linguagem Java, utilizado como estudo de caso. As diretrizes propostas pela Abordagem foram utilizadas e notou-se que elas guiam o engenheiro de software para a reestruturar um sistema orientado a objetos para sistema orientado a aspectos (Ramos e outros, 2004b).

A etapa de Entendimento da Funcionalidade foi efetuada com sucesso, porém é uma etapa que demanda tempo e esforço por parte do engenheiro de software, que podem ser

minimizados com a utilização de ferramentas de engenharia reversa para que se obtenha a documentação do sistema.

A Abordagem *Aspecting* não dá insumos para realizar o refinamento dos aspectos, ponto esse que pode ser melhorado se forem utilizados idiomas, como os de Hanenberg e Unland (2002). A melhoria da qualidade do software gerado, quando os aspectos forem refinados e generalizados, pode ser comprovada com a utilização das métricas de tamanho, pois com a generalização há diminuição de código redundante.

A Lista de Indícios não considera atributos e comandos que auxiliam a implementação do interesse e que não tenham palavras reservadas no contexto em que o interesse está inserido. Assim, o engenheiro de software deve analisar esses casos para adicioná-los ou não ao aspecto que será implementado. Um exemplo desse caso é o atributo `cod`, do tipo `int`, na classe `Fatura` no método `Incluir()` exibida na Figura 5.13. Pela Lista de Indícios ele não está marcado, mas faz parte da funcionalidade do método. Logo, é também incluído no aspecto.

O interesse de Persistência em Banco de Dados Relacional pode ser considerado um interesse funcional, já que todos os sistemas no contexto de sistemas de informação, necessitam persistir seus dados. Se assim for considerado, a implementação com aspectos não atinge seu objetivo, pois pode ocorrer maior dificuldade no entendimento do sistema, uma vez que vários fragmentos o código funcional encontram-se espalhados.

Algumas ferramentas para auxiliar encontrar os indícios dos interesses não foram utilizadas neste estudo de caso. A Ferramenta *Aspect Mining Tool* AMT (Hannemam e Kiczalez, 2001) pode ser utilizada juntamente com a abordagem para agilizar o passo de marcar o interesse. Outros recursos como o próprio serviço de busca (*Search*) do ambiente de desenvolvimento pode ser utilizado para auxiliar a busca de indícios.

A etapa II contém o passo II.4 que propõe que se teste a funcionalidade do sistema orientado a aspectos. Nesse passo somente é possível testar se o sistema gerado atende aos mesmos requisitos descritos pelos casos de teste gerados na primeira etapa. Para alguns sistemas não basta realizar testes somente verificando-se os dados de entrada e saída, seria necessário que a estrutura do sistema também fosse testada. Para esses casos de teste não se pode assegurar que a funcionalidade permanece inalterada somente com teste de regressão, e devem ser aplicados outros tipos de teste, como por exemplo, o teste de unidade. Para comparar todos os benefícios da Abordagem *Aspecting* para os sistemas Orientados a Aspectos outras métricas devem ser aplicadas, como as propostas por Sant´Anna (2003).

Um ponto a ser ressaltado é que não foi considerado, neste estudo de caso, o tempo de aprendizagem dos conceitos da Programação Orientada a Aspectos pelo engenheiro de software. Isso pode ser um fator importante quando se considera a medição do tempo gasto para a realização da reorganização do código.

Capítulo 6

Considerações Finais

6.1. Introdução

Sistemas Orientados a Objetos não possuem mecanismos que possibilitam a separação de interesses, referentes aos requisitos não funcionais, do código fonte funcional. Assim, muitos inconvenientes, como por exemplo, a dificuldade de interação com novas tecnologias, o aumento dos custos de manutenção e a reusabilidade do código ficam comprometidas (Tarr e outros, 1999). Abordagens que podem amenizar esses problemas são de extrema relevância.

Sendo assim, este trabalho realizou estudos de caso com a aplicação de técnicas de separação de interesses e, em especial, as técnicas da programação Orientada a Aspectos visando elaborar uma abordagem que guie o engenheiro de software de forma fácil e ágil a migrar de sistemas orientados a objetos para sistemas orientados a aspectos.

