PEDRO AUGUSTO FELIPE MACHADO GAZOLLA

MECANISMO VISUAL BASEADO EM ASPECTOS PARA AUTOMATIZAÇÃO DE LOGGING

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS - BRASIL
2008

Ficha catalográfica preparada pela Seção de Catalogação e Classificação da Biblioteca Central da UFV

T
G291m
2008

Gazolla, Pedro Augusto Felipe Machado, 1983-
Mecanismo visual baseado em aspectos para automatização de logging / Pedro Augusto Felipe Machado Gazolla. – Viçosa, MG, 2008.
x, 84f. : il. (algumas col.) ; 29cm.

Orientador: Vladimir Oliveira Di Iorio.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f. 81-84.

1. Linguagem de programação (Computador). 2. Programação (Computador). 3. Programação orientada a aspectos (Computação). 4. Programação visual (Computação). 5. Java (Linguagem de programação de computador). 6. Programas de computador - Testes. I. Universidade Federal de Viçosa. II.Título.

CDD 22.ed. 005.13

PEDRO AUGUSTO FELIPE MACHADO GAZOLLA

MECANISMO VISUAL BASEADO EM ASPECTOS PARA AUTOMATIZAÇÃO DE LOGGING

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADA: 3 de Abril de 2008.

Prof. Alcione de Paiva Oliveira (Co-Orientador)	**Prof. José Luís Braga (Co-Orientador)**
Prof. Jugurta Lisboa Filho	**Prof. Antônio M. Pereira de Resende**

Prof. Vladimir Oliveira Di Iorio (Orientador)

AGRADECIMENTOS

Agradeço primeiramente a Deus, que em todos os momentos guiou meus passos e iluminou minha vida, me dando a força e sabedoria necessária para vencer cada novo desafio.

Aos meus pais, Afrânio e Marisa, por todo amor e ensinamentos que sempre me deram. Sem apoio de vocês, com certeza, essa caminhada não teria sido possível. Obrigado por fazerem dos meus sonhos, seus sonhos, e da minha vida, suas vidas.

À minha irmã caçula, Patrícia, que mesmo a distância, trouxe incentivo, apoio e colaboração.

Ao meu irmão, João, que durante todos esses anos em Viçosa, tive o prazer de sua companhia, principalmente por estarmos distantes do restante da família. Obrigado pelo companheirismo e por sempre me ajudar de todas as formas possíveis.

Ao meu grande amor, Elenice, que me faz a cada dia a pessoa mais feliz do mundo. O seu carinho, paciência e cumplicidade foram fundamentais.

À Universidade Federal de Viçosa, especialmente ao Departamento de Informática, pela oportunidade de realizar este trabalho.

Em especial, ao meu orientador, Professor Vladimir Oliveira Di Iorio, pelos ensinamentos, paciência, atenção e disponibilidade em sempre ajudar. O seu apoio e calma foram essenciais para eu subir mais esse degrau.

Aos professores do DPI, que contribuíram com a minha formação acadêmica e amadurecimento pessoal. Aos funcionários, especialmente, ao Altino, com quem sempre tive a certeza de poder contar.

Aos colegas de mestrado, os agradáveis momentos de convivência, pelo apoio, incentivo e troca de idéias. Em especial, ao Estevão e Bernardo pela paciência, amizade e camaradagem.

A todos que, de algum modo, contribuíram para a conclusão dessa etapa da minha vida. Tenho consciência de que, se cheguei até aqui, é porque tive a oportunidade crescer a aprender ao lado de pessoas muito importantes.

BIOGRAFIA

PEDRO AUGUSTO FELIPE MACHADO GAZOLLA, filho de Afrânio Gonçalves Gazolla e Marisa Felipe Machado Gazolla, nasceu em 13 de Julho de 1983, em São Luís, Maranhão.

Em 2000, concluiu o 2º Grau no Colégio Pitágoras, em São Luís - MA.

Em 2001, iniciou o curso de Ciência da Computação na Universidade Federal do Maranhão, onde permaneceu até o fim do terceiro período. No segundo semestre de 2002, transferiu-se para a Universidade Federal de Viçosa (MG) e deu continuidade ao curso de Ciência da Computação, que foi concluído no final do ano de 2005. Durante a graduação, teve a oportunidade de realizar estágio no CPD/UFV e de desenvolver trabalhos de iniciação científica por um ano.

Em Maio de 2006, ingressou no Curso de Mestrado do Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Viçosa, atuando na linha de pesquisa "Linguagens de Programação". Nesse tempo, trabalhou por um ano e meio como Analista de Sistemas em uma empresa da cidade. Em abril de 2008, submeteu-se à defesa desta dissertação.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE LISTAGENS

LISTA DE QUADROS

RESUMO

GAZOLLA, Pedro Augusto Felipe Machado, M.Sc., Universidade Federal de Viçosa, abril de 2008. **Mecanismo visual baseado em aspectos para automatização de logging.** Orientador: Vladimir Oliveira Di Iorio. Co-Orientadores: Alcione de Paiva Oliveira e José Luís Braga.

A Programação Orientada a Objetos (POO) é um paradigma de programação capaz de modularizar interesses de negócio, porém não permite gerenciar eficientemente interesses transversais, o que gera problemas de entrelaçamento e espalhamento de código. Novas abordagens surgiram propondo soluções para esse tipo de problema, dentre as quais a Programação Orientada a Aspectos (POA) se destacou. Entretanto, as linguagens orientadas a aspectos possuem deficiências que dificultam uma maior popularização: sintaxe e conceitos complexos. Com o objetivo de abstrair a complexidade dessas linguagens, surgiram alguns trabalhos que utilizam a POA para resolver problemas ligados a um interesse transversal específico por meio de recursos visuais. Foi verificada uma carência de trabalhos que tratam do requisito de *logging*. Dessa forma, o objetivo deste trabalho é analisar o uso de técnicas de programação orientada a aspectos na modularização do interesse transversal de *logging* a fim de desenvolver um mecanismo visual para a geração de aspectos relacionados a esse interesse, de forma interativa e transparente. Para se alcançar esse objetivo, passou-se pelas seguintes etapas: definição de um conjunto de características relacionadas a *logging*; idealização de um mecanismo capaz de oferecer essas funcionalidades; escolha de um ambiente de desenvolvimento para a integração do mecanismo; realização do projeto e construção do mecanismo. O resultado é uma solução que abstrai a complexidade de utilizar uma abordagem orientada a aspectos no tratamento de *logging*, melhorando o gerenciamento desse interesse transversal.

ABSTRACT

GAZOLLA, Pedro Augusto Felipe Machado, M.Sc., Universidade Federal de Viçosa, April, 2008. **Visual mechanism based on aspects for logging automation.** Adviser: Vladimir Oliveira Di Iorio. Co-Advisers: Alcione de Paiva Oliveira and José Luís Braga.

Object Oriented Programming (OOP) is a programming paradigm well suited for modularizing business concerns, but it does not allow an efficient management of crosscutting concerns, what generates problems of tangling and scattering of code. Several approaches have proposed solutions for these problems. Among them, the Aspect Oriented Programming (AOP) is distinguished. However, aspect oriented languages usually have complex syntax and complex concepts which make them difficult to become popular. Some works have tried to overcome this complexity by presenting visual tools which help to define simpler solutions for specific crosscutting concerns. Logging is a concern that apparently was not well explored. The objective of this research is to analyze the use of techniques of aspect oriented programming to modularize the logging crosscutting concern in order to develop a visual mechanism for the generation of aspects related to this concern, in an interactive and transparent way. To achieve this objective, this work has followed these steps: definition of a set of features related to logging; proposal of a mechanism capable of offering these features; choice of a development environment for the integration of the mechanism; design and construction of the mechanism. The result is a solution that abstracts the complexity of using an aspect-oriented approach when dealing with logging, improving the management of this crosscutting concern.

1 INTRODUÇÃO

A engenharia de software é uma área em constante evolução. Cada vez mais, os problemas são tratados com um nível mais alto de abstração. A preocupação com instruções de máquina é passado. Hoje, o objetivo é ver um sistema como uma associação de objetos que colaboram entre si (LADDAD, 2003). A este paradigma de desenvolvimento de software dá-se o nome de Programação Orientada a Objetos (POO), que segundo Filman *et al.* (2005), atualmente, é o paradigma dominante.

É notável o avanço proporcionado pela POO. A partir dela, os problemas passaram a ser vistos de uma forma mais natural. Segundo Filman *et al*. (2005), o programador passa a pensar no universo como um conjunto de instâncias de classes particulares que provêem métodos para descrever o comportamento de todos os seus objetos. A interação entre estes objetos forma um sistema, que Gradeck e Lesiecki (2003) definem como uma coleção de módulos (classes) que trabalham em conjunto para fornecer as funcionalidades desejadas a partir de um conjunto de requisitos. A modularização é discutida pela primeira vez em Parnas (1972) como um mecanismo para atingir maiores níveis de manutenibilidade, evolução e reúso.

O princípio da separação de interesses (*separation of concerns*) busca facilitar a compreensão do desenvolvedor, que dividindo o problema em partes menores (módulos), concentra-se em cada funcionalidade do sistema por vez (DIJKSTRA, 1976). Um interesse (*concern*) pode ser uma característica qualquer do problema com responsabilidades bem-definidas. Nessa perspectiva, Gradeck e Lesiecki (2003) conceituam um interesse como alguma funcionalidade ou requisito do sistema, o qual deve ser codificado. Indo além, Laddad (2003) classifica os interesses em duas categorias: interesses do negócio (*core concerns*), que capturam a funcionalidade central de um módulo, e interesses transversais (*crosscutting concerns*), que capturam requisitos periféricos do sistema, atravessando vários módulos. São exemplos de interesses transversais*:* autenticação, segurança, desempenho, *logging*, integridade de transações, distribuição, persistência e *profiling*.

Porém, devido à natureza da POO, o ideal de modularização é raramente realizado, sendo executado parcialmente. Segundo Laddad (2003), isso acontece pelo

fato de existir uma lacuna entre o entendimento dos requisitos de um sistema e a forma como são implementados. O autor declara que esta lacuna acontece devido à existência de interesses transversais. Isso acarreta dois problemas no código:

- O **entrelaçamento (*tangling*)** acontece quando um mesmo módulo do sistema contém mais de um interesse, em geral, um interesse do negócio e um ou mais interesses transversais. A Figura 1.1 ilustra esse problema.

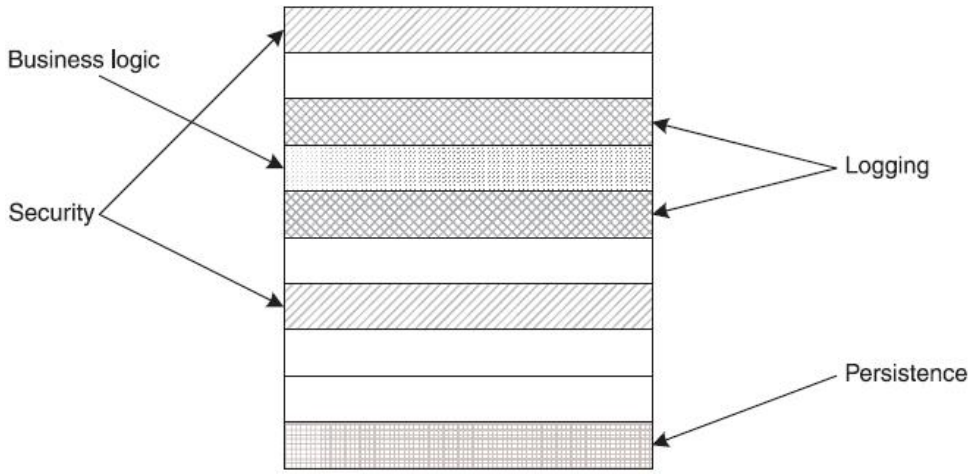

Figura 1.1 - Código entrelaçado.
Fonte: Extraída de Laddad (2003).

- O **espalhamento (*scattering*)** diz respeito à presença de um mesmo interesse transversal por vários módulos do sistema, como mostrado na Figura 1.2.

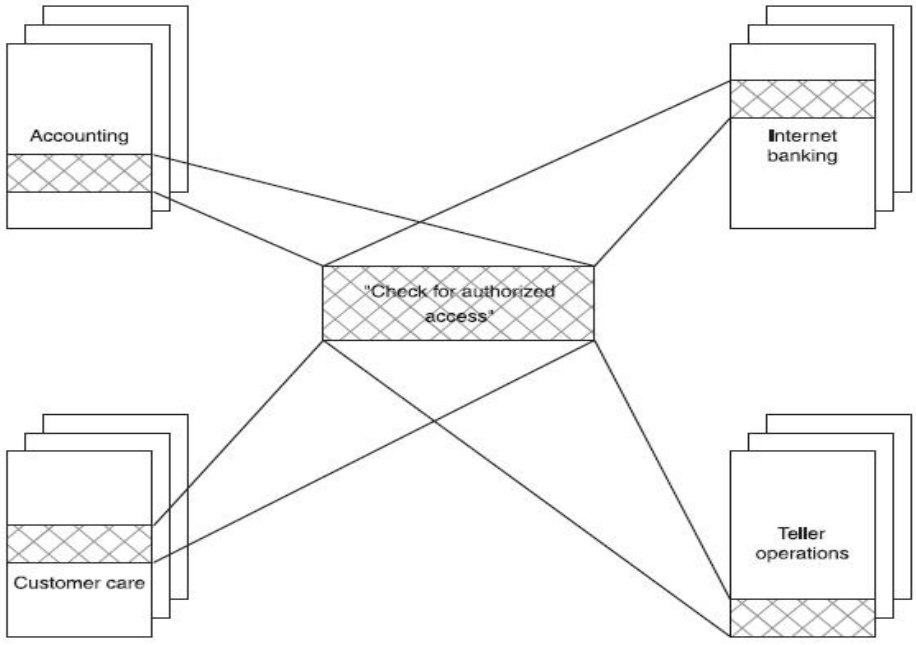

Figura 1.2 - Espalhamento de código.
Fonte: Extraída de Laddad (2003).

Filman *et al*. (2005) afirmam que o progresso das linguagens de programação e das metodologias de projeto é baseado na invenção de estruturas que provêem níveis adicionais de modularidade. Nessa perspectiva, a mais nova e promissora evolução se chama Programação Orientada a Aspectos (POA) (KICZALES, 1997). Laddad (2003) afirma que a POA é uma metodologia capaz de permitir a separação de interesses transversais através de uma nova unidade de modularização – o aspecto (*aspect*) - que entrecorta outros módulos. Com a POA, interesses transversais são implementados em aspectos, ao invés de ficarem espalhados dentro dos outros módulos.

O registro de ações executadas por um sistema - o *logging* – para fins de possíveis análises posteriores é um clássico exemplo de interesse transversal tratado elegantemente pela POA e que faz parte da maioria dos sistemas. Uma estratégia eficiente de *logging* possibilita economizar horas, quando se está diagnosticando um problema ocorrido em um sistema.

Assim como a POA, outras abordagens surgiram para fornecer um nível avançado de separação de interesses, contornando os problemas de entrelaçamento e espalhamento: Separação Multidimensional de Interesses (OSSHER e TARR, 2001), Programação Adaptativa (LIEBERHERR, 1995), Filtros de Composição (AKSIT *et al*., 1993) e Programação Orientada a Sujeitos (HARRISON e OSSHER, 1993). Contudo, estas abordagens não obtiveram o mesmo destaque que a POA.

Em uma visão evolucionária das linguagens de programação, Laddad (2003) comenta que a Programação Procedural (PP) introduziu a abstração funcional, a POO introduziu a abstração de objetos, e agora, a POA introduziu a abstração de interesses. Apesar do seu caráter evolucionário, a POA não substitui a POO, e sim, trabalha em conjunto com ela. As classes continuam responsáveis por conter os interesses do negócio, porém, sem se preocupar com os interesses transversais, que passam a ser modularizados pelos aspectos.

Por fim, a POA não resolve nenhum problema novo no ponto de vista computacional, assim como a POO também não. Entretanto, os problemas passam a ser resolvidos de uma forma melhor e mais simples, resultando em ganhos de qualidade e tempo no ciclo de desenvolvimento. Segundo Laddad (2003), a POA significa, definitivamente, o começo de uma nova forma de ver um sistema, uma composição de interesses independentes.

1.1 O problema e sua importância

Os trabalhos pioneiros de Parnas (1972) e Dijkstra (1976) propuseram a separação de interesses – estágio avançado da modularização - de um sistema como uma forma de facilitar a compreensão humana. Quanto maior o nível de abstração, mais fácil se torna esta compreensão.

Apesar de a POO possibilitar a abstração dos interesses de negócio de forma eficiente, ela falha por não suportar a modularização de interesses transversais, acarretando em entrelaçamento e espalhamento de código. Entretanto, utilizando-se a POA, esses problemas são resolvidos. De uma maneira geral, a POA traz vários benefícios ao sistema, tais como: redução do acoplamento, aumento da coesão, maiores níveis de reutilização, mais qualidade, diminuição do tempo de desenvolvimento e ganhos na manutenibilidade.

Dentre as linguagens orientadas a aspectos, a de maior destaque é AspecJ (ASPECTJ, 2008; KICZALES, 2001). Ela é uma extensão de Java (JAVA, 2008), a linguagem orientada a objetos mais popular no mundo. Vários ambientes de desenvolvimento suportam essas linguagens. Qualquer requisito que se encaixe na categoria de interesse transversal é candidato a ser implementado através de uma linguagem orientada a aspectos, como AspectJ.

Logging é um interesse transversal presente na grande maioria dos sistemas. Existem boas bibliotecas de *logging* para Java, como a Java Logging (JAVA LOGGING, 2008) introduzida no Java Developer Kit (JDK) da versão Java 1.4, e Log4J (LOG4J, 2008) da Apache. Porém, mesmo usando essas bibliotecas, o desenvolvedor necessita escrever trechos de código de *logging* que se misturam com o restante dos códigos das classes, o que contraria a busca do ideal de modularização. Portanto, a POA é uma excelente opção no tratamento de *logging*.

Contudo, devido às dificuldades de se aprender uma nova técnica de programação, neste caso, uma linguagem orientada a aspectos, soluções para *logging* utilizando a POA são pouco difundidas. Uma das dificuldades encontra-se nos problemas em lidar com conceitos e sintaxes de uma nova linguagem, que possui um nível alto de dificuldade e que muitas vezes é vista como desafiadora pelos desenvolvedores, ou seja, tem uma longa curva de aprendizado. Nas empresas, existem outros empecilhos, tais como: os gastos com treinamento de pessoal, a

resistência às inovações e a falta de tempo para aprender uma filosofia diferente de desenvolvimento.

Para contornar esses problemas e usufruir dos benefícios que uma linguagem orientada a aspectos proporciona no tratamento do interesse transversal de *logging*, uma alternativa seria permitir ao desenvolvedor usá-la de forma transparente e simples na modularização desse interesse. Em outras palavras, não seria necessário ter nenhum conhecimento técnico da linguagem para utilizá-la. Nesse contexto, surge a seguinte questão: como abstrair o uso da programação orientada a aspectos, pelo desenvolvedor, no gerenciamento do requisito transversal de *logging*?

Tornando isso possível, seriam alcançados ganhos de produtividade no ciclo de desenvolvimento e de qualidade no sistema. Isso, sem a necessidade de se dominar uma linguagem orientada a aspectos, que é um dos empecilhos que impedem a popularização da POA.

1.2 Objetivos

O objetivo geral deste trabalho foi analisar o uso de técnicas de programação orientada a aspectos na modularização do interesse transversal de *logging* a fim de desenvolver um mecanismo visual para a geração de aspectos relacionados a esse interesse, de forma interativa e transparente.

Especificamente, os objetivos foram:

1. Definir um conjunto de funcionalidades importantes relacionadas a *logging*, que a linguagem orientada a aspectos AspectJ consiga tratar;
2. Idealizar um mecanismo capaz de oferecer essas funcionalidades de *logging* através de configurações visuais, que resultassem na geração automática de aspectos;
3. Investigar e escolher um ambiente de desenvolvimento que suporte esse mecanismo, além de facilitar a sua adoção;
4. Projetar o mecanismo e a sua integração ao ambiente de desenvolvimento escolhido, considerando a sua boa usabilidade e interatividade;
5. Construir o mecanismo proposto.

1.3 Estrutura da Dissertação

Na Seção 2 é apresentado o referencial teórico, que é resultado da pesquisa bibliográfica sobre as áreas e tecnologias relacionadas ao trabalho. Essa seção é subdividida em:

- Introdução sobre a programação orientada a aspectos;
- Apresentação dos conceitos e sintaxes da linguagem orientada a aspectos AspectJ;
- Definição de *logging*, discussão sobre a sua importância e de sua relação com a programação orientada a aspectos;
- Apresentação da biblioteca de *logging* Log4J para a linguagem Java, explicando seus conceitos e sintaxes;
- Visão geral sobre o ambiente de desenvolvimento Eclipse e de sua extensão utilizada para o desenvolvimento de *plugins* (PDE);
- Apresentação do JAXB, uma API Java para o mapeamento objeto-XML;
- Discussão sobre os principais trabalhos que influenciaram na realização da pesquisa.

Na Seção 3 são descritos os principais métodos e soluções adotadas durante o desenvolvimento desta dissertação, baseando-se na ordem cronológica de realização de cada um dos objetivos específicos (Subseção 1.2);

Na Seção 4 é apresentado o mecanismo visual desenvolvido para simplificar o gerenciamento do interesse transversal de *logging* através da geração automática de aspectos. As funcionalidades oferecidas por cada um de seus submecanismos são descritas, assim como são mostradas as suas respectivas interfaces gráficas. Também é apresentada a estrutura dos aspectos e das classes criadas automaticamente pelo gerador de código.

Na Seção 5 são apresentadas as conclusões desta dissertação, assim como as principais contribuições oferecidas por este trabalho. Também são enumeradas possíveis melhorias e possibilidades de trabalhos futuros que poderão contribuir com essa linha de pesquisa.

2 REFERENCIAL TEÓRICO

Esta seção tem por objetivo introduzir os principais temas relacionados a este trabalho, e para facilitar o entendimento, ela é dividida em partes.

Como a POA está no centro do trabalho, ela é a primeira a ser apresentada, seguida pela linguagem orientada a aspecto AspectJ.

Depois, é feita uma discussão sobre *Logging*, área que está estritamente relacionada ao problema resolvido, e é apresentada a API Java Log4J para *logging*.

Na seqüência, descreve-se a parte que consiste na apresentação do ambiente de desenvolvimento Eclipse, onde o mecanismo desenvolvido foi integrado como um *plugin*. Para controlar a persistência desse mecanismo, foi utilizada a API Java JAXB, que também é apresentada.

Na última parte, foram descritos os principais trabalhos que contribuíram com idéias para conceber e planejar o mecanismo desenvolvido.

2.1 Programação Orientada a Aspectos

> AOP is a new evolution in the line of technology for separation of concerns - technology that allows design and code to be structured to reflect the way developers want to think about the system. (KICZALES, 2001, p. 1)

O objetivo da Programação Orientada a Aspectos (POA) (KICZALES *et al.*, 1997) é tornar o projeto e o código de sistemas mais modulares, permitindo que seus vários interesses fiquem localizados, ao invés de entrelaçados e espalhados, e que tenham relações de interface bem definidas com o restante do sistema (ELRAD *et al.*, 2001).