A abordagem denominada *Aspecting* propõe uma forma direta e ordenada na migração de sistemas Orientados a Objetos para sistemas Orientados a Aspectos. Para contornar dificuldade de reconhecer imediatamente um interesse, utilizou-se de indícios que possam

indicar a presença desse entrelaçado ao código fonte. Esses foram reunidos em uma lista, denominada de Lista de Indícios, e sete tipos de interesses podem ser identificados espalhados e entrelaçados ao código funcional do sistema com o auxílio dessa lista. Diretrizes que auxiliam a modelagem dos aspectos em diagramas de classe de projeto, utilizando extensões da notação UML, foram criadas. A partir do código fonte Orientado a Objetos e do diagrama de classes de projeto, diretrizes foram criadas para auxiliar a implementação desses aspectos na linguagem AspectJ.

Outras diretrizes como: para extrair a funcionalidade do sistema a partir da sua execução; para extrair o diagrama de classes de projeto a partir do código fonte de sistemas Orientados a Objetos e implementados na linguagem Java; e para guiar a comparação da funcionalidade dos sistemas originais, Orientados a Objetos, com a versão Orientada a Aspectos resultante, também foram criadas para completar a abordagem *Aspecting*.

A *Aspecting* tem três etapas distintas:

- Entender a Funcionalidade do Sistema: tem dois passos distintos que cuidam da geração dos diagramas de classes de projeto e dos de casos de uso a partir do código fonte e de sua execução;
- Tratar Interesses: identifica e, posteriormente, implementa os interesses encontrados um a um. É composta de três passos, que de forma evolutiva faz o refinamento para que a implementação resultante tenha o código funcional separado do não funcional. Os passos são: marcar um interesse no código fonte a partir da Lista de indícios; modelar o interesse encontrado com o auxílio das diretrizes para modelagem; e implementar o aspecto que foi adicionado ao diagrama de classes de projeto.
- Comparar Sistema Orientado a Aspectos com o Orientado a Objetos: avalia a funcionalidade do sistema Orientado a Aspectos resultante comparando-o com a descrição de casos de uso, gerados na etapa I, verificando que a funcionalidade do sistema foi preservada.

A vantagem que pode ser apontada pelo fato da etapa Tratar Interesse ser evolutiva é que a cada iteração, um tipo de interesse é implementado, deixando, assim, o código fonte com menos um interesse entrelaçado e espalhado. Com isso é possível que o engenheiro de software tome a decisão de parar e não mais implementar outros interesses, por encontrar o código pouco entrelaçado.

A desvantagem observada quanto à iteratividade da etapa é: por serem implementados aspectos de um tipo de interesse a cada iteração, são descartadas as interações e os

relacionamentos existentes entre os interesses. Nesses casos, a funcionalidade pode ser alterada, pois existem alguns casos em que um interesse pode ter precedência sobre outro interesse. Quando isso ocorrer torna-se necessário seguir a ordem pré-estabelecida da Lista de Indícios. Portanto, quando o engenheiro de software decidir adicionar um novo indício de um interesse na Lista de Indícios ele deve antes verificar se esse novo interesse tem ou não precedência sobre outro(s) interesse(s) da Lista.

Na abordagem proposta deseja-se reusar o código fonte existente, separando os interesses que estão espalhados e entrelaçados. Porém, se o processo é de engenharia avante a elaboração de casos de uso já possibilita a implementação com aspectos (Sousa e outros, 2003), (Souza e outros, 2004). Desse modo, as diretrizes elaboradas para a modelagem do diagrama de classes de projeto com aspectos e para a implementação de aspectos podem ser seguidas, sem que a marcação dos interesses no código fonte original tenha que ocorrer.

6.2. Resultados Obtidos

A abordagem Aspecting foi elaborada a partir da realização de três estudos de caso em diferentes contextos. Os sistemas utilizados estavam implementados no paradigma Orientado a Objetos na linguagem Java. Os estudos de caso iniciaram-se com a obtenção da funcionalidade dos sistemas, a partir de sua execução e da extração do diagrama de classes em nível de projeto, a partir do código fonte. Esses sistemas passaram por um processo de identificação de interesses (Kiczales, 1997) que estivessem entrelaçados e espalhados em seu código fonte. Após essa identificação, passou-se à fase de modelagem dos interesses em aspectos, incorporando-os ao diagrama de classe obtido anteriormente. Em seguida, esses aspectos foram implementados na linguagem AspectJ, extensão da linguagem Java, juntamente com os requisitos funcionais de forma a preservar a funcionalidade do sistema original. A comparação dos dois sistemas, original e o atual, quanto à funcionalidade é realizada com base nos artefatos gerados na primeira etapa da abordagem.

Após a observação dos três sistemas dos estudos de caso, foi possível aprimorar cada uma das fases, e criar uma abordagem que conduz de forma ordenada à explicitação dos aspectos em um sistema Orientado a Objetos e a sua posterior implementação no paradigma Orientado a Aspectos.

Para cada etapa, foram analisadas técnicas e ferramentas CASE que podem agilizar e melhorar a documentação. A Tabela 6.1 mostra o mapeamento feito nos três estudos de caso,

quais passos foram abstraídos para a abordagem *Aspecting* e quais técnicas e ferramentas foram propostas para esses passos.

Tabela 6.1 - Comparação das Etapas Realizadas nos Estudos de Casos com os Passos da Abordagem *Aspecting*.

Técnicas utilizadas nas etapas dos três estudos de caso	**Passos da Abordagem *Aspecting***	**Técnicas e/ou Ferramentas utilizadas para apoiar a *Aspecting***
1- Obtenção do documento de requisitos de forma *Ad-hoc*.	I.1 Gerar Diagramas de Casos de Uso	- Diagramas de casos de uso; - Diretrizes para extrair os diagramas de casos de uso a partir da execução do sistema.
2- Obtenção dos Casos de Teste	I.2. Gerar Casos de Teste	- Ferramenta Jtest (2004).
3- Obtenção do Diagramas de Classes de projeto de forma *Ad-hoc*.	I.3 Gerar Diagrama de Classes de Projeto	- Ferramenta: Omondo (2004); - Diretrizes para extrair o diagrama de classes a partir do código fonte
4- Busca dos interesses conhecidos na literatura da área, que possam estar espalhados e entrelaçados no código fonte do sistema.	II.1 Marcar o Interesse	- Ferramenta: Aspecting Mining Tool (Hannemann e Kiczales, 2001); - Lista de Indícios.
5- Modelagem dos interesses de forma *Ad-hoc*, com a utilização de diversas extensões da notação UML para a Programação Orientada a Aspectos.	II.2 Modelar o Interesse	- Diretrizes para a modelagem do interesse; - Padronização da notação utilizada (Palwak, 2002), (Camargo, 2004).
6- Implementar os interesses em aspectos de forma *Ad-hoc*.	II.3 Implementar o Interesse	- Diretrizes para implementação do interesse.
7- Testar a funcionalidade dos sistemas Orientados a Aspectos a partir dos Casos de Teste.	II.4 Aplicar Testes de Regressão.	- Técnicas de Testes de Regressão - Ferramenta: Jtest (2004)..

Um quarto sistema, desenvolvido em Java, para atender os pedidos de uma distribuidora de bebida, foi utilizado como estudo de caso para validar a abordagem *Aspecting*. No sentido de comparar o sistema original e o orientado a aspectos, métricas de tamanho do software foram utilizadas. Os resultados indicaram que a versão do sistema orientada a aspectos, com *Aspecting,* é mais fácil de ser entendido, portanto de mais fácil manutenção devido à redução no número de linhas de código e a organização final do código em classes e aspectos.

6.3. Trabalhos Futuros

Como continuidade deste trabalho existem algumas possibilidades descritas a seguir:

- Elaborar uma ferramenta que busque pelo código fonte de sistemas Orientados a Objetos, implementados em Java, por indícios de interesses pré-relacionados baseando-se Lista de indícios, utilizando gramáticas de livre contexto criadas para eles.
- Adaptar as etapas da abordagem *Aspecting* em outros paradigmas e outras linguagens, visando não somente o paradigma Orientado a Objetos e a linguagem Java. Assim, a abordagem se tornaria genérica, podendo auxiliar a migração de qualquer paradigma para o orientado a aspectos.
- Aplicar as Diretrizes de modelagem e implementação criadas para a abordagem *Aspecting,* no desenvolvimento de softwares Orientados a Aspectos.
- Realização de mais estudos de casos para adicionar indícios de outros interesses, não encontrados nos estudos de caso realizados, na Lista de Indícios e na criação de diretrizes para modelagem e implementação desses interesses.
- Criar generalizações do código fonte dos aspectos que implementam os interesses, a fim de elaborar pequenos *Frameworks* de implementação. Esses *Frameworks* podem ser utilizados no processo de desenvolvimento de softwares Orientados a Aspectos.