Kiczales *et al.* (1997) definiram dois importantes termos básicos: o componente, que encapsula as propriedades do sistema através de uma abordagem estruturada ou orientada a objetos, e o aspecto, que permite encapsular, por meio de uma abordagem orientada a aspectos, propriedades que essas abordagens não são capazes de tratar.

Um aspecto é responsável por modularizar os interesses transversais de um sistema. Essa modularização é possível, na filosofia orientada a aspectos, porque as estruturas responsáveis por encapsular os interesses do negócio - procedimentos ou objetos – passam a não ter consciência da existência destes interesses transversais. Em contrapartida, os aspectos precisam saber os pontos onde os interesses transversais devem se relacionar com os interesses de negócio. Ou seja, ocorre uma inversão de controle. A Figura 2.1 ilustra a estrutura de um típico sistema orientado a objetos. Já na Figura 2.2, o interesse transversal é modularizado em um aspecto, onde se pode perceber a existência da inversão de controle.

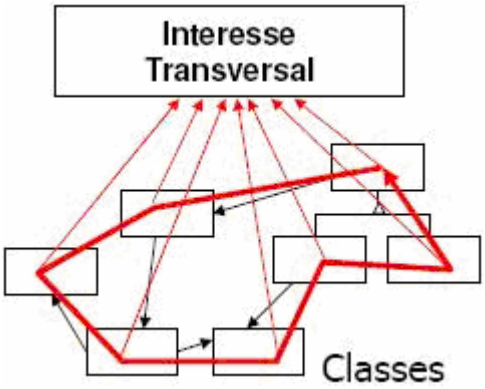

Figura 2.1 - Sistema orientado a objetos.
Fonte: Extraída de Kulesza *et al*. (2005).

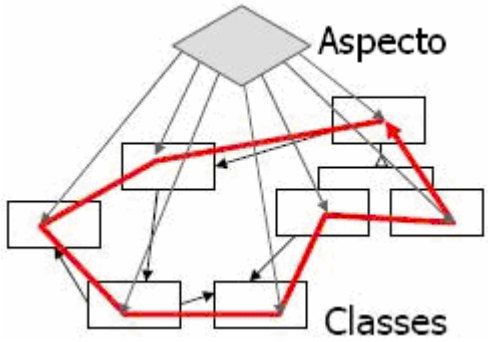

Figura 2.2 - Sistema orientado a aspectos.
Fonte: Extraída de Kulesza *et al*. (2005).

A POA envolve basicamente três etapas distintas de desenvolvimento, que podem ser vistas na Figura 2.3, e as quais Laddad (2003) descreve como:

- **Decomposição Aspectual (*Aspectual Decomposition*)**: nessa etapa, os interesses do sistema são decompostos em interesses de negócio, que

normalmente são requisitos funcionais, e interesses transversais, geralmente requisitos não-funcionais do problema.

- **Implementação dos interesses (*Concern Implementation*)**: nesse ponto, cada interesse deve ser implementado separadamente. Por exemplo, as classes seriam responsáveis por modularizar os interesses de negócio e os aspectos pelos interesses transversais.
- **Recomposição Aspectual (*Aspectual Recomposition*):** nessa etapa, as regras de recomposição devem ser especificadas nos aspectos. O processo de recomposição (*weaving*) se utiliza dessas regras para compor o sistema final.

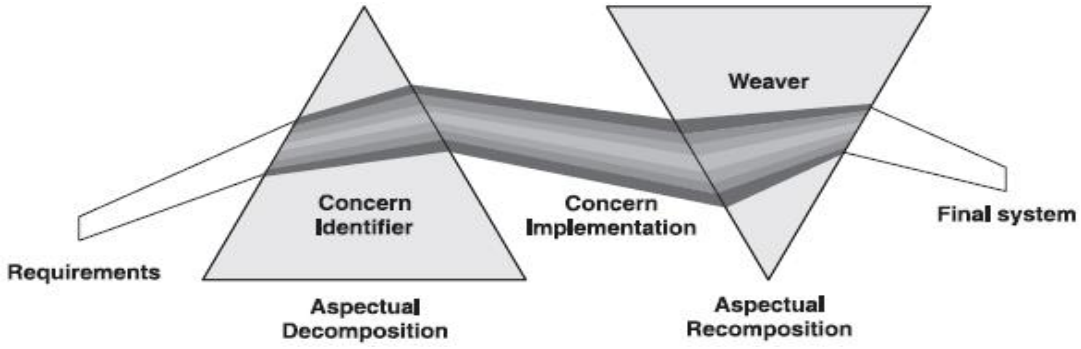

Figura 2.3 - Etapas do desenvolvimento de software orientado a aspectos.
Fonte: Extraída de Laddad (2003).

O que tem sido visto nos últimos anos leva a crer que a POA está, de fato, melhorando a metodologia orientada a objetos através de uma forte separação de interesses. Tanto que, o desenvolvimento de software baseado em aspectos se reflete em vários benefícios para o sistema e para o ciclo de desenvolvimento: cada módulo tem suas responsabilidades bem definidas; alto nível de modularização; facilidade de evolução do sistema; amarração tardia de implementação de requisitos; maior reutilização de código; menor tempo de desenvolvimento; e menores custos (GRADECK E LESIECKI, 2003; FILMAN *et al*., 2005; LADDAD, 2003).

Por conta das boas perspectivas para a POA, surgiram várias linguagens orientadas a aspectos, onde se pode destacar: AspectJ (KICZALES *et al.*, 2001), HyperJ (OSSHER e TARR, 2000), AspectC++ (ASPECTC++, 2008) e AspectC (COADY *et al*., 2001). AspectJ é descrita na Subseção 2.2. HyperJ oferece suporte à metodologia de separação multi-dimensional e integração de requisitos em Java. Os requisitos são implementados em dimensões distintas e o ambiente de compilação é responsável pela composição deles. AspectC++ é um projeto inspirado em AspectJ

que oferece uma implementação orientada a aspectos para C++. AspectC é uma extensão orientada a aspectos para a linguagem C (TIRELO *et al*., 2004).

2.2 AspectJ

A linguagem AspectJ (KICZALES *et al.*, 2001; GRADECK e LESIECKI, 2003; LADDAD, 2003; TIRELO *et al*., 2004; FILMAN *et al.*, 2005) é uma extensão orientada a aspecto da linguagem Java desenvolvida por um grupo de pesquisadores no Xerox Palo Alto Research Center. Kiczales *et al.* (2001) afirmam que a decisão de tornar AspectJ uma extensão compatível com Java facilitaria a sua adoção pelos próprios programadores Java. Para torná-la compatível, eles traçaram quatro metas:

- **Compatibilidade Extensiva (*Upward Compatibility*)**: todos os programas em Java deveriam ser programas em AspectJ.
- **Compatibilidade de Plataforma (*Platform Compatibility*)**: todos os programas em AspectJ deveriam executar na máquina virtual Java.
- **Compatibilidade de Ferramentas (*Tool Compatibility*)**: deveria ser possível estender as ferramentas existentes para suportar AspectJ de uma forma natural.
- **Compatibilidade de Programador (*Programmer Compatibility*)**: programar em AspectJ deveria ser como programar em uma extensão natural de Java.

AspectJ suporta dois mecanismos de implementação transversal: o de transversalidade dinâmica (*dynamic crosscutting*) e o de transversalidade estática (*static crosscutting*). A transversalidade dinâmica permite definir uma implementação adicional para rodar em pontos bem definidos na execução de um programa, como por exemplo, pode-se fazer com que uma determinada ação seja executada antes da execução de um método, incrementando seu fluxo de execução. Já a transversalidade estática, possibilita adicionar novas operações e membros sobre as estruturas de um programa, além de permitir a alteração da hierarquia de classes (ELRAD *et al.*, 2001).

Respeitando a meta de Compatibilidade de Programador, AspectJ oferece os mesmos recursos de Java, tais como: classes, métodos, atributos e outros. Adicionalmente, AspectJ tem o objetivo de prover uma boa integração entre os interesses de negócio e os interesses transversais de um sistema. Para isso, são fornecidos novos conceitos e construções. A seguir, apenas os principais recursos de

AspectJ utilizados neste trabalho serão apresentados segundo a definição de Gradeck e Lesiecki (2003):

- **Ponto de Junção (*Join Point*)**: é um ponto bem definido na execução de um programa. AspectJ define uma grande variedade de construções usadas para caracterizar os pontos de junção, entre os quais:
  - **`call`**: usado para definir chamadas de métodos e construtores. Por exemplo, a expressão `call(public int Aluno.getMatricula())` identifica os pontos de junção que representam a chamada do método de assinatura `public int Aluno.getMatricula()`;
  - **`execution`**: usado para definir execuções de métodos e construtores. A expressão `execution(void Aluno.set*(..))` caracteriza os pontos de junção que representam a execução de um método da classe `Aluno` que inicie com o nome "set" e não retorne nenhum valor, independente dos seus parâmetros, já que o elemento '`..`' representa qualquer combinação de parâmetros. O elemento '`*`' tem a função de curinga (*wildcard*) e pode representar qualquer coisa, e se fosse necessário, na expressão, poderia ser utilizado no lugar do `void` para indicar qualquer tipo de retorno;
  - **`get`**: usado para definir quando o membro de uma classe é lido. Por exemplo, a expressão `get(public String Aluno.nome)` define os pontos de junção que representam a leitura do membro `nome` da classe `Aluno`;
  - **`set`**: usado para definir quando o membro de uma classe tem seu valor alterado. Por exemplo, a expressão `set(public int Aluno.matricula)` define os pontos de junção que representam a alteração do valor do membro `matricula` da classe `Aluno`;
  - **`cflow`**: retorna os pontos de junção presentes na execução de outro ponto de junção.
- **Objetos com Reflexão**: quando um ponto de junção é acionado, pode-se ter acesso às informações do seu contexto a partir dos objetos `thisJoinPointStaticPart`, `thisEnclosingJoinPointStaticPart` e `thisJoinPoint`;

- **Conjunto de Junção (*Pointcut*)**: é um conjunto de pontos de junção. Além disso, permite obter informações do contexto desses pontos. Operadores lógicos podem ser usados na sua formação: && (e), || (ou) e ! (negação). Por exemplo, a expressão `pointcut: call(* Aluno.get*()) || call(void Aluno.set*(..)` define um conjunto de junção que agrupa os pontos de junção da classe `Aluno` que representam as chamadas de métodos que se iniciam com o nome "get" ou "set";
- **Adendo (*Advice*)**: é uma ação a ser executada quando um ponto de junção definido em um conjunto de junção é atingido. AspectJ permite que essas ações sejam executadas em três momentos: antes (`before`), depois (`after`) e durante (`around`). As variações existentes são:
  - **`before`**: determina que uma ação seja realizada antes do ponto de junção;
  - **`after`**: determina que uma ação seja realizada depois do ponto de junção;
  - **`after returning`**: determina que uma ação seja realizada depois do ponto de junção que tenha sido executado com sucesso;
  - **`after throwing`**: determina que uma ação seja realizada depois do ponto de junção que tenha sido executado com falha, ou seja, quando é lançada exceção;
  - **`around`**: esse é o tipo de adendo mais poderoso, pois permite alterar completamente o fluxo de execução de um ponto de junção. Isto é, existe liberdade total de manipulação. A execução do ponto de junção é representada pelo elemento de sintaxe `proceed()`. Podem-se criar ações para serem executadas antes e depois, além de desvios no fluxo de execução.
- **Aspecto (*Aspect*)**: é a unidade modular utilizada para a implementação de requisitos transversais. Sua estrutura é semelhante à de uma classe. Ele encapsula pontos de junção, conjuntos de junção e adendos. Além disso, pode conter seus próprios membros e métodos. Assim como as classes, os aspectos podem ter visibilidade e serem abstratos, porém jamais poderão ser instanciados.

Elrad *et al*. (2001) comentam que a reutilização em aspectos é uma questão chave. Por isso, AspectJ possui um mecanismo de reúso similar aos encontrados na POO, onde aspectos podem ser estendidos ou especializados, formando uma hierarquia de aspectos. Por exemplo, o programador pode definir um aspecto abstrato com um ou mais conjuntos de junção abstratos que podem ser sobrescritos e alguns adendos que podem ser herdados pelo aspecto concreto. Dessa forma, suporta-se a criação de bibliotecas de aspectos.

Com o objetivo de ilustrar os recursos da linguagem AspectJ já explicados, é mostrado na Listagem 2.2 o exemplo de um aspecto chamado `AlunoAspect` que é responsável por registrar as operações realizadas na classe `Aluno`, mostrada na Listagem 2.1.

```
1  public class Aluno {
2    public String nome;
3    public int matricula;
4
5    public Aluno(String nome, int matricula) {
6      this.nome = nome;
7      this.matricula = matricula;
8    }
9
10   public String getNome() {..
13   public void setNome(String nome) {..
16   public int getMatricula() {..
19   public void setMatricula(int matricula) {..
22 }
```

Listagem 2.1 - Aluno.java
Fonte: Elaborada pelo autor.

```
1  public aspect AlunoAspect {
2    pointcut callGetMatrMethod(): call(public int Aluno.getMatricula());
3    pointcut executionSetMethods(): execution(void Aluno.set*(..));
4    pointcut setMemberMatricula(): set(public int Aluno.matricula);
5    pointcut getMemberNome(): get(public String Aluno.nome);
6
7    before() : callGetMatrMethod() {
8      System.out.println("Antes da chamada do método " +
9          thisJoinPoint.getSignature().getName());
10   }
11   after() : executionSetMethods() {
12     System.out.println("Depois da execução do método " +
13         thisJoinPoint.getSignature().getName());
14   }
15   after() returning() : setMemberMatricula() {
16     System.out.println("Alteração de valor do membro " +
17         thisJoinPoint.getSignature().getName());
18   }
19   String around() : getMemberNome() {
20     System.out.println("Antes da leitura do nome do aluno.");
21     String nome = proceed();
22     System.out.println("O nome desse aluno é " + nome);
23     return nome;
24   }
25 }
```

Listagem 2.2 - AlunoAspect.aj
Fonte: Elaborada pelo autor.

Nesse aspecto, são definidos quatro conjuntos de junção. O `callGetMatrMethod` (linha 2) caracteriza as chamadas ao método `getMatricula`. Já o `executionSetMethods` (linha 3) representa as execuções dos métodos cujos nomes iniciam com "set". Enquanto isso, o `setMemberMatricula` (linha 4) caracteriza as alterações de valor do membro `matricula`. Por fim, o `getMemberNome` (linha 5) representa os pontos onde há leitura do membro `nome`.

Foram também criados quatro adendos. O adendo `before` (linhas 7 a 10) gera uma mensagem antes da chamada de um dos pontos de junção definido pelo conjunto de junção `callGetMatrMethod`. Já o adendo `after` (linhas 11 a 14) registra uma mensagem depois dos pontos definidos por `executionSetMethods`. Enquanto isso, o adendo `after returning` (linhas 15 a 18) cria uma mensagem depois que os pontos definidos por `setMemberMatricula` são executados com sucesso. Por último, o adendo `around` (linhas 19 a 25) monitora através de mensagens, antes e depois, os pontos definidos por `getMemberNome`.

Para ilustrar o resultado da transversalidade inserida pelo aspecto `AlunoAspect` sobre a classe `Aluno`, é mostrada na Listagem 2.3 uma classe `Teste`, responsável por executar operações sobre um objeto do tipo `Aluno`.

```
1 public class Teste {
2   public static void main(String[] args) {
3     Aluno aluno = new Aluno("Paulo", 50000);
4     aluno.setMatricula(50001);
5     aluno.getMatricula();
6     aluno.setNome("Pedro");
7     aluno.getNome();
8   }
9 }
```

Listagem 2.3 - Teste.java
Fonte: Elaborada pelo autor.

O resultado da execução dessa classe seria:

```
Alteração de valor do membro matricula
Alteração de valor do membro matricula
Depois da execução do método setMatricula
Antes da chamada do método getMatricula
Depois da execução do método setNome
Antes da leitura do nome do aluno
O nome desse aluno é Pedro
```

2.3 Logging

Até mesmo os melhores sistemas sofrem problemas - os *bugs* – que comprometem o seu funcionamento desejado. Para resolvê-los, é necessário obter informações que ajudem a detectar a causa do problema o mais rápido possível, o que muitas vezes é mais demorado do que reparar o próprio problema (GULCU, 2004). O processo de armazenamento das ações executadas por um sistema é chamado de *logging*. Em geral, não se sabe se as informações serão usadas, porém, caso algum problema seja detectado, elas podem ser consultadas.

Laddad (2003) define *logging* como uma técnica para entender o funcionamento do sistema, que na sua forma mais simples, registra mensagens descrevendo as operações realizadas. Já Gupta (2003) descreve o *logging* como uma técnica sistemática e controlada para representar o estado de uma aplicação de uma forma legível ao homem. Por exemplo, em um sistema bancário, cada transação sobre uma conta pode ser registrada com informações da natureza da transação, do número da conta e do valor envolvido. Outro exemplo seria a utilização do *logging* para colher informações que ajudem a analisar o desempenho de uma aplicação.

Quando um sistema está em ambiente de produção e algum problema ocorre, uma forma de detectá-lo é conversando com o usuário para que ele possa descrever a situação que levou àquilo. Porém, normalmente o usuário não se lembra do contexto em que ocorreu a falha, logo essa não é uma boa solução. A melhor maneira de rastrear um problema é através de uma estratégia eficiente de *logging*. Caso contrário, pode-se levar dias para diagnosticar um problema.

Nessa perspectiva, Gupta (2003) afirma que toda aplicação comercial necessita de *logging*, caso contrário, essa aplicação será difícil de depurar e perderá valor de mercado. Ele ainda comenta a importância do *logging* na qualidade do código produzido e no aumento da manutenibilidade da aplicação.

Algumas decisões são necessárias para se atingir uma estratégia eficiente de *logging*, e requer, por parte do projetista, planejamento e disciplina. A tecnologia a ser utilizada pode facilitar bastante esse processo, onde a POA tem se mostrado uma excelente alternativa. Deve-se preocupar também com a natureza do sistema, como por exemplo, aqueles em que o processo de *logging* está distribuído sobre vários processos. O excesso de pontos monitorados, principalmente em grandes sistemas, deve ser evitado para facilitar o manuseio dos registros. Um arquivo de *logging* deve

facilitar a compreensão humana através do registro de mensagens em formato bem informativo e claro (GUPTA, 2003; GULCU, 2004; CHAN, 2005).

Gupta (2003) enumera um lista de benefícios oferecidos pelo uso do *logging*, tais como:

- **Identificação de problemas -** Independente da aplicação, sempre existirá um problema a ser descoberto. Logo, uma boa utilização do *logging* permite diagnosticar problemas mais cedo e com mais precisão.
- **Depuração Rápida –** A criação de trechos de *logging* bem planejados e escritos permite encontrar com precisão a localização de problemas ocorridos durante o desenvolvimento da aplicação, o que simplifica bastante a tarefa de resolvê-lo.
- **Facilidade de Manutenção –** Uma aplicação apoiada em uma boa estratégia de *logging* simplifica a contextualização dos problemas que ocorreram durante a sua execução no ambiente de produção.
- **Histórico –** O *logging* permite armazenar um histórico de informações que poderão vir a ser importantes para o esclarecimento de alguma dúvida ou recuperação de informações referentes ao uso da aplicação.
- **Otimização de Tempo e Custos –** Com base nos benefícios descritos anteriormente, não há dúvidas quanto aos ganhos de tempo e de custos no ciclo de vida de uma aplicação.

Contudo, ele ainda menciona que esses benefícios não vêm sem nenhum custo. Uma estratégia de *logging* eficiente requer um planejamento de antemão para evitar que os registros de *logging* causem maiores problemas de desempenho à aplicação e que suas informações confundam a mente de quem os lê. Ainda assim, é impossível evitar a sobrecarga na geração dos registros, principalmente com as operações de entrada e saída, e a sobrecarga de programação, pois se necessita de mais tempo para codificar os trechos de *logging*.

Um tipo especial de *logging* é denominado *tracing*. No processo de *tracing*, a entrada e/ou a saída de um método selecionado é registrado. Esta técnica é muito útil para o entendimento do sistema durante a fase de desenvolvimento, principalmente quando o depurador (*debugger*) não é uma opção, seja pela velocidade que as atividades ocorrem ou a natureza distribuída das aplicações (LADDAD, 2003). A complexidade e tamanho do sistema também são fatores que inviabilizam a depuração. Porém, a partir da

leitura do registro de *tracing*, é possível restringir o espaço onde ocorreu um problema. Em geral, os registros são apagados após o uso pelo desenvolvedor e as rotinas de *tracing* não fazem parte do sistema final.

O lançamento de exceção também é considerado um evento importante no sistema, portanto, registrar estas ocorrências é normalmente desejável. A esta extensão importante do conceito de *logging*, dá-se o nome de *logging de exceção* (LADDAD, 2003).

Segundo Happel e Schmidt (2007), o *logging* se tornou uma boa prática na engenharia de software e está sendo largamente utilizado. Conseqüentemente, os desenvolvedores começaram a visualizá-lo como um interesse que necessita ser modularizado, o que, por exemplo, facilita alterações no formato de saída das mensagens. Apesar da relativa simplicidade em torno do processo de *logging*, esta técnica está intimamente ligada à qualidade final do sistema, o que justifica a sua importância e adoção em grande parte das aplicações.

2.3.1 *Logging* com POA

Atualmente, a grande maioria dos desenvolvedores implementa a técnica de *logging* da forma clássica, ou seja, usando uma abordagem orientada a objetos. Logo, os trechos de *logging* se misturam com os módulos que deveriam ser responsáveis por conter somente os interesses de negócio, o que gera entrelaçamento e espalhamento de código. Por isso, alterar a estratégia de *logging* significa alterar vários módulos. Laddad (2003) afirma que a POA e AspectJ podem ajudar na modularização do *logging*, retirando a responsabilidade por esse interesse dos módulos de negócio e repassando-a para um aspecto, que a centraliza. Isso é feito de forma consistente e eficiente.

A POA pode ser usada eficientemente para acoplar ao sistema uma funcionalidade de *logging* com o propósito de monitorar pontos da aplicação. Se o monitoramento não for mais necessário, a funcionalidade pode ser "desencaixada", bastando para isso retirar o aspecto do projeto. Os mecanismos de reflexão providos por AspectJ permitem que várias informações sobre o contexto do problema possam ser obtidas através de reflexão, sejam elas estáticas ou dinâmicas, como por exemplo: o nome da classe, o nome do método, os valores e tipos dos parâmetros, o valor e tipo do retorno, a linha onde ocorreu o evento, dentre outras.

2.4 Log4j

Log4j (GULCU, 2004; LOG4J, 2008) é um projeto de código aberto (*open source*) desenvolvido pelo projeto Jakarta da Apache, que consiste em uma API Java que possibilita a criação de códigos de *logging* para supervisionar o comportamento de uma dada aplicação.