Referências Bibliográficas

AspectJ Team. *The AspectJ Programming Guid*e. 2003. Disponível em: http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/index.html. Último acesso em: 12/2003.

Bennet, K. H. e Rajlich, V. T. Software Maintenance and Evolution: a Roadmap. In: Anthony Finkelstein, ed. The Future of Software Engineering. Limerick, Ireland. ACM Press, Páginas 75-87. 2000.

Binder, R. V. Testing. Object-Oriented System, Addison Wesley Longman. 1143 p. 2000.

Cagnin, M. I.; Penteado, R. D.; Masiero, P. C. – Reengenharia com o uso de Padrões de Projeto. In: Simpósio Brasileiro de Engenharia de Software, SBES'99, 13, Florianópolis-Santa Catarina. Anais, páginas. 273-288, Outubro 1999.

Camargo, V. V. Reengenharia Orientada a Objetos de Sistemas COBOL com a Utilização de Padrões de Projeto e Servlets. In: Dissertação de Mestrado – Programa de Pós Graduação em Ciência da Computação, Universidade Federal de São Carlos, São Carlos - SP, 2001.

Camargo, V.V.; Ramos, R.A.; Penteado, R.A.D.; Masiero, P.C. Projeto Baseado em Aspectos do Padrão Camada de Persistência. In: Simpósio Brasileiro de Engenharia de Software (SBES), Manaus, 2003.

Camargo, V.V. Um Perfil UML para Projeto de Sistemas Orientados a Aspectos. In: Relatório Técnico - Universidade de São Paulo – ICMC/USP, São Carlos – SP, Abril de 2004.

Chavez, C.F.G.; Lucena, C.J.P. Design Support for Aspect-Oriented Software Development. In: Doctoral Symposium and Pôster Section of OOSPLA 2001. Tampa Bay, Florida, USA, Outubro, 2001.

Chikofsky E. J.; J. H. Cross II. *Reverse Engineering and Design Recovery: A Taxonomy*. In: *IEEE Software*, pág. 13–17, Janeiro 1990.

Chung, L.; Nixon, B.; Yu, E.; Mylopoulos, J. Non-functional requirements in software engineering. In: Boston: Kluwer Academic, pág. 439, 1999.

Coady, Y.; Kiczales, G.; Feeley, M.; Smolyn G. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In: Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9), Setembro 2001.

Coleman, D. Object-Oriented Development - The Fusion Method. Prentice Hall, 1994.

Cysneiros, L.M.; Leite, J.C.S.P. Definindo Requisitos Não Funcionais. In: Simpósio Brasileiro de Engenharia de Software (SBES´97)„ pág. 49-54, Outubro 1997.

Czarnecki, K.; Eisenecker, U. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

Deitel, H.M.; Deitel, P.J. Java. Como Programar. Bookman, 3ª Edição, 2000.

Demeyer, S.; Ducasse, S.; Nierstrasz, O. A Pattern Language for Reverse Engineering. In: 4ª European Conference on Pattern Languages of Programming and Computing, (EuroPLOP'1999). Paul Dyson (Ed.) Universitätsverlag Konstanz GmbH, Konstanz, Alemanha, Julho 1999.

Demeyer, S.; Ducasse, S.; Nierstrasz, O. - A Pattern Language for Reverse Engineering. In: 5th European Conference on Pattern Languages of Programming and Computing, (EuroPLOP'2000), Andreas Ruping(Ed.), 2000a.

Demeyer, S.; Ducasse, S.; Nierstrasz, O. - Tie Code and Questions: a Reengineering Pattern, In: 5th European Conference on Pattern Languages of Programming and Computing, (EuroPLOP'2000), Andreas Ruping(Ed.), 2000b..

Eisenecker, U. W.; Czarnecki, K. *Template-Metaprogrammierung in C++*. IN: OOP '99, pág. 181-198, München, Alemanha 1999.

Elrad, T. Aksit, M.; Kiczales, G.; Lieberherr, K.; Ossher, H.. *Discussing Aspects of AOP*. In: Anais do ACM, 2001a.

Elrad, T.; Filman R. *Bader A. Aspect-Oriented Programming.* In: Anais do ACM, 2001b.