Essa API possui três componentes principais: *loggers*, *appenders* e *layouts*. Juntos, eles oferecem um conjunto de funcionalidades que permite ao desenvolvedor elaborar uma política de *logging* que leva em consideração o tipo e o nível da mensagem, além de controlar em tempo de execução o seu formato e onde ela será exibida (GULCU, 2004). Na Figura 2.4, é ilustrada a relação entre os componentes de Log4J. Uma solicitação de *logging* com um determinado nível de prioridade é feita a um *logger* para registrar uma mensagem. O *logger*, por sua vez, repassa ao *appender* essa solicitação, pois é ele quem sabe o formato (*layout*) e o destino desse registro. Esse processo de *logging* é bem flexível e ágil, e pouco compromete o desempenho da aplicação. Em seguida, esses componentes são comentados em mais detalhes.

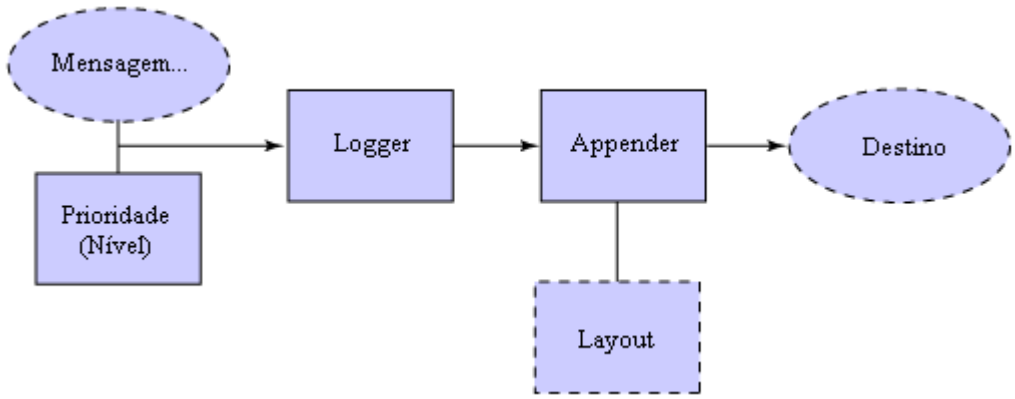

Figura 2.4 - Componentes principais de Log4J.
Fonte: Adaptada de Chan (2005).

O *logger* é o componente responsável por receber uma requisição de *logging* e executá-la. Cada classe de um sistema pode possuir seu próprio *logger* ou compartilhar um, o qual deve possuir um nome que se aconselha ser o nome totalmente qualificado de uma classe ou pacote do sistema. Baseado nisso, Log4j oferece o conceito de hierarquia de *loggers*, permitindo que configurações sejam compartilhadas entre eles. Um *logger* é tido como pai se seu nome precedido de um ponto (‘.’) é prefixo do nome de outro *logger* e não exista nenhum outro ancestral entre eles. Log4j sempre cria um *logger* raiz (*root logger*) pré-configurado, que é

ancestral de todos os outros *loggers*. Por exemplo, o *logger* "a.b" é pai do *logger* "a.b.Classe", e o *logger* raiz é ancestral dos dois.

A cada *logger* pode ser atribuído um nível. Caso não seja, ele herdará o nível do seu *logger* pai. Do menor para o maior, os níveis são: *trace*, *debug*, *info*, *warn*, *error* e *fatal*. Se um pedido de *logging* no nível *y* for feito a um *logger* no nível *x* ele só será executado se *y* for maior ou igual a *x*.

Um *logger* precisa saber para qual destino enviar os pedidos de *logging* feitos a ele. Nesse momento, os *appenders* entram no contexto. Um *appender* é uma saída onde as mensagens são mostradas, como por exemplo: o console, um arquivo, um componente de interface gráfica, um banco de dados, dentre outros. Vários *appenders* podem ser adicionados a um mesmo *logger*. Além disso, um *logger* também herda todos os *appenders* de seu *logger* pai, a não ser que seja explicitamente configurado para isso não acontecer. Dessa forma, um pedido de *logging* feito a um *logger* poderá ter múltiplos destinos, um para cada *appender*.

Loggers e *appenders* são o gerenciador de pedidos de *logging* e o destino desses pedidos, respectivamente. Porém, ainda existe a preocupação do que será registrado, isto é, qual o formato das mensagens? Os *layouts* permitem que essas mensagens possam ser personalizadas a fim de torná-las as mais informativas possíveis perante o funcionamento da aplicação, na visão do usuário. Todo *appender* deve ter um *layout* associado a ele. Essa personalização do formato das mensagens obedece a um padrão de conversão similar à da função `printf` da linguagem C (SCHILDT, 1997). O Quadro 2.1 mostra uma listagem com alguns caracteres de conversão da classe `PatternLayout` de Log4j:

Caractere de Conversão	**Efeito**
%c	Informa o nível do evento de *logging*.
%C	Informa o nome totalmente qualificado da classe onde o evento de *logging* ocorreu.
%d	Informa a data e/ou horário do evento de *logging*.
%L	Informa o número da linha onde a requisição de *logging* foi feita.
%m	Informa a mensagem associada a um determinado nível de um *logger*.
%M	Informa o nome do método onde a requisição de *logging* foi feita.
%r	Informa o tempo decorrido em milisegundos da construção do *layout* até a criação do evento de *logging*.
%t	Informa o nome da *thread* que gerou o evento de *logging*.

Quadro 2.1 - Caracteres de conversão da classe PatternLayout.
Fonte: Adaptado de Log4J (2008).

O básico do funcionamento de uma classe que utiliza o Log4J como mecanismo de geração de *logging* pode ser observado na Listagem 2.4. Essa classe declara um membro `logger` (linha 14), que é responsável por fazer os pedidos de *logging*, onde o nome do *logger* retornado será o nome totalmente qualificado da classe `ExemploLog4j`, isto é, "br.com.ExemploLog4j". O membro `layout` (linha 18) é definido para personalizar o formato das mensagens. Logo depois, um `appender` (linha 19) que tem como destino o console é criado e a ele é associado o `layout`. Em seguida, o `appender` é adicionado ao `logger` para que ele saiba o destino das mensagens (linha 20). O nível do `logger` é definido como "*info*" (linha 21), portanto, somente pedidos de *logging* com nível maior ou igual a esse serão atendidos. A requisição com o nível "*debug*" (linha 24) não é executada, pois não satisfaz a condição desse nível ser maior ou igual ao nível do `logger` (linha 23). Já a requisição de *logging* com o nível "*info*" é executada (linha 26), pois passa no teste da condição (linha 25). Por fim, o resultado da execução desse programa resulta na impressão no console de "0 [main] INFO br.com.Logging - Mensagem Info".

```
 1 package br.com;
 2
 3 import java.io.IOException;
11
12 public class ExemploLog4j {
13
14     private static Logger logger = Logger.getLogger(ExemploLog4j.class);
15
16     public static void main(String[] args) throws IOException {
17
18         Layout layout = new PatternLayout("%r [%t] %-p %c - %m%n");
19         Appender appender = new ConsoleAppender(layout);
20         logger.addAppender(appender);
21         logger.setLevel(Level.INFO);
22
23         if(logger.isDebugEnabled())
24             logger.debug("Mensagem Debug");
25         if(logger.isInfoEnabled())
26             logger.info("Mensagem Info");
27     }
28 }
```

Listagem 2.4 - ExemploLog4j.java
Fonte: Elaborada pelo autor.

Nessa listagem, independente de existir o teste da condição sobre o `logger` estar habilitado para o nível *debug* (linha 23), a solicitação de *logging* com esse nível (linha 24) não seria executada. Esse teste existe para evitar o custo

desnecessário de construção do parâmetro usado na solicitação de *logging* (linha 24), que pode ser alto. Do ponto de vista computacional, Gulcu (2004) afirma que o custo para testar essa condição é muito pequeno e não chega a 1% do tempo de uma operação de *logging*.

Log4j é totalmente configurável tanto em nível de programação quanto por arquivos de configuração. Segundo Gulcu (2004), a primeira e maior vantagem de uma API de *logging* sobre uma simples chamada do método `System.out.println` é a capacidade de habilitar certos trechos de *logging* enquanto outros permanecem desabilitados.

2.5 Eclipse

O Eclipse (ECLIPSE, 2008) é uma plataforma que disponibiliza uma IDE (*Integrated Development Environment*) que fornece um conjunto de funcionalidades que dão suporte ao ciclo de desenvolvimento de aplicações, tais como: editores, compiladores, depuradores, modelagem, refatoração, distribuição, testes automatizados e geração de código. Tudo isso torna o desenvolvimento bastante rápido e prático. Dentre as linguagens suportadas pelo Eclipse, pode-se citar: C, C++, PHP, Ruby, dentre outras. Porém, o seu uso é maior em aplicações Java.

O projeto Eclipse surgiu a partir de um consórcio de empresas lideradas pela IBM em Novembro de 2001. No início de 2004, ele se tornou um *software* gratuito, livre e de código aberto sob a licença EPL (*Eclipse Public License*). Desde então, o Eclipse vem evoluindo com o apoio da comunidade de colaboradores e de grandes empresas e instituições. Essa evolução “distribuída” só é possível porque esse projeto é efetivamente bem coordenado e organizado.

O ponto forte do Eclipse é a sua arquitetura baseada em *plugins*. Um *plugin* é uma funcionalidade integrada a um ambiente de desenvolvido com a finalidade de facilitar o trabalho de seu usuário. No Eclipse, praticamente tudo é *plugin*, com exceção da Plataforma de Inicialização (*Platform Runtime*), que é o seu núcleo. Este é responsável por iniciar o ambiente e carregar todos os outros *plugins* e componentes. Esse modelo de arquitetura permite que o desenvolvedor personalize o ambiente de trabalho com os *plugins* desejados de acordo com o perfil do projeto em desenvolvimento. Existe também um ambiente, chamado de PDE (*Plugin*

Development Environment), que fornece meios para a criação de seus próprios *plugins*.

Uma variedade de recursos inovadores é oferecida, proporcionando mais produtividade. Mecanismos avançados de refatoração, ferramentas de geração de código, recursos que auxiliam na escrita de código e atalhos são alguns exemplos. Além disso, o ambiente gráfico do Eclipse é construído com uma biblioteca de componentes própria, a SWT (*Standard Widget Toolkit*). Ela oferece uma riqueza de componentes gráficos e interfaces de usuário que, entretanto, não comprometem a sua leveza e desempenho.

O conjunto de características aqui discutidas sobre o Eclipse o credenciou como a IDE Java mais popular no mundo. Isso demonstra que os termos transparência e colaboração são fortes ingredientes na busca da IDE ideal.

2.5.1 PDE (*Plugin Development Environment*)

O PDE (PDE, 2008) é uma extensão do Eclipse que fornece um ambiente completo para a construção, compilação, depuração, teste e empacotamento de um *plugin*. No Eclipse, um *plugin* é um programa escrito em Java que respeita uma série de padrões e especificações, que permitem utilizar os recursos da plataforma e se comunicar com ela. Dentre esses recursos, podemos citar: o SWT, o JFace e o JDT (*Java Development Tools*). O SWT oferece uma biblioteca de componentes gráficos, como botões, campos e tabelas. O JFace trabalha em conjunto com SWT e trata de operações rotineiras relacionadas às interfaces de usuário. Já o JDT dá suporte ao desenvolvimento de aplicações ou *plugins* Java.

A idealização de um *plugin* pode ser dividida em três partes: onde, como e o que fazer. A primeira está relacionada ao local onde uma funcionalidade será disponibilizada na bancada de trabalho (*workbench)*, chamado de ponto de extensão. Por exemplo, isso pode ser feito através de um ícone na barra de ferramentas ou de uma nova opção de menu. A segunda parte começa depois do acionamento da funcionalidade, onde pode ser necessária a exibição de interfaces gráficas contendo informações, campos, botões e opções para configurar o que, por fim, na terceira parte, irá acontecer. O resultado final pode ser uma geração de código, uma refatoração, uma formatação de classes ou a configuração de alguma variável do ambiente.

Com o PDE, qualquer pessoa pode desenvolver seu próprio *plugin* e disponibilizá-lo para a comunidade, o que gera um ciclo virtuoso de auto-ajuda, pois na maioria das vezes o *plugin* com a funcionalidade que se precisa já existe. Baseado em uma filosofia de cooperação, grande parte dos *plugins* têm seu código fonte disponível para que a comunidade ajude a incorporar novas funcionalidades e a melhorá-los. Dessa forma, o ambiente se torna cada vez mais rico à medida que novas extensões são adicionadas ou aperfeiçoadas. Pode-se afirmar, com bastaste certeza, que é o PDE é um dos grandes responsáveis pelo sucesso do Eclipse.

2.6 JAXB (*Java Architecture for XML Binding*)

Na maioria dos sistemas desenvolvidos, nos deparamos com o problema de persistir na memória física instâncias de uma classe – objetos. Geralmente, as formas de se armazenar requerem um mapeamento entre o objeto e sua forma persistida, seja em um banco de dados ou em arquivos XML. Isso acontece pela diferença na forma de representação da informação em nível de execução do sistema e em nível de persistência.

JAXB (JAXB, 2008) é uma API Java que permite o mapeamento de classes Java para representações XML que respeitam as definições de Esquemas XML (XML *Schemas*) pré-definidos. Estes são usados para a criação das classes Java através de uma ferramenta de geração de código disponibilizada pelo projeto JAXB, o que torna esse processo bastante simples.

O JAXB possui duas características principais, que são o núcleo do seu sistema. A operação de *marshal* é responsável por fazer a tradução de objetos Java em XML. E a operação *unmarshal*, que faz o inverso, lê um XML e o transforma em objetos Java. Em outras palavras, todo o processo de mapeamento é transparente para o desenvolvedor. Além disso, o JAXB oferece suporte aos principais tipos de dados da linguagem Java, o que o torna bastante flexível, pois representa os objetos de forma mais fiel. A Figura 2.5 mostra em detalhes o funcionamento da arquitetura JAXB.

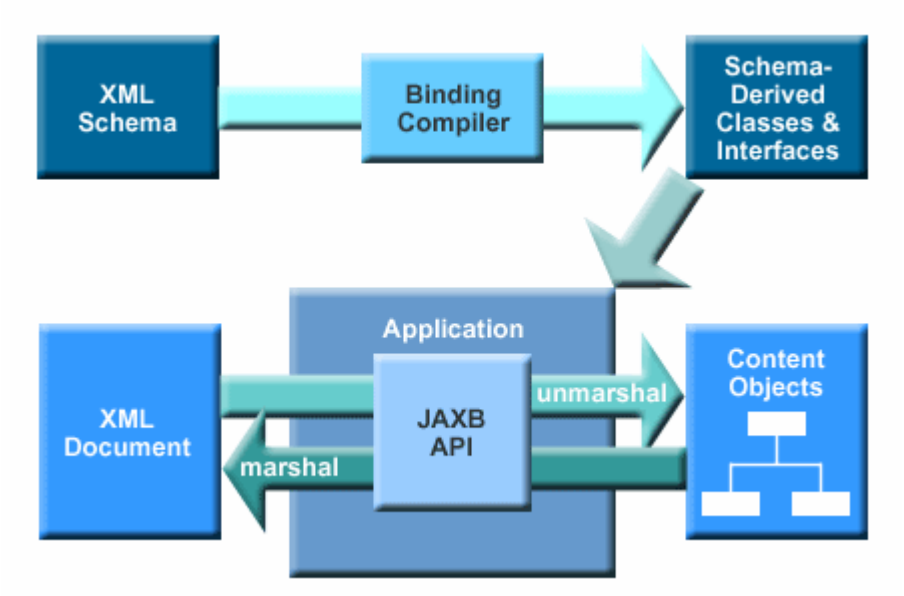

Figura 2.5 - Arquitetura JAXB.
Fonte: Ort e Mehta (2003).

2.7 Trabalhos Relacionados

Nesta subseção, são apresentados alguns trabalhos que influenciaram no amadurecimento da idéia do mecanismo visual desenvolvido, que visa simplificar o gerenciamento do interesse transversal de *logging* através da geração de aspectos. A seguir, cada um desses trabalhos é discutido sob o ponto de vista da sua contribuição para com o desenvolvimento dessa dissertação.

2.7.1 Analisador de Performance de Davies *et al.* (2003)

Davies *et al.* (2003) descrevem um analisador de performance (*profiler*) integrado ao Eclipse como um *plugin*, que se baseia na tecnologia orientada a aspectos para identificar pontos onde recursos de *caching* ajudariam na melhoria do desempenho das aplicações. O uso de orientação a aspectos permite uma instrumentação não-invasiva, flexiva e adaptativa durante a investigação. Buscou-se utilizar essa tecnologia em todas as fases da performance – medição, detecção de problemas, análise (de oportunidades de *caching*) e na investigação e criação das formas de melhorias.

Todo o processo de avaliação de performance é transparente para o usuário, inclusive quanto ao uso de aspectos. O resultado pode ser acompanhado em um ambiente gráfico, através de interfaces fáceis de manipular até mesmo para os mais inexperientes desenvolvedores. O analisador indica os locais de uma aplicação onde

haveria ganho de desempenho com uso de *caching*, e caso o desenvolvedor julgue necessário, pode solicitar a geração automática de aspectos para tratar desse assunto.

A habilidade de identificar, analisar e resolver potenciais problemas de performance dentro de um ambiente de desenvolvimento é uma necessidade, visto a sua importância no desenvolvimento de software. A ferramenta criada no trabalho de Davies *et al*. (2003) expõe uma solução interessante, eficiente e inovadora para solucionar esse tipo de questão.

2.7.2 *Framework* para Testes de Sistemas Distribuídos de Hughes e Greenwood (2003)

A realização de testes é um ponto crítico durante o ciclo de desenvolvimento de qualquer aplicação, porém eles são normalmente negligenciados devido às dificuldades e gastos para realizá-los com sucesso. Isso se torna ainda mais complexo quando se fala em sistemas distribuídos, que exigem um grande esforço de coordenação para testar simultaneamente os componentes distribuídos. Com base nisso, Hughes e Greenwood (2003) propuseram um *framework* implementado usando POA e mecanismos de reflexão, cujo objetivo é simplificar o teste desses tipos de sistema e de outros que apresentam problemas similares.

Esse *framework* propõe uma estrutura de *templates* para a criação de aspectos em AspectJ, onde *tags* personalizadas seriam substituídas, por meio de reflexão, por informações de objetos escolhidos. Por exemplo, o adendo `before() : call(void <CLASSE>.<METODO>){...}` possui a *tag* `<CLASSE>` que seria substituída pelo tipo da classe e a *tag* `<METODO>` que seria substituída pelo nome do método.

Ao se idealizar esse *framework*, a intenção foi a de que ele funcionasse em conjunto com um mecanismo que permitisse a visualização e seleção dos objetos. Amparado nisso, Hughes *et al*. (2004) desenvolveram um ambiente para possibilitar a inserção e remoção automática de cenários de teste em sistemas distribuídos, reduzindo assim o tempo gasto nessa fase de testes. Além disso, é garantido que os processos de teste não comprometerão a corretude do sistema final.

2.7.3 *Framework* para Utilização de Aspectos no Desenvolvimento de Software Dirigido por Modelo

Aspects for Model-Driven Development (HAWKINS e JANUARY, 2006) é um *framework* no formato de um *plugin* para o Eclipse, que permite aos projetistas e desenvolvedores usarem o IBM Rational Software Architect (IRSA, 2008) para aplicar código em AspectJ sem a necessidade de nenhum conhecimento técnico de aspectos. Para isso, é necessário que os especialistas em AspectJ criem recursos nessa linguagem para serem utilizados através de um processo de desenvolvimento de *software* dirigido por modelo.

Esses recursos são disponibilizados por meio da criação de bibliotecas de aspectos que tratam de um único requisito transversal. Essa biblioteca precisa ter um aspecto abstrato com um ponto de variabilidade (*hot spot*) na forma de um conjunto de junção (*pointcut*) abstrato. Além disso, é necessário definir comportamentos relacionados a esse conjunto de junção abstrato através de adendos (*advices*). Por fim, é preciso mapeá-lo a um estereótipo para que possa ser utilizado em diagramas de classe da UML. Por exemplo, ao vincular no diagrama o método `AcessarConta` ao estereótipo `<<controleAcesso>>`, um aspecto responsável pelo controle de acesso durante a execução desse método é gerado automaticamente.

Apesar de muito poderosa, essa ferramenta apresenta limitações de flexibilidade, pois independente do método relacionado a um determinado estereótipo, o comportamento (adendo) será sempre o mesmo. Para ilustrar isso, suponha que se deseje ter uma forma de controle de acesso o mínimo possível diferente da anterior. A solução seria criar uma nova biblioteca e mapeá-la a outro estereótipo, como `<<controleAcesso2>>`. Esse problema poderia gerar um excesso de estereótipos, a ponto de confundir quem os estará utilizando. Não foi possível fazer uma avaliação mais detalhada do *framework*, pois era necessário adquirir a licença do IBM Rational Software Architect.

2.7.4 *Framework* para Implementação Automatizada de Persistência de Couto (2006)

Segundo a definição de Camargo e Masiero (2005), um *Framework* Orientado a Aspectos (FOA) é um sistema semi-completo e reutilizável, formado por classes, cuja presença não é obrigatória, e aspectos, que pode ser instanciado por um desenvolvedor de aplicações. Eles ainda definem *Framework* Transversal (FT) como

sendo um FOA que possui uma arquitetura abstrata destinada ao tratamento de um único interesse transversal e que depende da existência de um sistema para ser utilizado.

A persistência é considerada um requisito transversal, e, além disso, está presente na grande maioria dos sistemas. Com o objetivo de evitar o espalhamento e entrelaçamento de código, Couto (2006) propôs um FT que trata dessa questão, o qual chamou de GAP (*Generator of Aspect for Persistence*). Mais especificamente, são propostas soluções para cada sub-interesse da persistência, tais como: controle de conexão e transação, sincronização de objetos e recuperação de dados. Essas soluções são disponibilizadas em uma biblioteca formada por aspectos abstratos que definem pontos de variabilidade (*hot spots*).

Nesse trabalho, Couto desenvolveu uma ferramenta para geração automática dos aspectos concretos que especializam os aspectos abstratos do GAP e de classes referentes aos objetos persistentes. Essa ferramenta tem como entrada arquivos XML de configuração preparados pelo usuário, por meio de interfaces, baseado no modelo de dados importado do banco. Por exemplo, pode-se definir o mapeamento dos nomes das estruturas no banco em relação ao que será usado na aplicação, os métodos que necessitam de controle de conexão e os métodos transacionais do sistema.

Através da utilização do GAP e da ferramenta de geração de código, é possível se obter uma sensível melhora no tratamento do interesse transversal de persistência. Além de a persistência ser tratada de forma modular, são obtidos ganhos de produtividade e qualidade no sistema final.

2.7.5 Log4E: um Mecanismo Visual Orientado a Objetos para Automatização de *Logging*

Assim como o mecanismo desenvolvido nesta dissertação, o Log4E (LOG4E, 2008) tem como objetivo resolver problemas relacionados a *logging*. O projeto Log4E teve início em Abril de 2004. Entre os dois, a principal diferença está nos meios de se alcançar isso. Enquanto o mecanismo desenvolvido nessa dissertação prioriza uma abordagem orientada a aspectos na automatização do processo de *logging*, o Log4E faz uso de uma abordagem orientada a objetos. Logo, este não resolve os problemas resultantes das limitações dessa abordagem, como o entrelaçamento e espalhamento de código, que acabam acarretando em mais

problemas durante o ciclo de desenvolvimento de *software*, como já discutido anteriormente.

O Log4E apresenta-se como um *plugin* para o Eclipse e possui suporte a bibliotecas de *logging* como Java Logging e Log4j. Ele ajuda o desenvolvedor em tarefas como: declaração de *loggers*, inserção de códigos de *logging* em qualquer parte de um método, substituição de `System.out`'s e monitoramento de variáveis. Também é permitido ao usuário criar *templates* para adaptar os trechos de *logging* de acordo com suas necessidades. Com exceção de algumas configurações que devem ser ajustadas manualmente, todo o processo de criação de *logging* é visual e se apresenta através de menus disponibilizados sobre o editor de código das estruturas do sistema.

3 MÉTODOS E SOLUÇÕES ADOTADAS

A motivação para desenvolver um mecanismo visual baseado em aspectos que facilite o gerenciamento do interesse transversal de *logging* se justifica pelo aparente fato de que, atualmente, os sistemas puramente orientados a objetos tratam dessa questão importante de forma inadequada à compreensão humana. Além disso, existe também uma certa inércia quanto ao aprendizado de novas técnicas de desenvolvimento capaz de resolver esse problema, como a POA. O mecanismo desenvolvido nessa dissertação ajuda a tornar os sistemas mais fáceis de entender, adaptar, manter, estender e evoluir.

O início do trabalho consistiu em um estudo aprofundado das principais áreas e tecnologias relacionadas ao que foi proposto como objetivo geral. Dentre elas, podemos citar: POA, AspectJ, *Logging* e alguns trabalhos relacionados. A finalidade desse estudo foi a obtenção dos fundamentos necessários para se planejar o prosseguimento do trabalho e dar respaldo às soluções adotadas. Em outras palavras, esse estudo permitiu o amadurecimento das idéias.

No decorrer desta seção, serão detalhadas as decisões tomadas e os caminhos percorridos durante o desenvolvimento desse trabalho relativos à concretização de cada um dos objetivos específicos traçados.

3.1 Definição de um conjunto de funcionalidades importantes relacionadas a *logging*

Primeiramente, antes de definir essas funcionalidades, foi necessário entender bem os conceitos de *logging*, assim como os contextos onde costuma ser utilizado. Para cada uma das funcionalidades levantadas, foi analisada a viabilidade de oferecê-la através de aspectos em AspectJ. Além disso, percebeu-se que a utilização da biblioteca de *logging* Log4j acrescentaria poder e flexibilidade ao mecanismo.

Em uma estratégia de *logging* eficiente, é preciso que os pontos monitorados durante a execução da aplicação sejam bem definidos para refletir com fidelidade o seu comportamento. Nesse sentido, AspectJ oferece um amplo e poderoso conjunto de recursos para a definição desses pontos, chamados pontos de junção. Além disso,

os adendos permitem definir quando (antes, durante ou depois) e o que fazer no momento em que um desses pontos for executado.

Outra característica importante do *logging* está nas informações contidas nas mensagens usadas para acompanhar o funcionamento do sistema. Quanto mais possibilidades de informações forem permitidas na criação de uma mensagem, mais útil ela se tornará, uma vez que descreverá com mais exatidão o que está acontecendo. Várias informações importantes podem ser capturadas usando os objetos `thisJoinPoint` e `thisJoinPointStaticPart` oferecidos por AspectJ, através de reflexão, quando um ponto de junção for executado. Log4j também permite a customização dessas mensagens com o uso de caracteres de conversão.

Para as mensagens de *logging* serem analisadas é necessário que elas sejam apresentadas em algum meio, como por exemplo: um console, um arquivo ou um banco de dados. Log4j permite definir destinos como esses para as mensagens – os *appenders*.

Ocasionalmente, pode não ser desejável manter o requisito de *logging* na aplicação final, o que acontece quando o seu principal objetivo é auxiliar as equipes de desenvolvimento e teste no acompanhamento do sistema para prevenção e busca por problemas. Por conta disso, poderia ser necessário adicionar e remover trechos de *logging* várias vezes. Entretanto, seria muito mais simples manter esses trechos na aplicação e permitir que eles fossem habilitados ou desabilitados através de uma central de configurações. Log4j dispõe de estruturas denominadas *loggers* que controlam o que deve ser monitorado através do conceito de nível e que podem ser configuradas tanto em nível de programação quanto por arquivos de configuração.

A sobrecarga proporcionada pelo *logging* é um fator preocupante. Por isso, desde o início do seu projeto, Log4j é uma biblioteca que tem a otimização como meta. Quando o *logging* está desabilitado para determinado ponto, o gasto será pequeno e se resumirá a uma comparação booleana (ver Subseção 2.4). O uso de Log4j no mecanismo permite a incorporação desses benefícios.

Para cada uma das funcionalidades descritas, foram verificadas alternativas que serviram de base para a idealização do mecanismo. Ao fim dessa etapa, concluiu-se que a utilização em conjunto de AspectJ e Log4j contribuiria bastante no resultado desta dissertação.

3.2 Idealização de um mecanismo capaz de oferecer essas funcionalidades de *logging*

Durante essa etapa, a investigação de trabalhos relacionados contribuiu para a idealização desse mecanismo. Foram pesquisados e analisados trabalhos que propunham abstrair o uso de aspectos das mais diferentes formas para outros tipos de interesses transversais que não fossem *logging*. Outro trabalho verificado oferecia um mecanismo visual para a automatização de *logging*, porém usando uma abordagem orientada a objetos, o que não resolve o entrelaçamento e espalhamento de código. Essa investigação permitiu assimilar as boas idéias, a fim de adaptá-las ao mecanismo desenvolvido e propor soluções mais interessantes para resolver o problema do *logging*.

A intenção foi desenvolver um mecanismo que fosse o mais simples e intuitivo possível de utilizar e que abstraísse totalmente o uso de aspectos. Em busca disso, uma solução visual foi imaginada, já que para a maioria das pessoas é mais fácil pensar e relembrar as coisas em termos de imagens. Em contrapartida, a programação textual é mais complicada, uma vez que exige muito mais do raciocínio, da memória e do tempo do desenvolvedor. Ao se remover a necessidade de transformar as idéias de *logging* diretamente em expressões textuais, permite-se suavizar a curva de aprendizado.

As funcionalidades levantadas na etapa anterior foram agrupadas em submecanismos visuais com a finalidade de trabalharem em conjunto para simplificar o gerenciamento do interesse transversal de *logging*. O mecanismo de customização de mensagens permitiria a criação de modelos de mensagem customizáveis e reutilizáveis. Já o mecanismo de configuração de *loggers* possibilitaria a manipulação das estruturas responsáveis por fazer o controle dos pedidos de *logging*. Enquanto isso, os mecanismos de configuração de *logging* seriam responsáveis pela definição da estrutura de cada aspecto usado para *logging*. Finalmente, o mecanismo de geração de código seria utilizado para a criação dos aspectos e classes relacionadas ao *logging*, baseado nas configurações definidas.

3.3 Investigação e escolha de um ambiente de desenvolvimento que suporte esse mecanismo

De nada adianta existir a solução para um determinado tipo de problema se essa solução não está acessível a quem irá utilizá-la. Nessa perspectiva, um dos pré-

requisitos para o desenvolvimento do mecanismo foi tentar torná-lo disponível ao maior número de interessados possíveis. Percebeu-se que isso poderia ser alcançado integrando o mecanismo a um ambiente de desenvolvimento popular no formato de um *plugin*.

Depois de uma pesquisa, dois ambientes de desenvolvimento foram escolhidos como candidatos: o Netbeans (NETBEANS, 2008) da Sun Microsystems e o Eclipse (ECLIPSE, 2008) do Projeto Eclipse. A escolha desses ambientes deveu-se ao fato deles serem gratuitos e de código aberto, além de suportarem a adição de *plugins*.

Como o Eclipse é a IDE Java mais conhecida e utilizada mundialmente, foi a primeira a ser analisada. Foram avaliados os recursos disponibilizados pelo Eclipse que permitiriam a integração do mecanismo idealizado a esse ambiente de desenvolvimento. Inicialmente, foi verificado que o Eclipse possui a extensão PDE, cujo objetivo é dar suporte a todas as etapas do desenvolvimento de um *plugin*. Essa extensão permite o uso de bibliotecas que oferecem uma riqueza de opções relacionadas a componentes gráficos, operações rotineiras e de suporte ao desenvolvimento de *plugins*.

Outra característica importante levada em consideração no momento da escolha da IDE foi o suporte a AspectJ. Verificou-se que o Eclipse possui um conjunto de *plugins*, chamado AJDT (*AspectJ Development Tools*) (AJDT, 2008) que oferece as condições necessárias para o desenvolvimento de software orientado a aspectos por meio de AspectJ. Além disso, este é um projeto de código aberto.

Por conta das excelentes opções oferecidas pelo Eclipse para a continuidade do desenvolvimento do trabalho e devido à sua popularidade, optou-se por não estender a investigação sobre o outro ambiente de desenvolvimento candidato - o Netbeans. Assim sendo, o Eclipse foi escolhido.

3.4 Projeto do mecanismo e da sua integração ao ambiente de desenvolvimento escolhido

Esse passo se concentrou na definição da arquitetura e do projeto do mecanismo criado. Para isso, as formas com que cada um dos submecanismos idealizados iria oferecer suas funcionalidades foram mais detalhadas e planejadas, assim como a integração deles ao Eclipse. Os protótipos para as interfaces gráficas

desses mecanismos também foram criados. Além disso, os meios de se persistir as configurações foram definidos.

O mecanismo desenvolvido tinha como principal objetivo oferecer ao usuário uma boa usabilidade. Por isso, as interfaces gráficas foram idealizadas recorrendo ao uso de recursos visuais, aumentando assim os níveis de interatividade. Dentre esses recursos, podemos citar: símbolos, botões, caixas de seleção, tabelas, árvores de itens, campos selecionáveis, mensagens de ajuda e advertência e outros.

Uma boa opção encontrada para armazenar as configurações criadas a partir de cada submecanismo seria através do uso de XML. Para isso, foi decidido que seria adotada a API Java JAXB, que simplifica a transformação de um objeto do sistema em XML, e vice-versa.

Como o mecanismo seria incorporado ao Eclipse como um *plugin*, foi necessário definir onde e como essas funcionalidades seriam disponibilizadas na bancada de trabalho. Para isso, o PDE disponibiliza os pontos de extensão. Optou-se por oferecer esses mecanismos através de menus sobre os elementos mostrados na visão da estrutura do projeto (*Package Explorer*), tais como: classes, pacotes e o próprio projeto. Sempre que um mecanismo for acionado, será exibida a interface gráfica referente a ele.

O projeto de cada submecanismo foi realizado considerando o que o PDE oferece. Além disso, o mecanismo de configuração de mensagens faria uso de recursos de AspectJ e Log4j para obter informações dinâmicas e estáticas dos pontos monitorados durante a execução da aplicação. Já o mecanismo de configuração de *loggers* usaria os conceitos definidos por Log4j para os *loggers* a fim de permitir a configuração e visualização hierarquizada deles. Enquanto isso, os mecanismos de configuração de *logging* dependeriam desses dois últimos e de alguns recursos de AspectJ ligados à criação de pontos de junção e adendos para definir a estrutura desejada de cada aspecto. Por fim, o mecanismo de geração de código seria sempre invocado de forma transparente ao fim do salvamento de cada configuração para criação dos aspectos e classes usadas para *logging*.

3.5 Construção do mecanismo proposto.

A construção do mecanismo se refere à fase de implementação do que foi especificado no projeto. Felizmente, tudo que foi planejado pôde ser realizado com sucesso. Foram codificados os pontos de extensão, as interfaces gráficas, a

persistência, a lógica por detrás de cada um dos submecanismos e a interação entre eles. Esse resultado é apresentado em detalhes na Seção 4.

Durante o desenvolvimento, processos de refatoração (*refactoring*) foram realizados a fim de melhorar a qualidade do código. Além disso, buscou-se fazer testes durante e ao fim do desenvolvimento como forma de garantir a qualidade e confiabilidade do sistema. A última ação dessa etapa consistiu no empacotamento do mecanismo como um *plugin*, a fim de disponibilizá-lo para uso através da IDE Eclipse.

4 ASPLOG: UM MECANISMO VISUAL PARA AUTOMATIZAÇÃO DE *LOGGING*

O mecanismo baseado em aspectos desenvolvido nesta dissertação, chamado Asplog (Aspect Logger), tem o objetivo de simplificar o processo de desenvolvimento de trechos de código relacionados a *logging*. A importância do *logging* em aplicações pode ser percebida na fase de desenvolvimento, em atividades de *tracing*, passando pelas fases de testes e integração. Entretanto, seu maior uso é após a implantação, já que a ele é atribuída a tarefa de acompanhar o funcionamento da aplicação, auxiliando assim, a equipe de suporte e manutenção.

Logging é um requisito que possui comportamento transversal e, portanto, pode acarretar problemas de entrelaçamento e espalhamento de código. Para mostrar o quanto o código de *logging* polui o código e diminui a sua legibilidade, observe o exemplo da Classe `Pessoa` na Listagem 4.1, que faz uso de Log4j, uma API Java de *logging*.

A classe `Pessoa` possui código relacionado à criação e inicialização do objeto `logger` (estático) pertencente à classe nas linhas 10 e de 13 a 17. Esse objeto é responsável por despachar as requisições de *logging*. Além disso, antes da execução dos blocos de código dos métodos `comer`, `dormir` e `trabalhar`, que foram substituídos por comentários por questão de objetividade, é solicitado ao objeto `logger` a criação de um registro informando que aquele método iniciará sua execução. Analisando essa classe, pode-se perceber o quanto sua coesão modular foi prejudicada pela inserção de trechos de código transversais de *logging* que se misturam com as suas funcionalidades principais - os interesses do negócio.

Ao se projetar a situação ocorrida na classe `Pessoa` em um sistema, depara-se com um problema que contamina todo o código, pois ocorre um espalhamento do *logging* pelas classes, aumentando substancialmente o tamanho de cada uma delas. A Figura 4.1 ilustra a solução de *logging* normalmente utilizada nas aplicações. Nela, sempre que uma classe necessitar de mensagens para acompanhar o seu comportamento, é feita uma chamada explícita ao *logger* correspondente. Vale

ressaltar ainda que a implementação desse interesse torna-se trabalhosa quando feita de forma convencional, ou seja, quando é necessário manipular diretamente as suas instruções. Além disso, remover as instruções de *logging* das classes pode ser uma tarefa bastante árdua e propensa a erros, visto que esse interesse não se apresenta modularizado no sistema.

```
1  package br.com;
2
3  import org.apache.log4j.ConsoleAppender;
7
8  public class Pessoa {
9
10     private static Logger logger = Logger.getLogger(Pessoa.class);
11     private String nome;
12
13     static {
14         logger.addAppender(
15                 new ConsoleAppender(new PatternLayout("%m%n")));
16         logger.setLevel(Level.DEBUG);
17     }
18
19     public Pessoa(String nome) {
22
23     public void comer() {
24         logger.info(nome + ": Antes de comer.");
25         //Código para comer.
26     }
27
28     public void dormir() {
29         logger.info(nome + ": Antes de dormir.");
30         //Código para dormir.
31     }
32
33     public void trabalhar() {
34         logger.info(nome + ": Antes de trabalhar.");
35         //Código para trabalhar.
36     }
37 }
```

Listagem 4.1 - Pessoa.java
Fonte: Elaborada pelo autor.

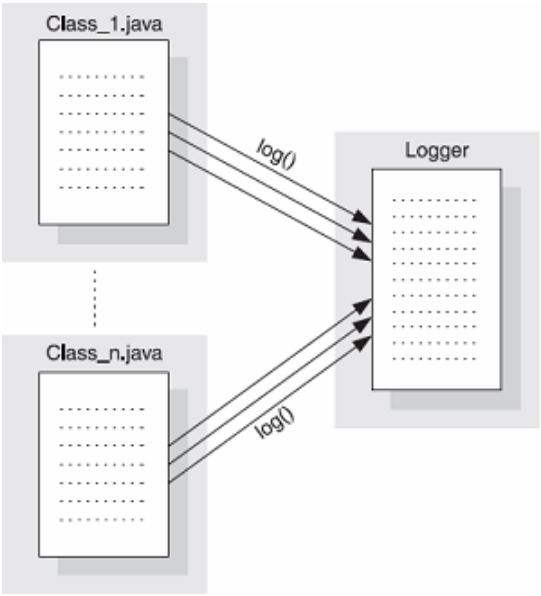

Figura 4.1 - Solução convencional de *logging*.
Fonte: Laddad (2003).

Na Figura 4.2, é mostrada em alto nível a estrutura de uma solução que faz uso de aspectos para atingir o objetivo de modularização de *logging*. Comparando os benefícios dessa solução com a da Figura 4.1, percebe-se que as classes não precisam mais se preocupar com *logging*, passando a focar apenas nos interesses de negócio. Isso permite que as classes se tornem estruturas mais coesas. Por conseqüência disso, é necessária a criação de um aspecto responsável por modularizar os códigos de *logging* de interesse daquelas classes e saber os pontos exatos onde devem ser executados. Em outras palavras, pode-se dizer que houve uma inversão de controle.

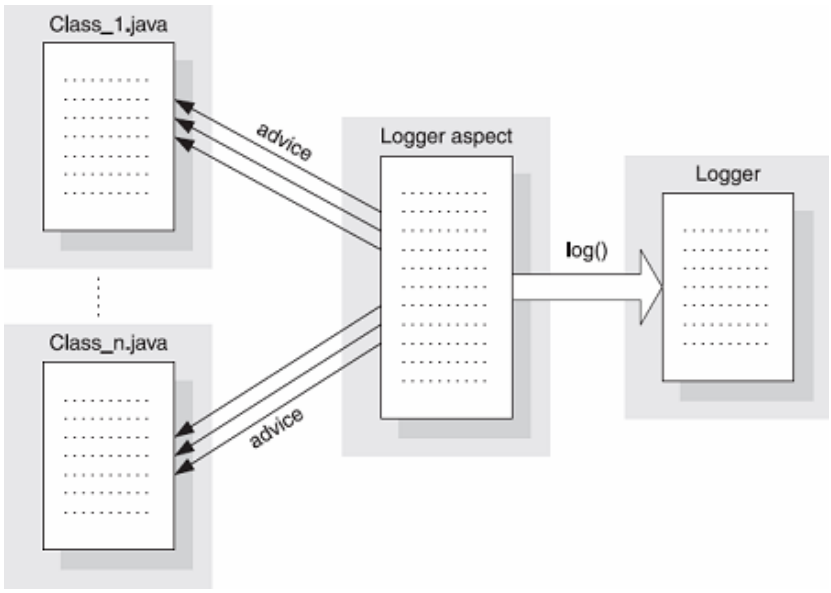

Figura 4.2 – Solução de *logging* baseada em aspectos.
Fonte: Laddad (2003).

Com o objetivo de resolver o problema de modularização do interesse transversal de *logging* e de facilitar a criação de seus trechos de código, auxiliando o desenvolvedor na construção de sistemas, o Asplog oferece os seguintes mecanismos, todos configuráveis visualmente:

- **Mecanismo de Customização de Mensagens** - define modelos de mensagem reutilizáveis usados para *logging*.
- **Mecanismo de Configuração de *Loggers*** - gerencia as estruturas responsáveis pelo controle das requisições de *logging*, chamadas de *loggers*.
- **Mecanismos de Configuração de *Logging*** - indicam pontos de uma aplicação onde o comportamento (métodos) ou estado (membros) das classes devem ser monitorados através de *logging*.
- **Mecanismo de Geração de Código -** baseia-se nas configurações definidas pelos outros mecanismos para criar os aspectos e classes responsáveis por modularizar o *logging*.

Resumindo, o Asplog visa simplificar o trabalho do desenvolvedor no que diz respeito ao interesse transversal de *logging*, através de um processo de criação completamente visual e flexível que possui como resultado final aspectos e classes responsáveis pela modularização desse interesse.

Ao se planejar o desenvolvimento do Asplog, uma das metas traçadas foi a de torná-lo bem acessível à comunidade de desenvolvedores Java. Para tanto, foi analisada a viabilidade de tornar isso possível e chegou-se à conclusão de que a integração desse sistema à IDE Eclipse seria uma ótima solução.

O esquema de funcionamento do Asplog é mostrado na Figura 4.3. Nela, são ilustrados os três mecanismos do Asplog com os quais o desenvolvedor pode interagir para criar as suas próprias configurações. Os mecanismos de configuração de *logging* são dependentes das configurações geradas pelo mecanismo de customização de mensagens e pelo mecanismo de configuração de *loggers*. Assim que é solicitado o salvamento de uma configuração, dois processos ocorrem. O primeiro é o seu armazenamento em um arquivo XML. Logo depois, o mecanismo de geração de código é acionado e as classes e aspectos responsáveis pela modularização do *logging* são criados em função do conteúdo lido dos arquivos XML. Toda a manipulação objeto-XML ocorre usando JAXB. As classes do sistema passam a ser entrecortadas pelos aspectos gerados. É também mostrado que os mecanismos do Asplog foram incorporados ao Eclipse como um *plugin*. No projeto do Asplog, foi planejada uma grande junção de conceitos entre AspectJ e Log4J que é detalhada no transcorrer desta seção.

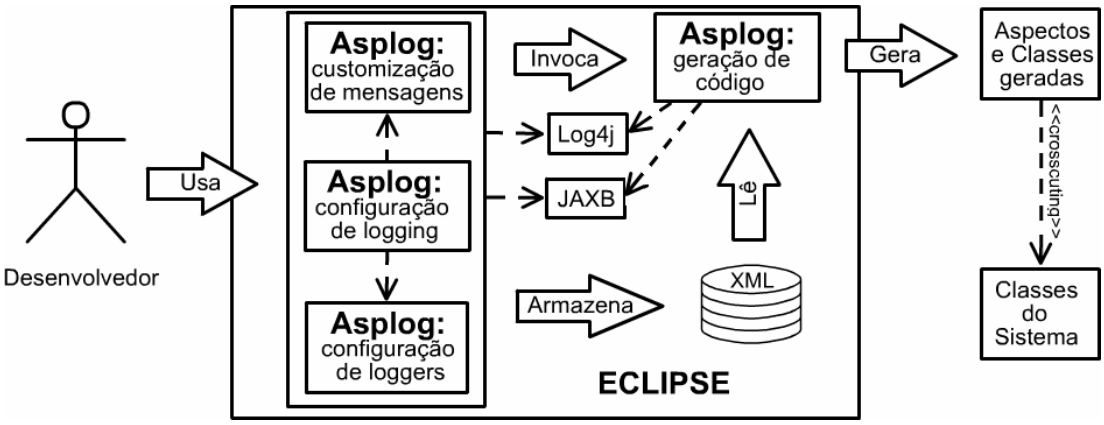

Figura 4.3 – Esquema de funcionamento do Asplog.
Fonte: Elaborada pelo autor.