Gamma, E.; Helm, R.; Johnson, R.E.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

GNU - Grupo de desenvolvedores para a Linguagem Java – Códigos fonte disponíveis em: http://www.cacas.org/java/gnu/regexp. Último acesso em 05/2003.

Hanenberg, S.; Unland, R. AspectJ Idioms for Aspect-Oriented Software Construction. In: 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), Boston, MA, Março 17, 2003.

Hannemann, J.; Kiczales, G. Overcoming the Prevalent Decomposition in Legacy Code. In: Workshop on Advanced Separation of Concerns (Proceedings), International Conference on Software Engineering. Toronto, Canada, Maio 2001.

Harrison, W.; Ossher, H. *Subject-Oriented Programming (a critique of pure objects)*. In: Conferência de Programação Orientada a Objetos: Sistemas, Linguagens e Aplicações (OOPSLA). pág. 411-428, Washington, ACM, 1993.

Highley, T.; Lack, M.; Myers, P. Aspect Oriented Programming: A Critical Analysis of a New Programming Paradigm. [S.l.], Maio 1999.

Hilsdale, E.; Hugunin, J. Advice Weaving in AspectJ. Submetido à 3rd International Conference on Aspect-Oriented Software Development – AOSD. April 2004.

Hirschfeld, R. AspectS – Aspect-Oriented Programming with Squeak. In: DoCoMo Euro-Labs Technical Report, ITR-FNL-022, Fevereiro 2003.

IBM e outros *Eclipse.or*g. Disponível em: http://www.eclipse.org. Último acesso em 04/2004.

Jacobson, I. Object-Oriented Software Engineering - A Use Case Driven Approach, Addison-Wesley Publish Company, 1992, 528 p.

Java, Sun Microsystems – Disponível em: http://java.sun.com/learning/tutorial/index.html. Último acesso em 04/2004.

Jtest – Ferramenta para geração de casos de testes de funcionalidade. Disponível em: http://www.ni.com.br/produtos/p_jtest.html. Último acesso em 05/2004

Kang, K. C. *Feature – oriented Development of Applications for a Domain.* In: 5ª Conferência Internacional de Reuso de Software. Victoria, British Columbia, Canada. IEEE , pág. 354-355. 2 - 5 de Junho 1998.

Kiczales, G.; Lamping, J.; Mendhekar, A. RG: A Case-Study for Aspect-Oriented Programming. In: SPL97. Xerox Palo Alto Research Center, Technical Report, 1997.

Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J. *Griswold, W.G. Getting Started with AspectJ*. In: Anais do ACM, pág. 59-65, Outubro 2001a.

Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M. *An Overview of AspectJ.* In: ECOOP, Technical Report, pág. 327-353, 2001b.

Kulak, D.; Guiney, E. *Use Cases: Requirements in Context*. Addison-Wesley, 2000. 329 p.

Kulesza, U.; Silva, D. M. Reengenharia do Projeto do Servidor Web JAWS Utilizando Programação Orientada a Aspectos. In: XIV Simpósio Brasileiro de Engenharia de Software, Sessões técnicas, SBES, 2000.

Laddad, R. AspectJ in Action: Practical Aspect-Oriented Programming. In: Manning Publications Company, Connecticut – USA, 2003. 512 p.

Lemos, G. S. PRE/OO – Um Processo de Reengenharia Orientado a Objetos com Ênfase na Garantia de Qualidade. In: Dissertação de Mestrado, Programa de Pós Graduação em Ciência da Computação, Universidade Federal de São Carlos, São Carlos - SP, 2002.

Lieberherr, K. J. e outros. *Adaptive Object-Oriented Programming Using Graph-Based Customization.* In: Anais do ACM, vol. 37, pág. 94-101, 1994.

Lippert M.; Lopes, C.V. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. In: Procceedings of the 22nd International Conference of Software Engineering (ICSE'2000). Limmerick, Ireland. IEEE Computer Society. Junho, 2000.

Mcgregor, J. Testing Object-Oriented Components. In: 10th European Conference on Object-Oriented Programming. Tutorial Notes. July 1996.

Noda, N.; Kishi, T. *Implementing Design Patterns Using Advanced Separation of Concerns.* In: OOPSLA 2001, Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa Bay, FL, Outubro 2001.

Omondo – Ferramenta para Modelagem – *Plug-in* disponível em http://www.omondo.com.Último acesso em 04/2004.