O diagrama de classes da UML (BOOCH *et al*., 1999) para o Asplog é mostrado em alto nível na Figura 4.4. Sugere-se, ao final da leitura desta seção, fazer uma releitura do trecho relacionado a explicações sobre esse diagrama para um maior entendimento. Esse diagrama foi dividido em três pacotes principais: `pages`, `jaxb` e `generation`.

No pacote `pages`, estão presentes as classes que representam as interfaces gráficas com as quais o usuário pode interagir. Esse pacote é dividido em outros três: `message`, `logger` e `logging`, que se referem, respectivamente, às classes do mecanismo de configuração de mensagens, do mecanismo de configuração de *loggers* e dos mecanismos de configuração de *logging*. Cada uma dessas classes implementa a interface `IPage`, que define um conjunto de métodos padrão para as páginas. São eles:

- `loadConfig` – carrega na interface gráfica uma configuração já existente.
- `saveConfig` – salva uma configuração criada ou editada pelo usuário.
- `showPage` – exibe a interface gráfica.
- `validateFields` – verifica se uma configuração está correta e completa.

O pacote `generation` está diretamente relacionado ao mecanismo de geração de código. Portanto, é nele que está a classe `Generator`, que é responsável pela geração de classes (`createLoggerInitializerClass`) e de aspectos (`createLoggingAspect`) responsáveis pelo controle de *logging*.

O pacote `jaxb` é dividido em dois pacotes: `generated` e `support`. O subpacote `generated` contém as classes de entidade geradas automaticamente pelo JAXB, a partir de XML *Schemas* definidos. Esse subpacote `generated` se divide

em outros três: `message`, `logger` e `logging`, onde suas classes são utilizadas, respectivamente, para representar as configurações criadas pelo usuário através do mecanismo de configuração de mensagens, do mecanismo de configuração de *loggers* e dos mecanismos de configuração de *logging*.

Enquanto isso, as classes do subpacote `support` são responsáveis por controlar a manipulação e persistência das estruturas definidas no subpacote `generated`. Essas classes são:

- `MessageSupport` – permite obter a lista de modelos de mensagem (`getMessageFormatList`) e alterar os elementos dessa lista (`setMessageFormatList`).
- `LoggerSupport` – permite obter a lista de *loggers* (`getLoggertList`) e alterar os elementos dessa lista (`setLoggerList`).
- `LoggingSupport` – permite obter a lista de configurações de *logging* (`getConfigList`) e alterar os elementos dessa lista (`setConfigList`).

Ainda nesse diagrama, é mostrada a relação de dependência entre alguns pacotes, o que permite entender um pouco mais sobre o esquema de funcionamento do Asplog (Figura 4.3). Abaixo, segue a lista de dependências entre os pacotes, do primeiro em relação ao segundo, e o objetivo da relação:

- `pages.logger e generation` – geração das classes de inicialização dos *loggers*.
- `pages.logging e generation` – geração dos aspectos responsáveis pelo *logging*.
- `pages e jaxb.support` – persistência das configurações criadas.
- `generation e jaxb.support` – obtenção de informações necessárias a geração de classes e aspectos.
- `jaxb.support` e `jaxb.generated` – utilização das classes de entidade para manipulação e persistência objeto-XML.

No transcorrer dessa seção, serão apresentados alguns diagramas de seqüência da UML, que se aproveitam do que foi definido no diagrama de classes, para detalhar um pouco mais o funcionamento do Asplog.

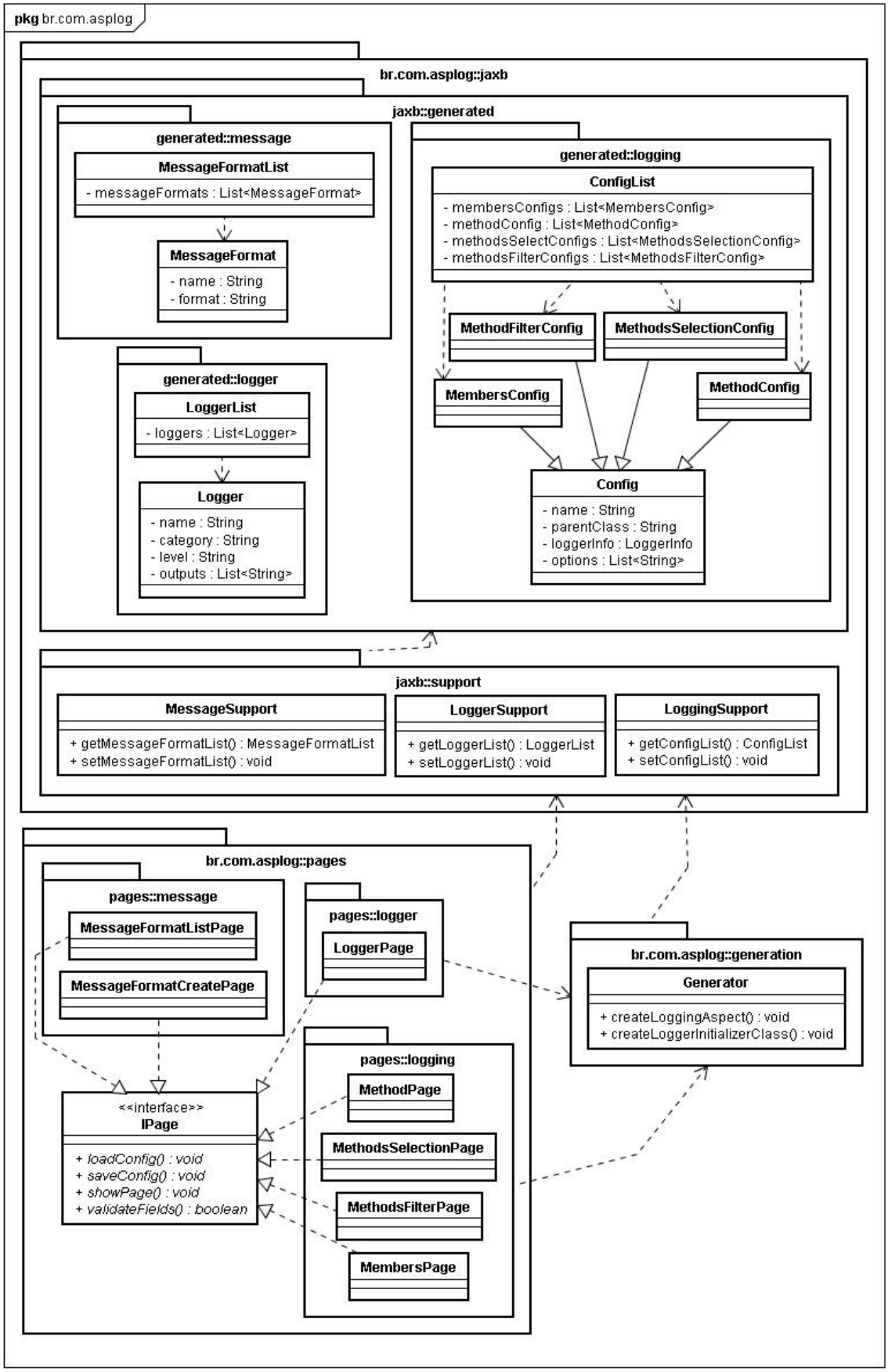

Figura 4.4 – Diagrama de classes do Asplog (em alto nível).
Fonte: Elaborada pelo autor.

A fim de tornar mais objetiva as explicações de como funcionam os mecanismos oferecidos pelo Asplog, foi criado no Eclipse um pequeno projeto chamado `Mercado`, que possui as classes `Item` e `NotaFiscal`, definidas na Listagem 4.2 e na Listagem 4.3, respectivamente. Os métodos mais simples foram sintetizados como forma de reduzir o tamanho das listagens.

A classe `Item` é responsável por armazenar informações sobre os itens vendidos em um mercado qualquer, como seu código, descrição e preço. Além disso, com o objetivo se simplificar o esquema, essa classe também é responsável por fazer o controle de estoque. Se um item for vendido, ele é subtraído do estoque. Se for devolvido, ele é adicionado. Quando a quantidade de um item chega à zero, ele passa a fazer parte do histórico.

Já a classe `NotaFiscal` representa uma compra feita por um determinado cliente, contendo informações como o nome e CPF desse cliente, a data da compra e o carrinho de compras, que armazena os itens que estão sendo comprados, junto com as suas respectivas quantidades. Itens podem ser adicionados ou removidos do carrinho. Essa classe permite imprimir a lista de itens da compra e calcular o valor total dela.

```java
package br.com.mercado;

import java.util.ArrayList;

public class Item {

    private static int geradorCodigo;
    public static List<Item> listaItens, listaItensHistorico;
    private int codigo, qtdeEstoque;
    private double preco;
    private String descricao;

    static {
        listaItens = new ArrayList<Item>();
        listaItensHistorico = new ArrayList<Item>();
    }

    public Item(String descricao, int qtdeEstoque, double preco) {
        this.codigo = ++geradorCodigo;
        this.descricao = descricao;
        this.qtdeEstoque = qtdeEstoque;
        this.preco = preco;
        listaItens.add(this);
    }

    public void aumentarEstoque(int qtde) {
    public void diminuirEstoque(int qtde) throws Exception {
        if(qtdeEstoque >= qtde)
            qtdeEstoque -= qtde;
        else
            throw new Exception("Estoque insuficiente.");

        if(qtdeEstoque == 0)
        {
            listaItensHistorico.add(this);
            listaItens.remove(this);
        }
    }
    public static Item getItemPorCodigo(int codigo) throws Exception {
        for(Item i: listaItens)
        {
            if(i.getCodigo() == codigo)
                return i;
        }
        throw new Exception("Item não encontrado.");
    }
    public int getCodigo() {
    public String getDescricao() {
    public double getPreco() {
}
```

Listagem 4.2 - Item.java
Fonte: Elaborada pelo autor.

```java
package br.com.mercado;

import java.util.Date;

public class NotaFiscal {

    private String cliente, CPF;
    private Date dataCompra;
    private Map<Item, Integer> carrinhoDeCompra;

    public NotaFiscal(String cliente, String CPF)
    {
        this.cliente = cliente;
        this.CPF = CPF;
        this.dataCompra = new Date();
        carrinhoDeCompra = new HashMap<Item, Integer>();
    }

    public void adicionarItem(int codigo, int qtde) throws Exception
    {
        Item item = Item.getItemPorCodigo(codigo);
        item.diminuirEstoque(qtde);
        carrinhoDeCompra.put(item, new Integer(qtde));
    }
    public void removerItem(int codigo) throws Exception
    {
        Item item = Item.getItemPorCodigo(codigo);
        item.aumentarEstoque(carrinhoDeCompra.get(item).intValue());
        carrinhoDeCompra.remove(item);
    }
    public double calculaTotal() {
        double total = 0;
        Set<Item> itens = carrinhoDeCompra.keySet();
        for(Item i: itens)
        {
            total += i.getPreco() * carrinhoDeCompra.get(i);
        }
        return total;
    }
    public void imprimeCarrinho()
    {
        Set<Item> itens = carrinhoDeCompra.keySet();
        for(Item i: itens)
        {
            System.out.println(
                    carrinhoDeCompra.get(i) + " - " + i.getDescricao());
        }
    }
    public String getCliente() {
    public String getCPF() {
    public Date getDataCompra() {
}
```

Listagem 4.3 – Pessoa.java
Fonte: Elaborada pelo autor.

As subseções seguintes descrevem em detalhes cada um dos mecanismos disponibilizados pelo Asplog.

4.1 Mecanismo de Customização de Mensagens

Uma estratégia de *logging* eficiente é muito importante nas aplicações, seja somente para acompanhar o seu funcionamento ou para diagnosticar um problema que muitas vezes é difícil de ser observado, e conseqüentemente, reproduzido. Um dos pré-requisitos para se atingir uma boa estratégia de *logging* é a preocupação com o conteúdo das mensagens que retratam o comportamento do sistema, que precisam ser tão informativas quanto à criticalidade dos pontos que estão sendo monitorados e a necessidade da pessoa responsável por analisá-las.

Um registro de *logging* pode conter diversas informações importantes sobre a execução do sistema, tais como:

- O horário e a data em que o evento ocorreu;
- O nome da classe e o número da linha que está sendo executada;
- O nome do método e a sua assinatura, assim como os valores dos seus parâmetros e do seu retorno;
- O nome e o valor de um membro de uma classe, que pode ser monitorado para indicar se houve alguma mudança no estado da classe;
- O nome da *thread* em execução, que é muito útil no caso de ambientes multi-processados;
- O nível das mensagens, que ajuda a classificá-las por importância;
- Uma mensagem personalizada, a qual pode ser bastante informativa.

O Asplog simplifica o processo de customização de mensagens através de um mecanismo para criação de modelos reutilizáveis, ou seja, uma vez criado um modelo, basta fazer uma referência ao seu nome para utilizá-lo em qualquer lugar onde o *logging* for necessário, como será visto adiante na Subseção 4.3. Várias informações podem ser adicionadas em um modelo, entre elas as que foram enumeradas anteriormente. Para disponibilizar essas informações, o mecanismo se baseia nos caracteres de conversão da classe `PatternLayout` de Log4j e no uso do objeto `thisJoinPoint` de AspectJ, que oferece informações dinâmicas do ponto de junção em execução, através do suporte à reflexão. O Quadro 4.1 descreve as informações que podem ser usadas no Asplog para a personalização de mensagens com os seus respectivos caracteres de conversão (código).

Tipo	Código	Efeito
Nome da Classe	&CN;	Informa o nome totalmente qualificado da classe onde o evento de *logging* ocorreu.
Data	&DT;	Informa a data que o evento de *logging* ocorreu.
Hora	&HR;	Informa a hora que o evento de *logging* ocorreu.
Nível	&LVL;	Informa o nível da mensagem de *Logging*.
Número da Linha	&LIN;	Informa o número da linha onde a requisição de *logging* foi feita.
Nome do *Logger*	&LON;	Informa o nome do *logger* responsável por despachar o pedido de *logging*.
Mensagem	&ME;	Informa a mensagem associada a um determinado nível de um *logger*.
Milisegundos	&MI;	Informa o tempo decorrido em milisegundos da construção do *layout* até a criação do evento de *logging*.
Nome da *Thread*	&TI;	Informa o nome da *thread* que gerou o evento de *logging*.
Nome do Método	&MTN;	Informa o nome do método onde a requisição de *logging* foi feita.
Assinatura do Método	&MTH;	Informa a assinatura do método onde a requisição de *logging* foi feita.
Tipos dos Parâmetros do Método	&MTPT;	Informa os tipos dos parâmetros do método onde a requisição de *logging* foi feita.
Valores dos Parâmetros do Método	& MTPV;	Informa os valores dos parâmetros do método onde a requisição de *logging* foi feita.
Tipo de Retorno do Método	&MTRT;	Informa o tipo de retorno do método onde a requisição de *logging* foi feita.
Valor de Retorno do Método	&MTRV;	Informa o valor de retorno do método onde a requisição de *logging* foi feita.
Nome do Membro	&MBN;	Informa o nome do membro sobre o qual requisição de *logging* foi feita.
Tipo do Membro	&MBT;	Informa o tipo do membro sobre o qual requisição de *logging* foi feita.
Valor do Membro	&MBV;	Informa o valor do membro sobre o qual requisição de *logging* foi feita.

Quadro 4.1 - Informações utilizadas para customização de mensagens.
Fonte: Elaborado pelo Autor

Na Figura 4.5 é mostrada a interface gráfica que lista todos os modelos de mensagem reutilizáveis já criados. No Eclipse, o seu caminho de acesso é "*Window → Preferences → Aspect Logger → Message's Formats*". Na tabela, cada modelo possui um nome (*name*), um tipo (*type*) e um formato (*format*). O tipo indica se aquele modelo será usado para gerar registros de *logging* que monitoram execuções de métodos ou alterações e leituras em membros de uma classe. O formato diz

respeito ao modelo customizado, propriamente. Na parte inferior da interface, é mostrado um exemplo de saída para o modelo selecionado na tabela. Mais abaixo, é dada a opção de se criar, editar ou apagar um modelo.

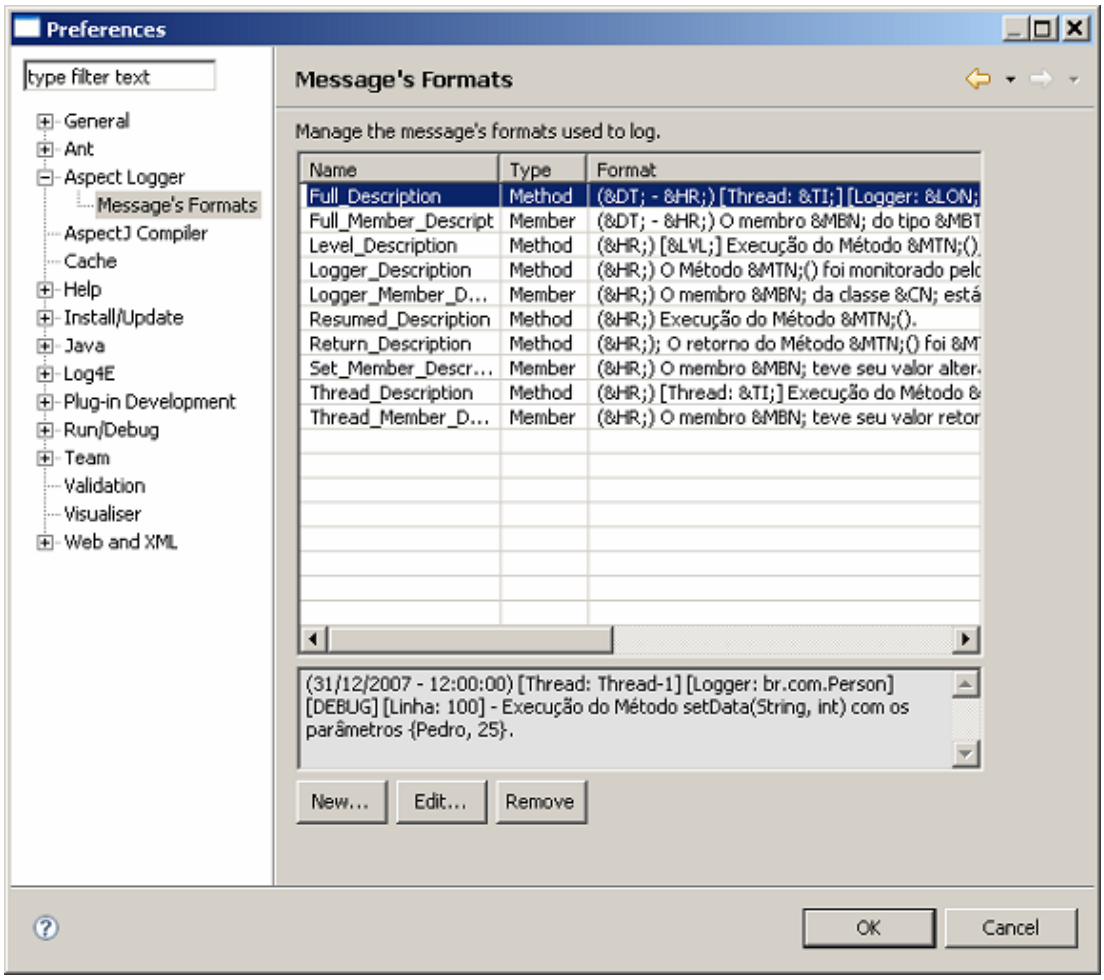

Figura 4.5 – Listagem dos modelos de mensagem reutilizáveis.
Fonte: Elaborada pelo autor.

Após o acionamento do botão para a criação ou edição de um modelo, é exibida uma interface gráfica como a da Figura 4.6, onde é necessário informar o seu nome e tipo. No campo onde se cria a mensagem customizada (*Customized Message Format*), pode-se misturar texto simples com os caracteres de conversão, cuja memorização não é necessária, pois ao clicar em um dos botões acima, o seu respectivo código é adicionado à mensagem na posição do cursor. Antes de salvar o modelo, é possível acompanhar na parte inferior da interface um exemplo de como ficaria uma mensagem de *logging* a partir dele. Depois de salvo, ele é armazenado no arquivo XML responsável pela lista dos modelos existentes.

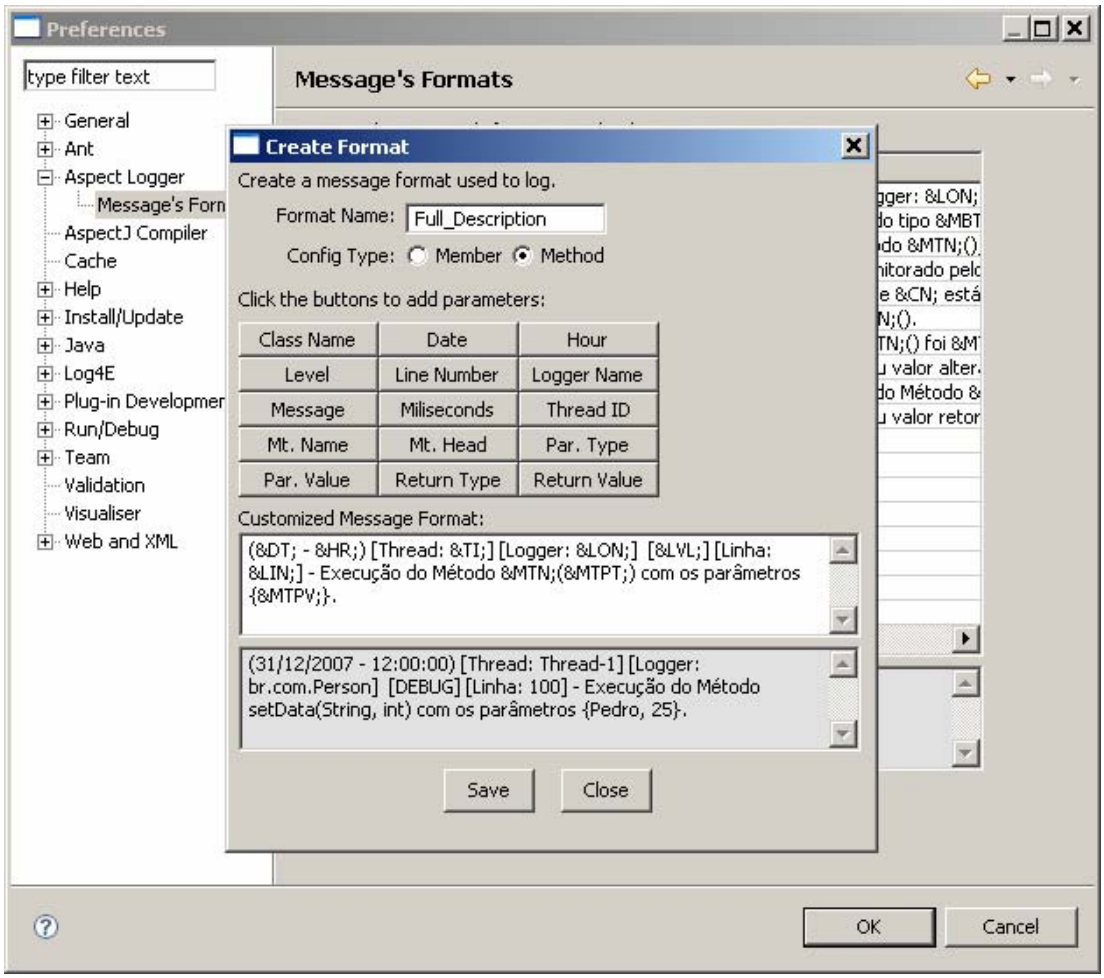

Figura 4.6 - Criação de um modelo de mensagem reutilizável.
Fonte: Elaborada pelo autor.

A fim de garantir um maior entendimento sobre o processo de criação de um modelo de mensagem reutilizável, é mostrado na Figura 4.7 o diagrama de seqüência da UML (BOOCH *et al.*, 1999) que ilustra a situação. Em 1, o usuário (`User`) cria uma configuração sobre a página `MessageFormatCreatePage`, para, posteriormente, em 2, solicitar a ela o salvamento da configuração (`saveConfig`), que só ocorre se o comando de guarda `validateFields` garantir que essa configuração está correta e completa. Então, em 2.1, a página repassa a solicitação à classe `MessageSupport`, através do método `setMessageFormatList`, sendo que essa classe é responsável pela persistência da configuração em um XML. Porém, antes disso acontecer, a classe `MessageSupport` solicita a atualização da lista de formatos de mensagem, em 2.1.1, por meio da chamada do método `setMessageFormats` da classe de entidade `MessageFormatList`.

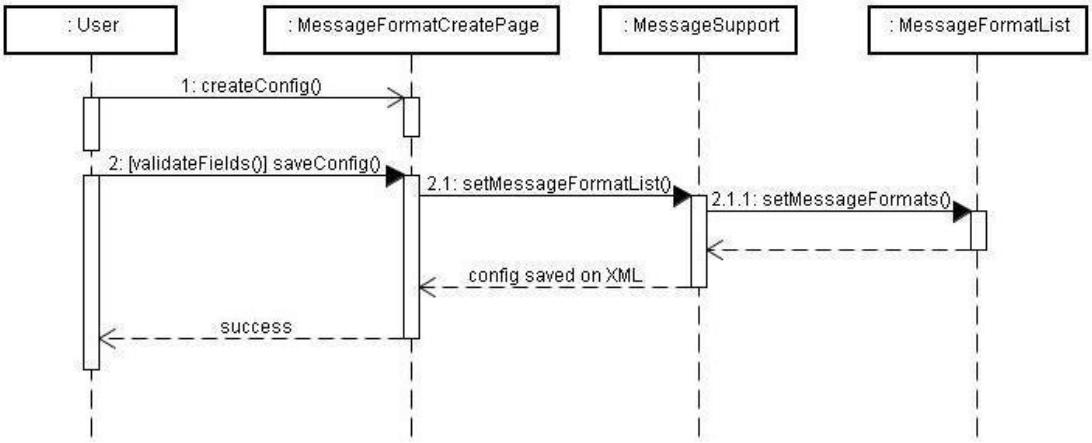

Figura 4.7 – Diagrama de seqüência da criação de um modelo de mensagem.
Fonte: Elaborada pelo autor.

Como mostrado no esquema de funcionamento do Asplog (Figura 4.3), a função desse mecanismo é dar suporte aos mecanismos de configuração de *logging*, oferecendo a eles uma lista de modelos reutilizáveis que podem ser usados para a criação de suas configurações.

Os recursos oferecidos por esse mecanismo permitem ganhos de produtividade e qualidade no trabalho do desenvolvedor, pois uma vez criado o modelo, ele pode ser reutilizado sempre que necessário em qualquer que seja o projeto, já que as configurações são vinculadas à IDE. Além disso, todo o processo de criação e edição é visual e flexível, podendo-se combinar uma boa quantidade de informações e texto livre.

4.2 Mecanismos de Configuração de *Loggers*

Em Log4j, o *logger* é a estrutura responsável por gerenciar requisições de mensagens de *logging*. É ele quem julga se um pedido é ou não procedente, verificando se o nível de tal requisição é maior ou igual ao seu nível. Se sim, ela é despachada para o destino vinculado a esse *logger*, denominado *appender*. Caso contrário, o pedido é simplesmente recusado.

O Asplog incorporou como parte da solução de seu projeto a estrutura *logger* definida por Log4j, sendo que seus conceitos e responsabilidades permanecem inalterados. Em geral, um *logger* precisará estar vinculado a cada configuração de *logging* criada para fazer o controle de requisições. Por essa razão, os mecanismos de configuração de *logging* são dependentes do mecanismo de configuração de *loggers*, como pode ser observado na Figura 4.3, que detalha o esquema de funcionamento do Asplog.

O mecanismo descrito nesta subseção tem a função de concentrar em um único local todos os *loggers* criados em um projeto, tornando-os mais fácil de

visualizar e configurar. O acesso a esse mecanismo é feito pela visão do Eclipse que permite acompanhar os projetos e suas respectivas estruturas (*Package Explorer*). A funcionalidade estará disponível sobre o projeto selecionado. Na Figura 4.8, é ilustrado o acesso a essa funcionalidade, a partir do projeto `Mercado`, através da opção "*Asplog → View Loggers*".

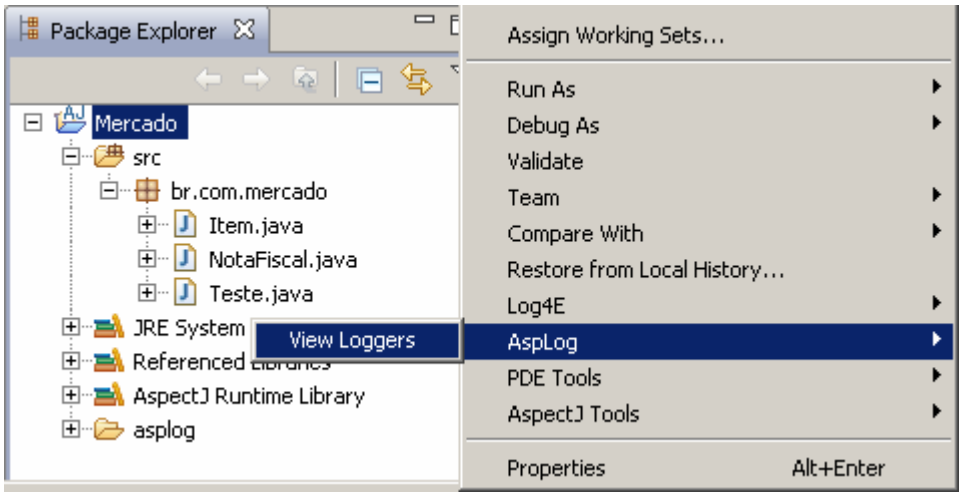

Figura 4.8 - Acesso ao mecanismo de configuração de *loggers*.
Fonte: Elaborada pelo autor.

Essa opção oferece uma interface gráfica que permite o gerenciamento dos *loggers* relacionados ao projeto. É apresentada, na Figura 4.9, a interface gráfica com os *loggers* que já fazem parte do projeto `Mercado`, e que em sua maioria foram criados a partir dos exemplos de configuração de *logging* que são mostrados na Subseção 4.3, mais especificamente, no agrupamento *Logger Configs* das interfaces mostradas nas Figura 4.12, Figura 4.15, Figura 4.17 e Figura 4.19.

No desenvolvimento dessa interface, foi criada uma tabela que possui os seguintes campos com informações sobre os *loggers*: nome (*name*), categoria (*category*), nível (*level*), saída no console (*cons.*), saída no arquivo (*file*) e nome do arquivo de saída (*filename*). O conceito de hierarquia de *loggers* definido por Log4j também foi adotado na visualização deles, usando um modelo de navegação em árvore sobre o campo nome. Só existirá um *logger* com aquele nome se todos os campos daquela linha estiverem preenchidos. Pode-se citar, como exemplo, o *logger* "ROOT", que sempre será ancestral de todos os outros, dentre eles, do *logger* "br.com.mercado.NotaFiscal", que por sua vez é pai do *logger* "br.com.mercado.NotaFiscal.adicionarItem(int, int)".

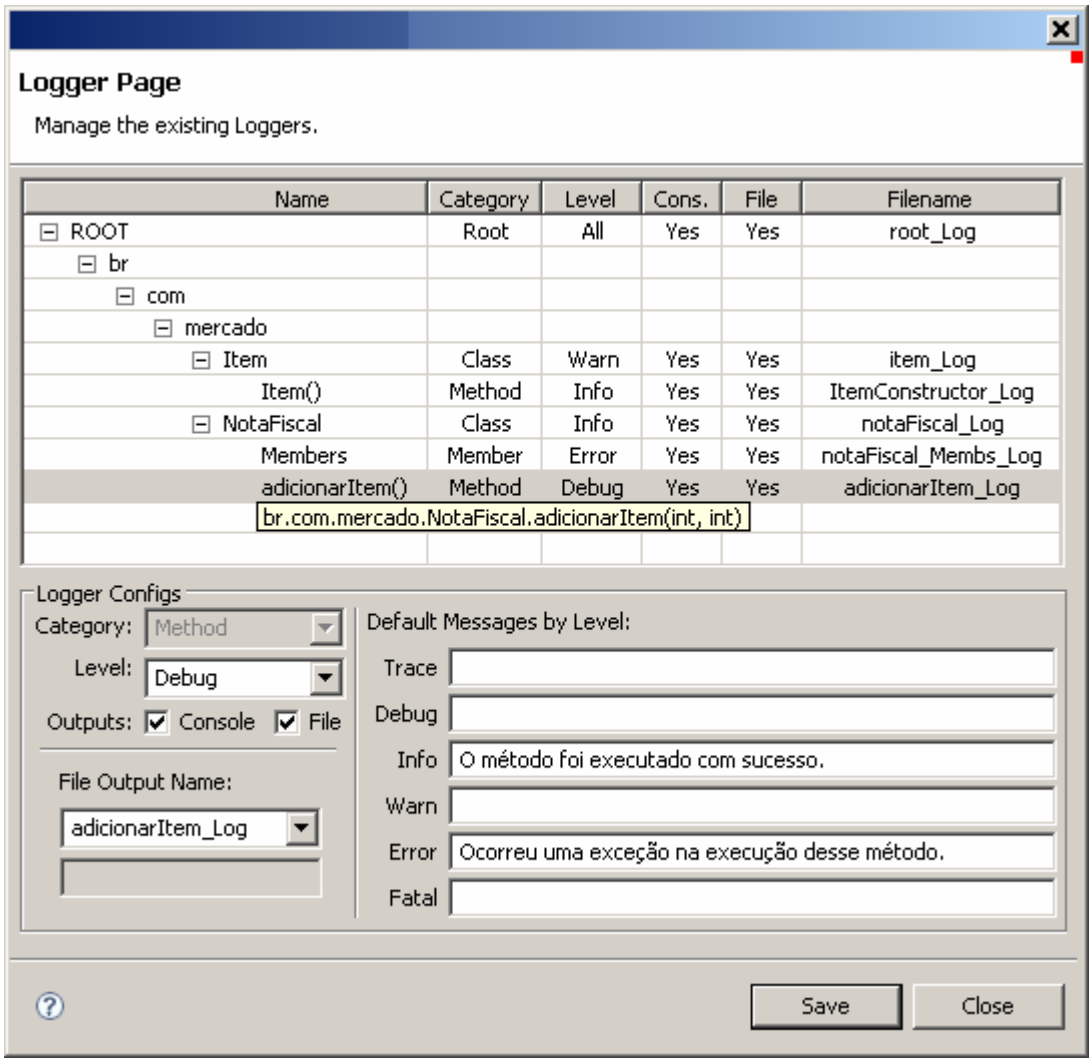

Figura 4.9 - Mecanismo de configuração de *loggers*.
Fonte: Elaborada pelo autor.

No Asplog, foram definidas quatro categorias para os *loggers* que podem ser criados e utilizados nas configurações de *logging*, são elas:

- ***Root* –** Indica o *logger* raiz do projeto. Seu nome sempre será “ROOT”.
- ***Class* –** Indica o *logger* vinculado à classe sobre a qual a configuração de *logging* está sendo feita. Seu nome será o nome totalmente qualificado daquela classe. Por exemplo, os *loggers* “br.com.mercado.Item” e “br.com.mercado.NotaFiscal”.
- ***Method* –** Indica o *logger* vinculado exclusivamente à configuração de *logging* de um único método. Seu nome será o nome totalmente qualificado da classe a qual esse método pertence, seguido de um ponto (‘.’) e de sua assinatura. Por exemplo, os *loggers* “br.com.mercado.Item.Item(java.lang.String, int, int)” e “br.com.mercado.NotaFiscal.adicionarItem(int, int)”.

- ***Member* –** Indica o *logger* vinculado exclusivamente à configuração de *logging* para membros de uma única classe. Seu nome será o nome totalmente qualificado da classe a qual esse método pertence, seguido de um ponto (‘.’) e da palavra “Members”. Por exemplo, o *logger* “br.com.mercado.NotaFiscal.Members”.

O nível de um *logger* diz respeito à relevância dos pedidos de *logging* que ele é capaz de tratar. As opções são aquelas oferecidas por Log4j: *trace*, *debug*, *info*, *warn*, *error* e *fatal*. E ainda há mais dois níveis, *all e off*, indicando que pedidos de todos os níveis serão aceitos ou nenhum, respectivamente.

Ainda nessa tabela, os campos saída no console e saída no arquivo dizem respeito aos locais onde o *logger* deve despachar suas requisições de *logging*, isto é, informam os tipos de *appenders* utilizados. Caso haja saída no arquivo, o campo nome do arquivo de saída deverá conter um valor.

É possível que cada *logger* tenha suas configurações editadas através do agrupamento *Logger Configs*. Com exceção do campo categoria, todos os outros podem ser alterados, tais como: nível, saída no console, saída no arquivo e nome do arquivo. Pode-se também definir mensagens associadas a um determinado nível de um *logger* (*Default Messages by Level*), que representa o tipo Mensagem (&ME;), descrito no Quadro 4.1, usado pelo mecanismo responsável pela criação de modelos de mensagem reutilizáveis de *logging* (Subseção 4.1).

Quando um *logger* é criado, o que sempre acontecerá através dos mecanismos de configuração de *logging*, ou editado usando o mecanismo descrito nessa seção, ele é salvo em um arquivo XML, cuja finalidade é armazenar a lista de todos os *loggers* de um projeto. Na seqüência, o mecanismo de geração de código (Subseção 4.4) é invocado para criar a classe responsável pela inicialização dos *loggers* usados durante a execução da aplicação.

Maiores detalhes sobre o processo de criação ou edição de um *logger* podem ser verificados no diagrama de seqüência ilustrado na Figura 4.10. Em 1, o usuário (`User`) cria uma configuração sobre a página `LoggerPage`, para, posteriormente, em 2, solicitar a ela o salvamento da configuração (`saveConfig`), que só ocorre se o comando de guarda `validateFields` garantir que essa configuração está correta e completa. Então, em 2.1, a página repassa a solicitação à classe `LoggerSupport`, através do método `setLoggerList`, sendo que essa classe é responsável pela persistência da configuração em um XML. Porém, antes disso acontecer, a classe `LoggerSupport` solicita a atualização da lista de *loggers*, em

2.1.1, por meio da chamada do método `setLoggers` da classe de entidade `LoggerList`. Já em 2.2, a página `LoggerPage` solicita a geração da classe de inicialização dos *loggers*, que se baseia na lista de *loggers* armazenada, através do método `createLoggerInitializerClass` da classe `Generator`.

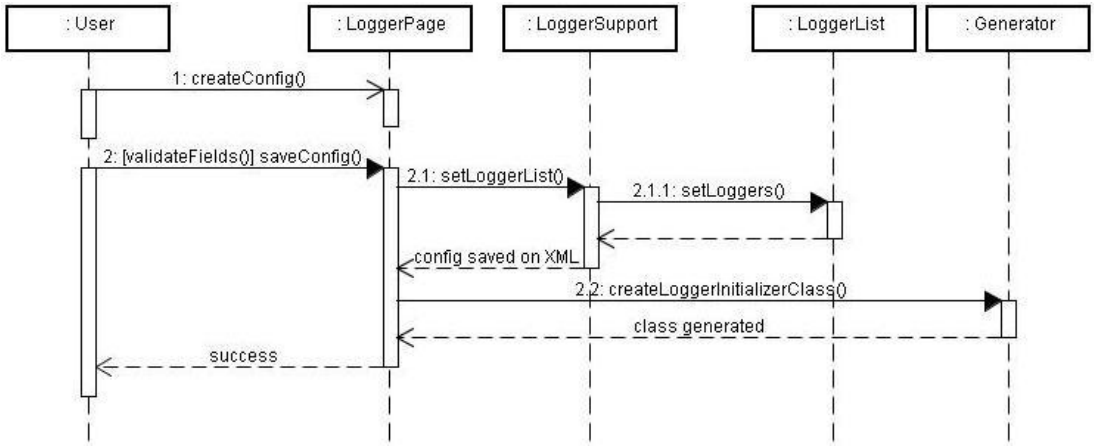

Figura 4.10 - Diagrama de seqüência da criação ou edição de um *logger*.
Fonte: Elaborada pelo autor.

Usando o mecanismo de configuração de *loggers*, o esforço de edição das configurações desses *loggers* é minimizado, já que são disponibilizados recursos que facilitam a visualização e entendimento deles de forma centralizada e hierarquizada. Por exemplo, o processo de ativação ou desativação de configurações de *logging* é muito simples e se resume a alterar, através desse mecanismo, o nível de seu *logger*. Por essas razões, essa funcionalidade agrega recursos indispensáveis para que Asplog atinja os objetivos propostos.

4.3 Mecanismos de Configuração de *Logging*

Dentre todos os mecanismos oferecidos pelo Asplog para ajudar na automatização do processo de *logging*, os que são descritos nesta subseção possuem o papel mais importante. É a partir deles que o desenvolvedor pode criar suas próprias configurações de *logging* responsáveis por definir como os aspectos, cuja finalidade é modularizar esse interesse, deverão ser criados pelo gerador de código.

No esquema de funcionamento do Asplog, mostrado na Figura 4.3, pode-se observar que esses mecanismos são dependentes do mecanismo de configuração de mensagens (Subseção 4.1) e do mecanismo de configuração de *loggers* (Subseção 4.2).

Os mecanismos encarregados de oferecer essa funcionalidade são quatro:

- Mecanismo de Configuração de *Logging* para Métodos;

- Mecanismo de Configuração de *Logging* Seletivo para Métodos;
- Mecanismo de Configuração de *Logging* por Filtro para Métodos;
- Mecanismo de Configuração de *Logging* Seletivo para Membros.

Nas subseções seguintes, é apresentado e explicado em detalhes o funcionamento de cada um desses mecanismos. Nelas, quando citada a palavra "métodos", os construtores devem ser considerados, a não ser que explicitamente citado o contrário.

4.3.1 Mecanismo de Configuração de *Logging* para Métodos

Ao se planejar essa funcionalidade, a intenção foi permitir que o desenvolvedor pudesse escolher e configurar visualmente o método de uma classe em que houvesse o interesse de se monitorar por *logging*.

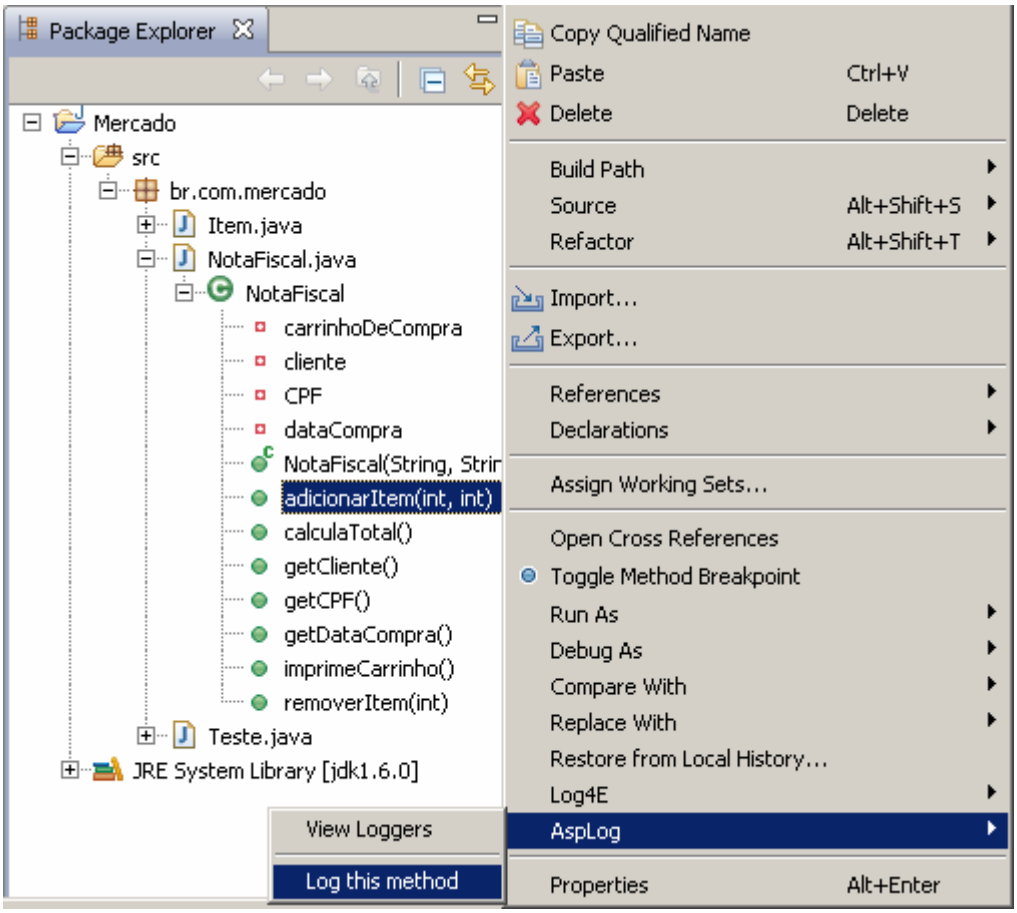

Figura 4.11 - Acesso ao mecanismo de configuração de *logging* para métodos.
Fonte: Elaborada pelo autor.

Para a escolha do método, deve-se expandir a visão de navegação da estrutura do projeto (*Package Explorer*) no Eclipse até o nível estrutural de uma classe e selecionar o método desejado. Essa funcionalidade estará disponível para ser

utilizada sobre o método. Na Figura 4.11, é ilustrado o acesso a essa funcionalidade para o método `adicionarItem` da classe `NotaFiscal`, a partir da opção do menu "*Asplog → Log this method*".

Assim que essa opção é solicitada, será aberta uma interface gráfica que permite a criação da configuração de *logging* desejada para o método selecionado. Dando continuidade ao exemplo, a Figura 4.12 mostra o que seria exibido para o método `adicionarItem`, com uma possível configuração.