Ossher, H.; Tarr, P. *Multi-dimentional separation of concerns in Hyperspace.* In: Aspect-Oriented Programming Workshop at ECOOP'99, Finlândia: Springer-Verlag, 1999.

Pawlak, R., Duchien, L., Florin G., Legong-Aubry, F., Seinturier, L, Martelli, L. *A UML Notation for Aspect-Oriented Software Design*. In: Workshop of Aspect Oriented Modeling with UML of Proceedings of Aspect Oriented Software Development Conference (AOSD) 2002.

Penteado, R. A. D. Um Método para Engenharia Reversa Orientada a Objetos. In: Tese de Doutorado, Instituto de Física de São Carlos, Universidade de São Paulo - São Carlos. 1996.

Penteado, R. A. D.; Masiero, P. C.; Braga, R. T. V**.** Improving the Quality of Legacy Code by Reverse Engineering. In: ISAS'98, 4th International Conference on Information Systems, (Analisys and Sinthesis), Orlando – Florida, p. 364-370, julho 1998.

Pfleeger, S. L.. *Software Engineering – Theory and Practic*e. Prentice Hall Inc., New Jersey, 2004.

Portal Java - Downloads de código fonte de sistemas implementados em Java. Disponível em: http://www.portaljava.com.br. Último acesso em 04/2004.

Pressman, R. Engenharia de Software. Makron Books, 5ª edição, 2002.

Ramos, R.A.; Camargo, V.V.; Penteado, R.A.D.; Masiero, P.C. Reuso da Implementação Orientada a Aspectos do Padrão de Projeto Camada de Persistência. In The Fourth Latin American Conference on Pattern Languages of Programming - SugarLoafPLoP, Fortaleza-CE, agosto 2004a.

Ramos, R.A.; Penteado, R.A.D.; Masiero, P.C. Um Processo de Reestruturação de Código Baseado em Aspectos. Em: Simpósio Brasileiro de Engenharia de Software (SBES), Brasília- DF, outubro 2004b.

Rashid, A.; Chitchyan R. Persistence as an Aspect. In: 2nd International Conference on Aspect-Oriented Software Development, AOSD 2003, Boston, Massachusetts, USA. ACM, março 2003.

Rational Corporation. Unified Modeling Language. Disponível em: http://www.rational.com. Disponível em 01/2004.

Recchia, E. L.; Penteado R. D. Família de Padrões para Conduzir o Processo de Engenharia Reversa Orientada a Objetos de Sistemas Legados Orientados a Procedimentos. In: *The Second Latin American Conference on Pattern Languages of Programming, 2002*.

Ross, D., T. Structure Analysis (SA): A language for communicating Ideas. In: IEEE Transaction Software Engineering, 1977.

Sant'Anna, C.; Garcia, A.; Chavez, C.; Lucena C.; Staa A. *On Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework*. In: Simpósio Brasileiro de Engenharia de Software (SBES), Manaus, 2003.

Stein, D., Hanenberg, S., Unland, R. Designing Aspect-Oriented Crosscutting in UML. In: Workshop Aspect-Oriented Modeling with UML, AOSD, Enschede, Abril, 2002.

Schneider, G.; Winters, J. P. *Applying Use Cases, A Practical Guide*. Second Edition, Addison-Wesley, 2001. 245 p.

Souza, G.; Silva, I.; Castro, J. *Adapting the FRN Framework to Aspect-Oriented Requirements Engineering*. In: Simpósio Brasileiro de Engenharia de Software (SBES), Manaus, 2003.

Souza, G.; Soares, S.; Borba, P.; Castro, J. Separation of Concerns from Requirements to Design: Adapting an Use Case Driven Approach. In: Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design. Workshop at International Conference on Aspect-Oriented Software Development, AOSD 2004. 22 a 26 de Março de 2004, Lancaster, UK.

Tarr, P.; Ossher, H.; Harrision, W.; Sutton, S. M. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: International Conference on Object-Oriented Programming (ICSE), 1999

Tarr, P.; Ossher, H. *Hyper/J user and installation manual*. Disponível em: http://www.research.ibm.com/hyperspace. Último acesso em: 05/2003.

Unland, R.; Hanenberg, S.; Stein, D. *Designing Aspect-Oriented Crosscutting in UML*. In: Workshop Aspect-Oriented Modeling with UML, AOSD, Enschede, Abril 2002.

UML – Unified Modeling Language. Página com mais informações disponível no endereço: http://www.uml.org. Último acesso em 02/2004.

Yoder, J. W.; Johnson, R. E.; Wilson, Q. D. Connecting Business Objects to Relational Databases. In: Conference on the Pattern Languages of Programs 5 (PLOP). Monticello-IL, EUA, 1998.

Weiss, D., Lai, C. T. R. Software Product-Line Engineering: a family-based software development process. Ed. Addison Wesley, 1999. 426 p.

Apêndice 1

Lista de Indícios

- **Rastreamento (*Tracing*):**

Métodos que imprimem mensagens de texto, tais como:

- `System.out.print("`<Mensagem>`");`
- `System.out.println("`Mensagem>`");`

Onde em <Mensagem> deverá conter mensagens que informam ao desenvolvedor os nomes ou valores dos métodos, ou mensagens indicando a posição física do método no código fonte.

- **Tratamento de Erros:**

Comandos de decisão, como:

`if`, `else` e/ou `switch` ; e

No bloco de decisão deverá conter:

- Impressão de mensagens de erros alertando o usuário do sistema, e/ou
- Tratamento da variável de entrada para que essa seja válida para aquele contexto, e/ou
- Finalização do Sistema.

- **Tratamento de Exceção**:

Trechos de código que sejam reconhecidos pela gramática livre de contexto abaixo:

```
<i.trat.exceção>= 'try' '{' <statements> '}' 'catch' '('<statements>
')' '{'<statements>'}' 'finally' '{' <statements> '}' |
'try' '{' <statements> '}' 'catch' '('<statements> ')'
'{'<statements>'}'
```

- **Persistência em Banco de Dados Relacional**:

Trechos de código que sejam reconhecidos pela gramática livre de contexto abaixo:

```
<i.Persistencia.BD> = 'Connection' <statements> |
 'Connection' <statements> <SQL>|
 <SQL> <statements> 'Connection'|
'PreparedStatement' <statements>|
'PreparedStatement' <statements> <SQL>|
 <SQL> <statements> 'PreparedStatement'|
'ResultSet' <statements>|
'ResultSet' <statements> <SQL>|
 <SQL> <statements> 'ResultSet'
```

- **Programação Paralela (*Thread*):**

Trechos de código que sejam reconhecidos pela gramática livre de contexto abaixo:

```
<i.Programacao.Paralela> = 'extends' 'Thread' <statements> 'run()'
'{'<statements> '}' | 'extends' 'Thread' <statements> 'run()' '{'
'Thread' '.' <statements> <statements> '}' | 'extends' 'Thread'
<statements> 'run()' '{' <statements> 'Thread' '.' <statements>
<statements> '}' | 'extends' 'Thread' <statements> 'run()' '{'
<statements> 'Thread' '.' <statements> '}' | 'implements' 'Runnable'
<statements> 'run()' '{'<statements> '}' | 'implements' 'Runnable'
<statements> 'run()' '{' 'Thread' '.' <statements> <statements> '}' |
'implements' 'Runnable' <statements> 'run()' '{' <statements> 'Thread'
'.' <statements> <statements> '}' | 'implements' 'Runnable'
<statements> 'run()' '{' <statements> 'Thread' '.' <statements> '}' |
'new' 'Thread' '(' <statements> ')'
```

- **Persistência em Memória Temporária (*Buffering*):**

Trechos de código que sejam reconhecidos pela gramática livre de contexto abaixo:

```
<i.Programacao.Paralela> = 'extends' 'Thread' <statements> 'run()'
'{'<statements> '}' | 'extends' 'Thread' <statements> 'run()' '{'
'Thread' '.' <statements> <statements> '}' | 'extends' 'Thread'
<statements> 'run()' '{' <statements> 'Thread' '.' <statements>
<statements> '}' | 'extends' 'Thread' <statements> 'run()' '{'
<statements> 'Thread' '.' <statements> '}' | 'implements' 'Runnable'
<statements> 'run()' '{'<statements> '}' | 'implements' 'Runnable'
<statements> 'run()' '{' 'Thread' '.' <statements> <statements> '}' |
'implements' 'Runnable' <statements> 'run()' '{' <statements> 'Thread'
'.' <statements> <statements> '}' | 'implements' 'Runnable'
<statements> 'run()' '{' <statements> 'Thread' '.' <statements> '}' |
'new' 'Thread' '(' <statements> ')'
```