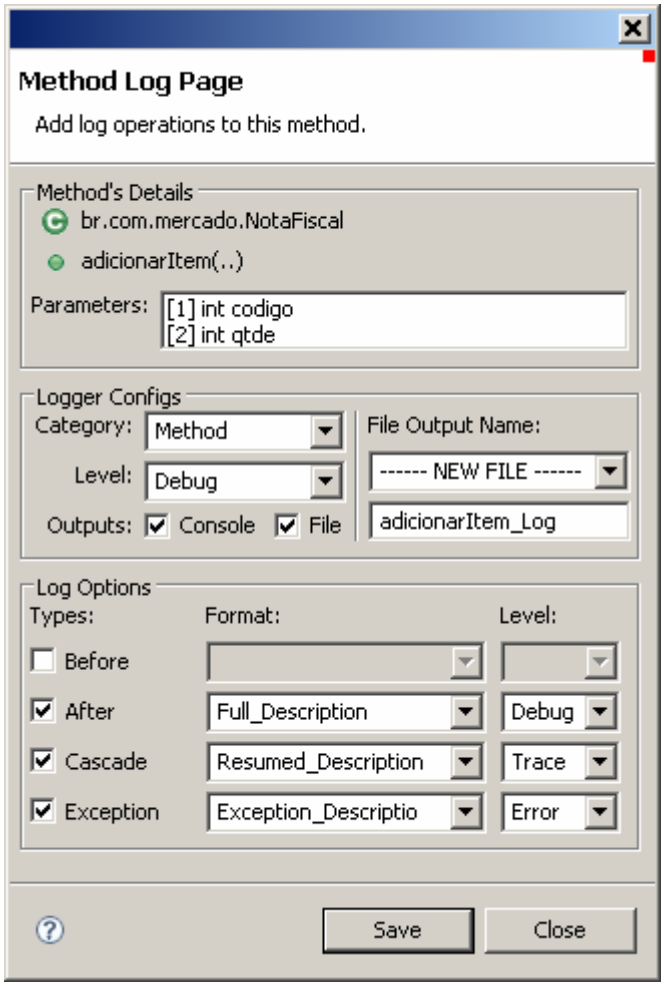

Figura 4.12 - Mecanismo de configuração de *logging* para métodos.
Fonte: Elaborada pelo autor.

Nessa interface, é exibido no agrupamento *Method's Details* alguns detalhes de caráter apenas informativo sobre o método selecionado para situar o desenvolvedor sobre a sua escolha. Dentre as informações presentes sobre o método, estão: seu nome, visibilidade e modificadores; tipo e nome dos seus parâmetros (*parameters*); e o nome, visibilidade e modificadores da classe a qual pertence.

O agrupamento *Logger Configs* vem logo a seguir. Explicações detalhadas sobre cada uma das opções desse agrupamento podem ser obtidas na Subseção 4.2. É nele que se configura o *logger* responsável por gerenciar o controle das requisições de *logging*, assim como o nível e os destinos da saída desse *logger*, que podem ser o console ou um arquivo. Para esse último, é necessário informar o nome do arquivo.

Por fim, temos o importante agrupamento *Log Options*. A partir de suas configurações, são criados os conjuntos de junção e adendos que fazem parte dos aspectos responsáveis pela modularização do interesse transversal de *logging*. Os quatro tipos de opções (*type*) desse agrupamento são:

- ***Before* –** Indica a criação de um adendo responsável por monitorar o momento anterior à execução do método em questão.
- ***After* –** Indica a criação de um adendo responsável por monitorar o momento posterior à execução com sucesso do método em questão.
- ***Cascade* –** Indica a criação de um adendo responsável por monitorar o momento anterior à execução do método em questão, assim como a de todos os outros métodos chamados a partir dele. Esse processo é recursivo enquanto houver métodos em execução.
- ***Exception* –** Indica a criação de um adendo responsável por monitorar o momento posterior à execução sem sucesso de um método, isto é, quando uma exceção for lançada.

Para cada uma dessas opções que for utilizada, deve ser atribuído um formato (*format*) e um nível (*level*). O formato está relacionado com a formatação e as informações que farão parte de cada registro de *logging*, cuja finalidade é retratar o comportamento da aplicação. Dentre as opções de formato, estarão aquelas criadas a partir do mecanismo de configuração de mensagens descrito na Subseção 4.1. Já o nível diz respeito à relevância daquela opção dentre aquelas oferecidas pelo Log4j: *trace*, *debug*, *info*, *warn*, *error* e *fatal*. A opção só estará habilitada se o seu nível for superior ou igual ao do *logger* da configuração. Esse esquema é uma espécie de chaveamento.

Após o término da criação de uma configuração, ela pode ser salva. Isso acontece em um arquivo XML responsável por armazenar a lista com todas as configurações já criadas. Logo depois, o gerador de código é acionado para a criação do aspecto referente a ela. Esse mecanismo é detalhado na Subseção 4.4.

O processo de criação ou edição de uma configuração de *logging* para métodos é mostrado em detalhes no diagrama de seqüência da Figura 4.13. Como o funcionamento dos outros três mecanismos de configuração de *logging*, mostrados nas subseções 4.3.2, 4.3.3 e 4.3.4, são muito similares a esse, não serão apresentados os seus respectivos diagramas de seqüência.

No passo 1 desse diagrama de seqüência, o usuário (`User`) cria uma configuração sobre a página `MethodPage`, para, posteriormente, em 2, solicitar a ela o salvamento da configuração (`saveConfig`), que só ocorre se o comando de guarda `validateFields` garantir que essa configuração está correta e completa. Então, em 2.1, a página repassa a solicitação à classe `LoggingSupport`, através do método `setConfigList`, sendo que essa classe é responsável pela persistência da configuração em um XML. Porém, antes disso acontecer, a classe `LoggingSupport` solicita a atualização da lista de configurações de *logging*, em 2.1.1, por meio da chamada do método `setMethodConfigs` da classe de entidade `ConfigList`. Já em 2.2, a página `MethodPage` solicita a geração do aspecto de *logging*, que se baseia na configuração que foi salva, através do método `createLoggingAspect` da classe `Generator`.

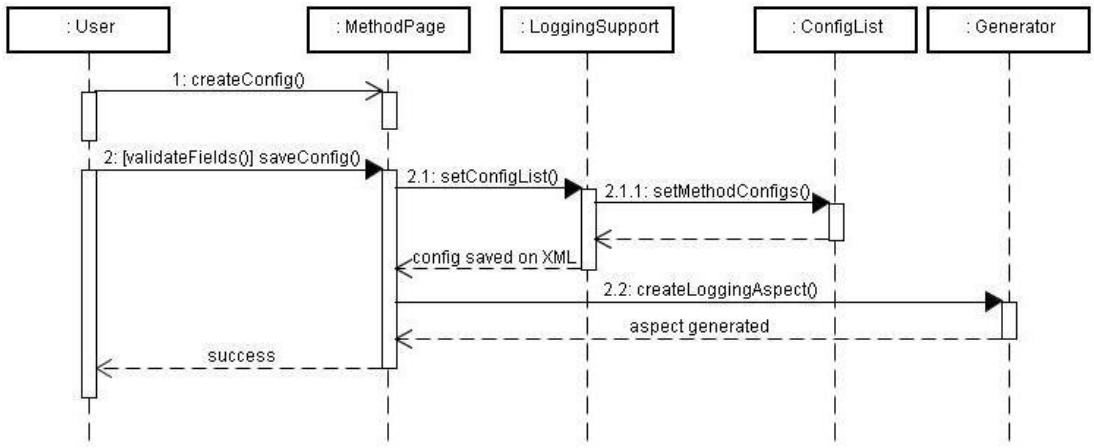

Figura 4.13 - Diagrama de seqüência da criação ou edição de uma configuração de *logging* para métodos.
Fonte: Elaborada pelo autor.

4.3.2 Mecanismo de Configuração de *Logging* Seletivo para Métodos

Esse mecanismo é praticamente uma extensão do anterior. A novidade fica por conta de permitir que não apenas um, mas que vários métodos de uma classe possam ser selecionados e agrupados para observação a partir da mesma configuração, simplificando e agilizando o trabalho do desenvolvedor.

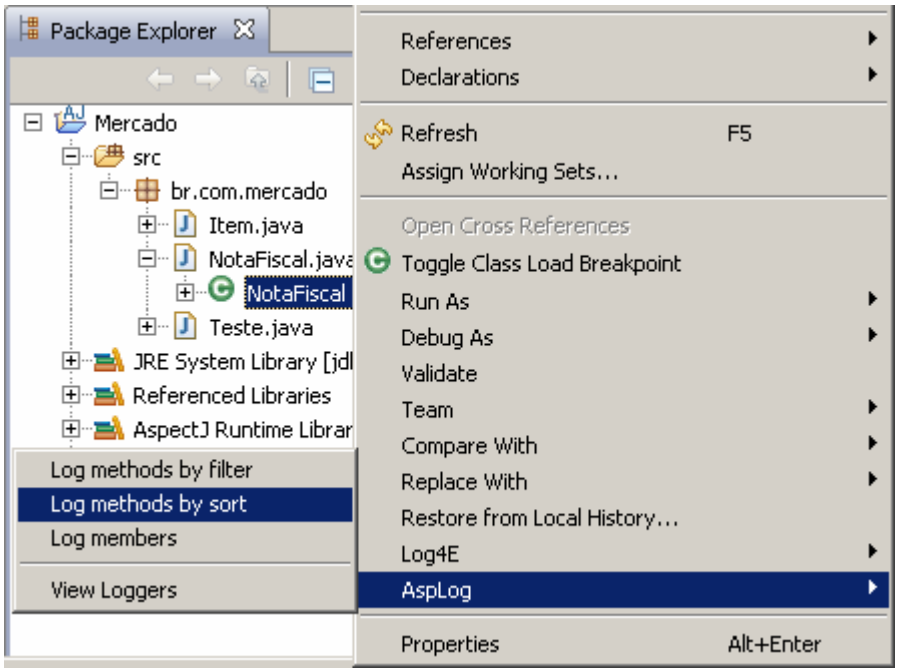

Figura 4.14 - Acesso ao mecanismo de configuração de *logging* seletivo para métodos.
Fonte: Elaborada pelo autor.

Para acessar essa funcionalidade, é necessário expandir a visão de navegação da estrutura do projeto (*Package Explorer*) até o nível da classe desejada. Essa funcionalidade estará disponível para ser utilizada sobre a classe. Na Figura 4.14, é ilustrado o acesso a essa funcionalidade para a classe `NotaFiscal`, a partir da opção do menu "*Asplog → Log methods by sort*".

Depois que essa opção é acionada, uma interface gráfica será exibida para que uma configuração de *logging* seletiva possa ser criada para os métodos da classe em questão. A Figura 4.15 ilustra essa interface, de acordo com a classe `NotaFiscal`. Nela, uma possibilidade de configuração é apresentada.

O agrupamento *Class' Details* apresenta alguns detalhes sobre a classe selecionada, tais como: nome, visibilidade e modificadores. Seu caráter é apenas informativo.

Através desse mecanismo, várias configurações diferentes podem ser criadas para a mesma classe. Para isso, existe o agrupamento *Config Options*. Nele, é possível escolher a configuração a qual se quer editar. Ou, se necessário for, criar uma nova configuração, que se diferenciam pelo nome único.

Os agrupamentos *Logger Configs* e *Log Options* são os mesmos em relação ao mecanismo descrito na Subseção 4.3.1. Portanto, não será necessário explicá-los

novamente. A novidade desse mecanismo está no agrupamento *Methods Selection*. Ele possui duas tabelas que controlam todos os métodos da classe selecionada, que farão ou não parte da configuração. A tabela da esquerda (*methods to use*) lista todos os métodos que não fazem parte da configuração. Enquanto isso, a tabela da direita (*methods in use*) lista os que foram adicionados à configuração. A transferência dos métodos entre as tabelas é facilitada pelo uso de dois botões (*left* e *right*), o que torna esse processo rápido.

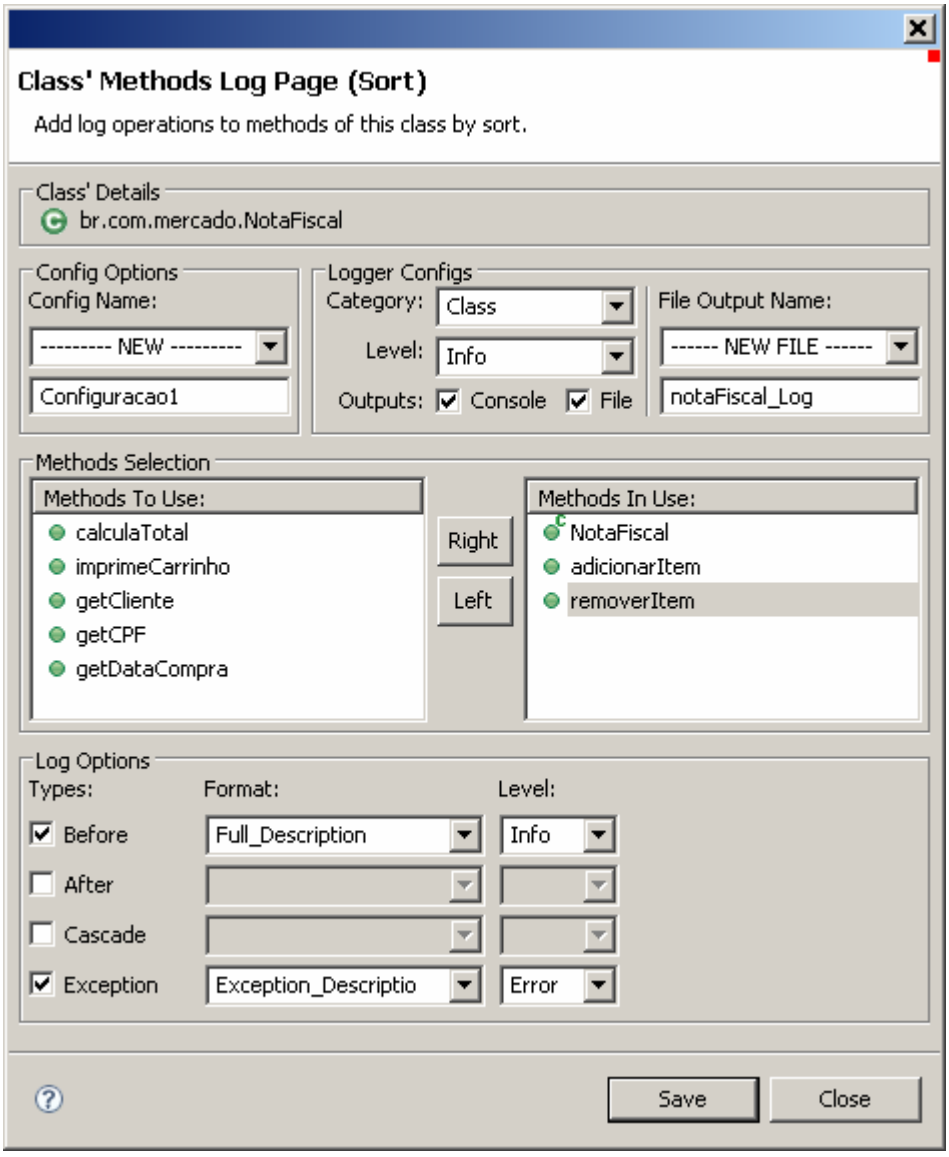

Figura 4.15 - Mecanismo de configuração de *logging* seletivo para métodos.
Fonte: Elaborada pelo autor.

Depois que o preenchimento de todas as opções for concluído e o salvamento solicitado, a configuração é armazenada em um arquivo XML, juntamente com as outras configurações existentes. Por fim, é acionado o gerador de código responsável pela criação do aspecto correspondente a essa configuração.

4.3.3 Mecanismo de Configuração de *Logging* por Filtro para Métodos

A diferença entre esse mecanismo e o que foi descrito na Subseção 4.3.2 é que, ao invés de se selecionar um a um os métodos que farão ou não parte da configuração, os métodos são selecionados a partir de um filtro com algumas opções.

Para acessar essa funcionalidade, é necessário expandir a visão de navegação da estrutura do projeto (*Package Explorer*) até o nível da classe desejada. Essa funcionalidade estará disponível para ser utilizada sobre a classe. Na Figura 4.16, é ilustrado o acesso a essa funcionalidade para a classe `NotaFiscal`, a partir da opção do menu "*Asplog → Log methods by filter*".

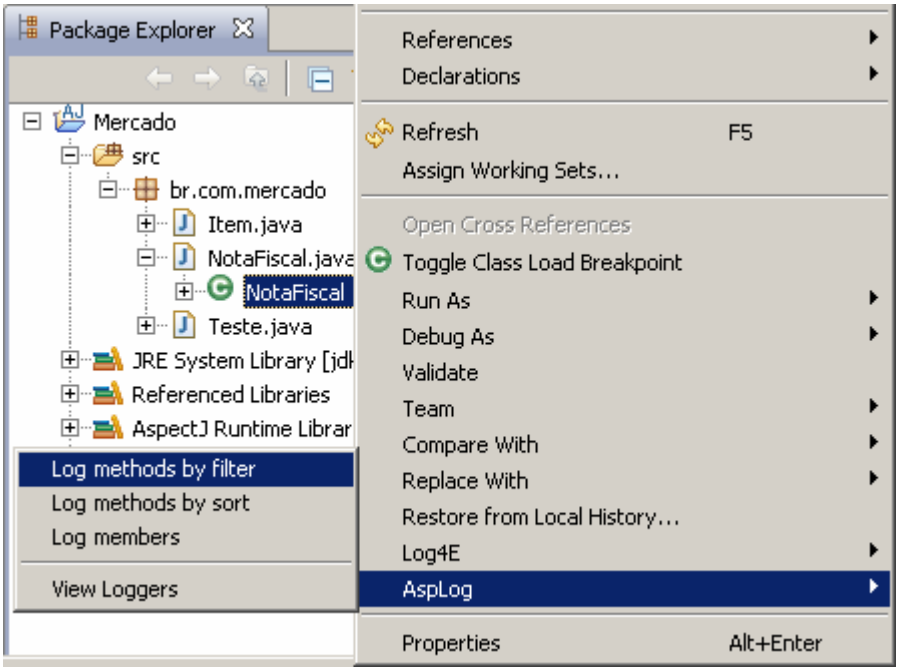

Figura 4.16 - Acesso ao mecanismo de configuração de *logging* por filtro para métodos.
Fonte: Elaborada pelo autor.

Essa opção disponibiliza uma interface gráfica que permite a criação de uma configuração de *logging* por filtro para acompanhar o comportamento dos métodos da classe desejada. A seguir, na Figura 4.17, essa interface pode ser vista carregada com informações da classe `NotaFiscal`, juntamente com uma dada configuração.

Com exceção do agrupamento *Methods Selection Filter*, todos os outros são iguais aos já descritos nos mecanismos da Subseção 4.3.1 e da Subseção 4.3.2. Portanto, somente esse é descrito aqui.

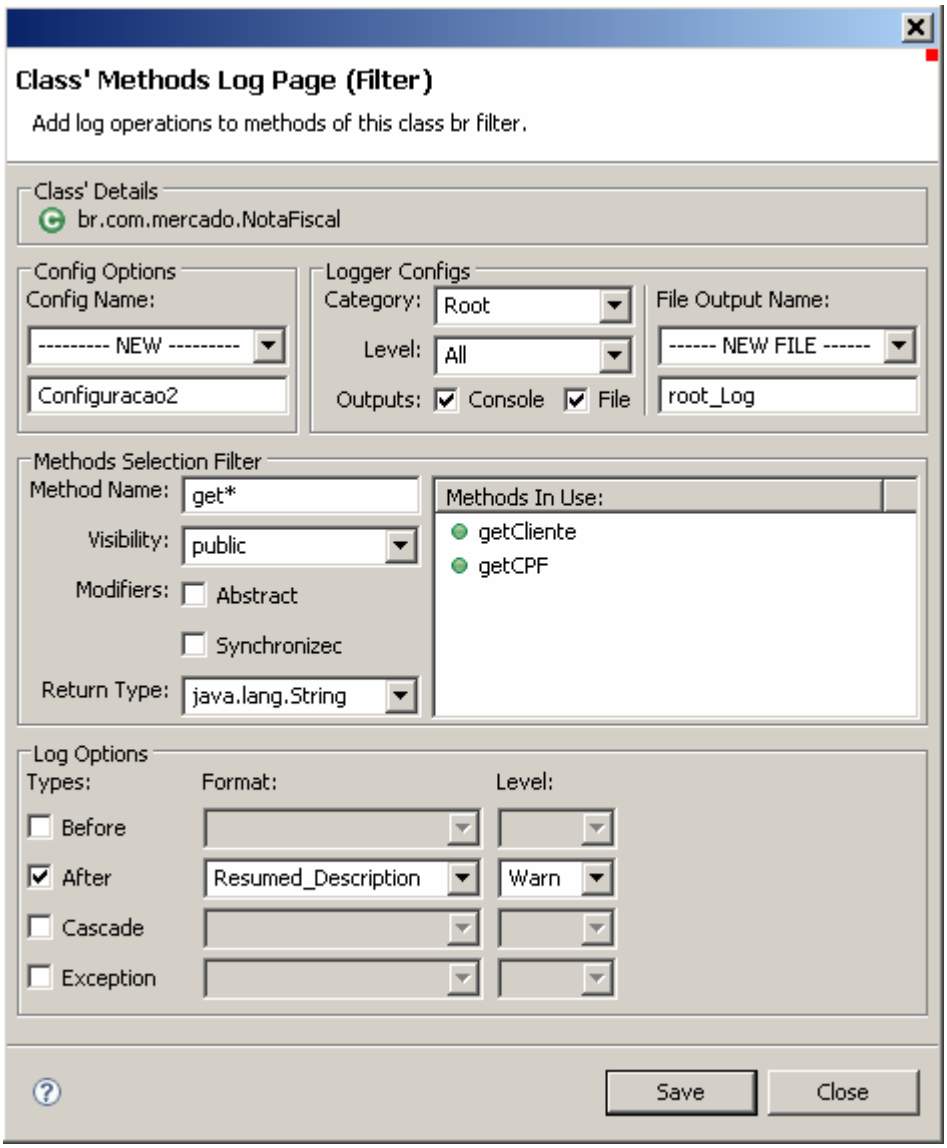

Figura 4.17 - Mecanismo de configuração de *logging* por filtro para métodos.
Fonte: Elaborada pelo autor.

O agrupamento *Methods Selection Filter* permite que seja criado um filtro para definir quais métodos farão parte da configuração. Esse recurso foi inspirado na flexibilidade que AspectJ oferece para a construção de pontos de junção através do uso de curingas (*wilcards*). As opções de filtro são:

- ***Method Name*** **–** Permite filtrar os métodos pelo nome. O operador '`*`' pode ser usado como curinga e representa "qualquer coisa".
- ***Visibility*** **–** Permite filtrar os métodos a partir de sua visibilidade: `public`, `protected` ou `private`.
- ***Modifiers*** **–** Permite filtrar somente os métodos que possuem os modificadores `Astract` e/ou `Synchronized`. Construtores não possuem esses modificadores.

- ***Return Type* –** Permite filtrar somente os métodos que possuem determinado tipo de retorno. Construtores não possuem retorno.

Para definir as opções do filtro desse agrupamento, buscou-se priorizar os recursos que poderiam ter maior relevância para o tratamento de *logging*. Porém, nada impede que, futuramente, essas opções sejam estendidas. Como por exemplo, permitindo que os métodos possam ser filtrados pelos tipos de seus parâmetros.

Assim que finalizada a criação da configuração, ela pode ser salva no arquivo XML responsável pelas configurações. Depois disso, o gerador de código é chamado automaticamente para produzir o aspecto referente a ela.

4.3.4 Mecanismo de Configuração de *Logging* Seletivo para Membros

O objetivo desse mecanismo é simplificar a criação de configurações que auxiliem na criação de estruturas que monitorem por *logging* os membros de uma classe. A intenção foi a de tornar todo o processo bem intuitivo e simples, através de configurações visuais.

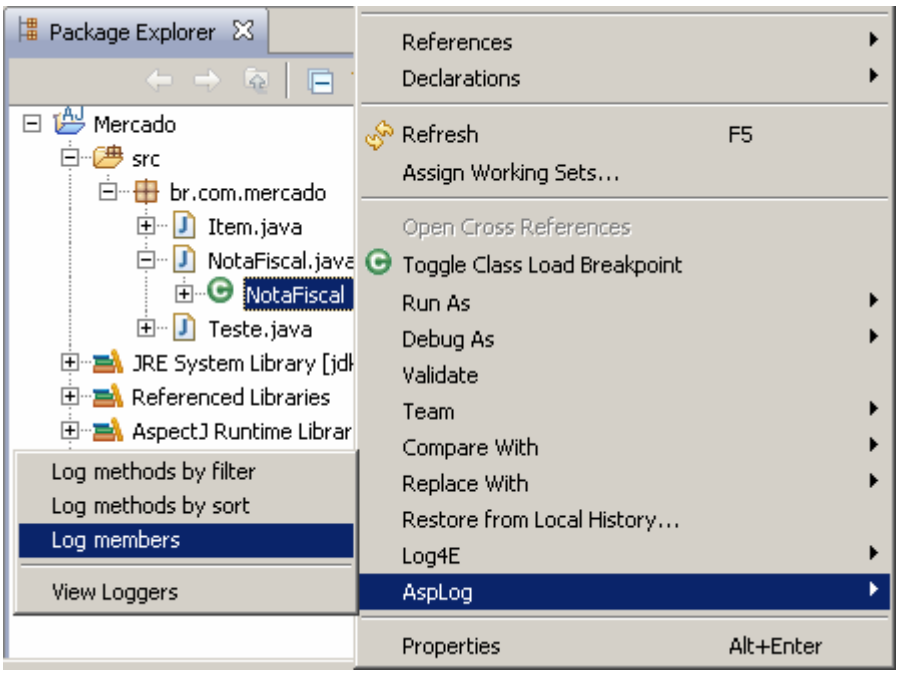

Figura 4.18 - Acesso ao mecanismo de configuração de *logging* seletivo para membros.
Fonte: Elaborada pelo autor.

Para acessar essa funcionalidade, é necessário expandir a visão de navegação da estrutura do projeto (*Package Explorer*) até o nível da classe desejada. Essa funcionalidade estará disponível para ser utilizada sobre a classe. Na Figura 4.18, é

ilustrado o acesso a essa funcionalidade para a classe `NotaFiscal`, a partir da opção do menu "*Asplog → Log members*".

Assim que é acionada, essa opção disponibiliza uma interface gráfica que deve ser usada pelo desenvolvedor na criação de uma configuração de *logging* para membros a fim de monitorar o estado de uma classe. Essa interface é mostrada na Figura 4.19, contendo dados relativos à classe `NotaFiscal` e o esboço de uma possível configuração

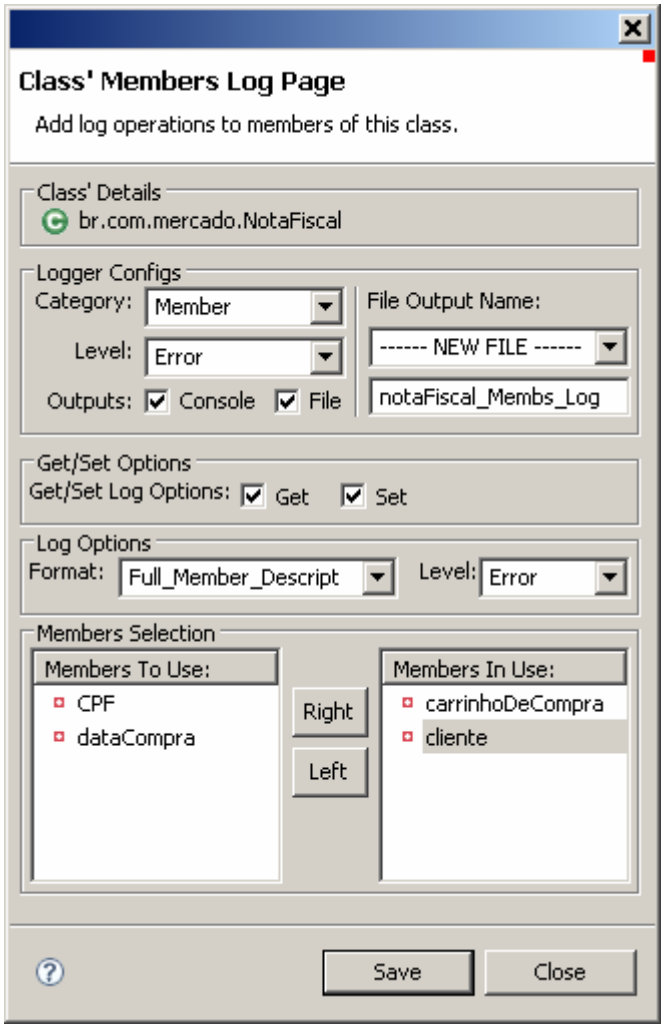

Figura 4.19 - Mecanismo de configuração de *logging* seletivo para membros.
Fonte: Elaborada pelo autor.

Os agrupamentos *Class' Details* e *Logger Configs* já foram descritos nos mecanismos da Subseção 4.3.2 e da Subseção 4.3.1, respectivamente. Portanto, eles não são descritos novamente.

É o agrupamento *Get/Set Options* que indica quando os membros devem ser monitorados. São dadas duas opções:

- ***Get* –** Permite monitorar um determinado membro sempre que a leitura do seu valor for solicitada.
- ***Set* –** Permite monitorar um determinado membro sempre que o seu valor for alterado, ou seja, quando houver uma atribuição.

O formato (*format*) e nível (*level*) das mensagens de *logging* a serem usadas também podem ser configurados através do agrupamento *Log Options*. Dentre as opções de formato, estarão aquelas que foram criadas a partir do mecanismo descrito na Subseção 4.1. O formato dita a formatação e informações contidas na mensagem. Enquanto isso, as opções de nível são aquelas disponibilizadas pelo Log4j. Se o nível escolhido for maior ou igual ao do *logger*, a configuração estará habilitada.

Por fim, tem-se o agrupamento *Members Selection*. A partir dele são selecionados os membros que farão parte da configuração. Nesse agrupamento, existem duas tabelas. A primeira (*members to use*) lista os membros da classe que não pertencerão à configuração. Enquanto isso, a segunda tabela (*members in use*) lista os membros que estarão presentes na configuração. O processo de escolha dos membros é totalmente dinâmico com o uso de dois botões (*left* e *right*) que permitem intercambiar os membros entre as tabelas. Não é possível observar o comportamento dos membros com o modificador `final`, já que AspectJ não permite. Isso acontece porque o valor desses membros não pode ser alterado depois de inicialmente atribuído, fato que só pode ocorrer no bloco de inicialização ou no construtor do objeto.

Ao término de todas as escolhas, a configuração pode ser salva e armazenada em um arquivo XML, juntamente com as outras configurações. Logo depois, o gerador de código é acionado para a criação do aspecto correspondente.

4.4 Mecanismo de Geração de Código

Enquanto que os outros três mecanismos descritos até aqui têm a função de criar as configurações relacionadas à *logging* e armazená-las em arquivos XML, o gerador de código é responsável por fazer a leitura desses arquivos, a fim de concretizar essas configurações, como pode ser observado na Figura 4.3, que detalha o esquema de funcionamento do Asplog. Isso é obtido através da criação de aspectos que entrecortarão as classes do sistema e de uma classe cuja finalidade é inicializar todos os *loggers* do projeto que serão utilizados para fazer o controle das requisições de *logging*.

Nas subseções seguintes, é apresentada em detalhes a relação de cada um dos outros três mecanismos com o mecanismo de geração de código.

4.4.1 Relação com o Mecanismo de Configuração de Mensagens

O mecanismo de configuração de mensagens produz configurações que dizem respeito à formatação e informações que farão parte de uma mensagem de *logging*, através da criação de modelos reutilizáveis. No momento de criação dos aspectos que modularizarão o *logging*, esses modelos são utilizados dentro dos adendos para caracterizar o *layout* das mensagens. Porém, é necessário haver uma tradução de cada caractere de conversão (código) utilizado no modelo para uma forma legível pelo aspecto, como mostrada no Quadro 4.2.

Código	Tradução	Fonte da Informação
&CN;	Nome da classe	Arquivo XML de configurações de *logging*.
&DT;	%d{dd/MM/yyyy}	Caractere de conversão de Log4j.
&HR;	%d{HH:mm:ss}	Caractere de conversão de Log4j.
&LVL;	%p	Caractere de conversão de Log4j.
&LIN;	thisJoinPointStaticPart. getSourceLocation().getLine()	Objeto `thisJoinPointStaticPart` de AspectJ.
&LON;	Nome do *logger*	Arquivo XML de configurações de *logging*.
&ME;	%m	Caractere de conversão de Log4j.
&MI;	%r	Caractere de conversão de Log4j.
&TI;	%t	Caractere de conversão de Log4j.
&MTN;	thisJoinPoint.getSignature(). getName()	Objeto `thisJoinPoint` de AspectJ.
&MTH;	thisJoinPoint.getSignature(). toLongString()	Objeto `thisJoinPoint` de AspectJ.
&MTPT;	Formatter.getArgsType(thisJoinPoint.getArgs())	Classe `Formatter` de suporte do Asplog mais Objeto thisJoinPoint de AspectJ.
& MTPV;	Formatter.getArgsValue(thisJoinPoint.getArgs())	Classe `Formatter` de suporte do Asplog mais Objeto thisJoinPoint de AspectJ.
&MTRT;	Formatter.getReturnType(o)	Classe `Formatter` de suporte do Asplog mais parâmetro do adendo do aspecto.
&MTRV;	Formatter.getReturnValue(o)	Classe `Formatter` de suporte do Asplog mais parâmetro do adendo do aspecto.
&MBN;	thisJoinPoint.getSignature(). getName()	Objeto `thisJoinPoint` de AspectJ.
&MBT;	Formatter.getReturnType(o)	Classe `Formatter` de suporte do Asplog mais parâmetro do adendo do aspecto.
&MBV;	Formatter.getReturnValue(o)	Classe `Formatter` de suporte do Asplog mais parâmetro do adendo do aspecto.

Quadro 4.2 - Tradução dos caracteres de conversão pelo gerador.
Fonte: Elaborado pelo Autor

Para exemplificar como o gerador interpreta um modelo de mensagens usado dentro de um adendo, suponha o seguinte modelo:

```
(&DT; - &HR;) [Thread: &TI;] [Logger: &LON;] [&LVL;] [Linha:
&LIN;] - Método &MTN;(&MTPT;) com os parâmetros {&MTPV;}
```

O resultado da interpretação pelo gerador seria a String abaixo:

```
"(%d{dd/MM/yyyy} - %d{HH:mm:ss}) [Thread: %t] [Logger: ROOT] [%p]
[Linha: " + thisJoinPointStaticPart.getSourceLocation().getLine() +
"] - Método " + thisJoinPoint.getSignature().getName() + "(" +
Formatter.getArgsType(thisJoinPoint.getArgs()) + ") com os
parâmetros {" + Formatter.getArgsValue(thisJoinPoint.getArgs()) +
"}\n";
```

4.4.2 Relação com o Mecanismo de Configuração de *Loggers*

Para que uma requisição de *logging* seja executada, é necessário fazer esse pedido ao *logger* responsável. Antes disso, porém, esse *logger* tem que ser inicializado, isto é, precisa ser carregado com as configurações definidas para ele na interface gráfica do mecanismo de configuração de *loggers*, mostrada na Figura 4.9. Para isso, o gerador de código do Asplog cria uma classe com a finalidade de inicializar todos os *loggers* de um projeto. Na Listagem 4.4, é mostrada a estrutura dessa classe, denominada `LoggerInitializer`. Os elementos variáveis dessa classe, que estão entre colchetes, podem ser acompanhados no Quadro 4.3. Todos esses elementos variáveis estão relacionados às configurações definidas para cada *logger* na interface.

```
1  public class LoggerInitializer extends LoggerManager {
2    public static LoggerInitializer GET = new LoggerInitializer();
3
4    public LoggerInitializer() {
5      super();
6    }
7
8    public void initializeLoggers() {
9      initializeLogger([NOME_LOGGER_1], Level.[NIVEL_1], [CONSOLE_1],
10         [ARQUIVO_1], [NOME_ARQUIVO_1], [MSG_TRACE_1], [MSG_DEBUG_1],
11         [MSG_INFO_1],[MSG_WARN_1], [MSG_ERROR_1],[MSG_FATAL_1]);
12                                  ...
13     initializeLogger([NOME_LOGGER_N], Level.[NIVEL_N], [CONSOLE_N],
14         [ARQUIVO_N], [NOME_ARQUIVO_N], [MSG_TRACE_N], [MSG_DEBUG_N],
15         [MSG_INFO_N],[MSG_WARN_N], [MSG_ERROR_N],[MSG_FATAL_N]);
16   }
17 }
```

Listagem 4.4 - Estrutura da classe de inicialização dos *loggers*.
Fonte: Elaborada pelo autor.

Código	Descrição	Tipo
NOME_LOGGER	Nome do *logger*.	`String`
NIVEL	Nível das requisições de *logging* que o *logger* é capaz de tratar.	`Level` (Log4j)
CONSOLE	Informa se haverá *appender* vinculado ao console.	`boolean`
ARQUIVO	Informa se haverá *appender* vinculado a um arquivo.	`boolean`
NOME_ARQUIVO	Se houver *appender* vinculado a um arquivo, informa seu nome.	`String`
MSG_TRACE	Mensagem associada ao nível *trace* de um *logger*.	`String`
MSG_DEBUG	Mensagem associada ao nível *debug* de um *logger*.	`String`
MSG_INFO	Mensagem associada ao nível *info* de um *logger*.	`String`
MSG_WARN	Mensagem associada ao nível *warn* de um *logger*.	`String`
MSG_ERROR	Mensagem associada ao nível *error* de um *logger*.	`String`
MSG_FATAL	Mensagem associada ao nível *fatal* de um *logger*.	`String`

Quadro 4.3 - Elementos variáveis da classe `LoggerInitializer`.
Fonte: Elaborado pelo Autor

Essa classe herda da classe abstrata `LoggerManager`, cuja finalidade é gerenciar os *loggers* de um projeto, e que foi desenvolvida dentro de uma biblioteca de suporte do Asplog. A classe `LoggerManager` controla a inicialização dos l*oggers*, a busca por eles e as requisições de *logging* feitas a eles. Para a inicialização dos *loggers*, a classe `LoggerManager` define o método abstrato `initializeLoggers`, que é implementado pela classe `LoggerInitializer` (linhas 8 a 16), sendo que nele, cada *logger* é criado através da chamada do método `initializeLogger` (linhas 9 a 15), onde os parâmetros de configuração são passados.

4.4.3 Relação com o Mecanismo de Configuração de *Logging*

O produto final de uma configuração criada a partir dos mecanismos de configuração de *logging* (Subseção 4.3) resulta em um aspecto criado pelo gerador de código. Esse aspecto entrecortará as classes do sistema e modularizará o interesse transversal de *logging*.

A estrutura geral do aspecto que será gerado por cada um dos quatro mecanismos de configuração de *logging* é descrito nas subseções seguintes.

4.4.3.1 Relação com o Mecanismo de Configuração de *Logging* para Métodos

Na Listagem 4.5, é apresentada a estrutura do aspecto que será gerado para o monitoramento de um método qualquer. Se um construtor fosse escolhido, a estrutura seria um pouco diferente, porém ela não será abordada, já que a intenção principal é mostrar o papel do gerador.

```
1  public aspect [ASPECTO] {
2    private static Logger [CLASSE].[MEMBRO_LOGGER] =
3            LoggerInitializer.GET.getLogger([NOME_LOGGER]);
4
5    private pointcut methodExecution() :
6      execution(* [CLASSE].[METODO]([ASSINATURA_METODO]));
7    private pointcut methodCascade(): cflow(methodExecution()) &&
8      (execution(* *(..)) || execution(new(..)));
9
10   before() : methodExecution() {
11     if([CLASSE].[MEMBRO_LOGGER].isEnabledFor(Level.[NIVEL])) {
12       String customizedMessage = [MENSAGEM_CUSTOMIZADA];
13       PatternLayout layout = new PatternLayout(customizedMessage);
14       LoggerInitializer.GET.[NIVEL]([CLASSE].[MEMBRO_LOGGER], layout);
15     }
16   }
17   after() returning(Object o): methodExecution() {
18     if([CLASSE].[MEMBRO_LOGGER].isEnabledFor(Level.[NIVEL])) {
19       String customizedMessage = [MENSAGEM_CUSTOMIZADA];;
20       PatternLayout layout = new PatternLayout(customizedMessage);
21       LoggerInitializer.GET.[NIVEL]([CLASSE].[MEMBRO_LOGGER], layout);
22     }
23   }
24   before() : methodCascade() {
25     if([CLASSE].[MEMBRO_LOGGER].isEnabledFor(Level.[NIVEL])) {
26       String customizedMessage = [MENSAGEM_CUSTOMIZADA];
27       PatternLayout layout = new PatternLayout(customizedMessage);
28       LoggerInitializer.GET.[NIVEL]([CLASSE].[MEMBRO_LOGGER], layout);
29     }
30   }
31   after() throwing: methodExecution() {
32     if([CLASSE].[MEMBRO_LOGGER].isEnabledFor(Level.[NIVEL])) {
33       String customizedMessage = [MENSAGEM_CUSTOMIZADA];
34       PatternLayout layout = new PatternLayout(customizedMessage);
35       LoggerInitializer.GET.[NIVEL]([CLASSE].[MEMBRO_LOGGER], layout);
36     }
37   }
38 }
```

Listagem 4.5 - Estrutura do aspecto relacionado ao *logging* de métodos.
Fonte: Elaborada pelo autor.

Os elementos variáveis do aspecto, que estão entre colchetes, podem ser acompanhados no Quadro 4.4. Como esse quadro é referenciado nas próximas subseções, ele apresenta os elementos variáveis de todas as estruturas possíveis de um aspecto que pode ser criado pelo gerador.

Código	Descrição
ABSTRACT	Indica se um ponto de junção relacionado à execução de métodos será filtrado pelo modificar `Abstract`.
ASPECTO	Indica o nome único do aspecto, combinado a partir de: o nome da classe entrecortada, o nome da configuração criada, o tipo de mecanismo de configuração de *logging* e o nome e assinatura do método monitorado.
ASSINATURA_ CONSTRUTOR	Assinatura de um construtor.
ASSINATURA_ METODO	Assinatura de um método.
CLASSE	Nome da classe entrecortada.
FILTRO_NOME_ METODO	Indica se um ponto de junção relacionado à execução de métodos será filtrado por nome.
MEMBRO	Nome do membro de uma classe.
MEMBRO_ LOGGER	Indica o nome único do membro *logger* usado no aspecto, combinado a partir de: o nome da classe entrecortada, o nome da configuração criada, o tipo de mecanismo de configuração de *logging* e o nome e assinatura do método monitorado.
MENSAGEM_ CUSTOMIZADA	Indica a mensagem customizada traduzida pelo gerador (Subseção 4.4.1).
METODO	Nome de um método.
NIVEL	Nível das requisições de *logging* solicitadas a um *logger*.
NOME_LOGGER	Nome do *logger*.
SYNCHRONIZED	Indica se um ponto de junção relacionado à execução de métodos será filtrado pelo modificar `Synchronized`.
VISIBILIDADE	Indica se um ponto de junção relacionado à execução de métodos ou construtores será filtrado pela sua visibilidade.
TIPO_RETORNO	Indica se um ponto de junção relacionado à execução de métodos será filtrado pelo seu tipo de retorno.

Quadro 4.4 - Elementos variáveis das estruturas dos aspectos gerados para *logging*.
Fonte: Elaborado pelo Autor

Todos esses elementos variáveis estarão relacionados às configurações que forem criadas a partir da interface gráfica do mecanismo de configuração de *logging* para métodos, mostrada na Figura 4.12. No aspecto apresentado na Listagem 4.5, é definido um *logger* (linhas 2 e 3) para controlar os pedidos de *logging* de acordo com o que for definido no agrupamento *Logger Configs* da interface. Dois conjuntos de junção podem ser criados: um para representar os pontos da aplicação onde há execução do método, denominado `methodExecution` (linhas 5 e 6), e outro para agrupar os pontos onde há execução dos métodos e construtores chamados em cascata a partir do conjunto de junção `methodExecution`, denominado `methodCascade` (linhas 7 e 8) e que usa o operador `cflow` de AspectJ.

As opções do agrupamento *Log Options* da interface estão relacionadas aos adendos, de onde são feitos os pedidos de *logging*. A opção *Before* representa o adendo entre as linhas 10 e 16, a opção *After* as de 17 a 23, a opção *Cascade* as de 24 a 30 e a opção *Exception* as de 31 a 36. A opção *Cascade* usa o conjunto de junção `methodCascade`, todos os outros usam o conjunto de junção `methodExecution`. Dentro de cada adendo é feito um teste (linhas 11, 18, 25 e 32) para verificar se o *logger* correspondente está apto a despachar os pedidos de *logging* no nível configurado na interface para esse adendo. Esse teste é uma comparação booleana simples e permite ganho de desempenho, uma vez que impede a execução desnecessária de trechos de código mais custosos. Caso o *logger* permita o *logging*, a mensagem obedecerá ao formato relacionado àquele adendo (linhas 12, 19, 26 e 33), conforme foi definido na interface.

4.4.3.2 Relação com o Mecanismo de Configuração de *Logging* Seletivo para Métodos

O aspecto que será criado pelo gerador, relacionado a uma configuração de *logging* seletivo para métodos, permite que um conjunto de métodos e construtores possa ser escolhido com alvo de monitoramento através de *logging*. A estrutura geral desse aspecto, apresentada na Listagem 4.6, possui uma estrutura similar ao que foi descrito na Subseção 4.4.3.1. Portanto, somente os pontos diferentes serão comentados. Os elementos variáveis dessa estrutura podem ser verificados no Quadro 4.4.

Todos esses elementos variáveis estarão relacionados à configuração que for criada a partir da interface gráfica do mecanismo de configuração de *logging* seletivo para métodos, mostrada na Figura 4.15. Os métodos e construtores monitorados serão aqueles presentes da tabela *Methods in Use* do agrupamento *Methods Selection* dessa interface. A principal diferença na estrutura desse aspecto está na declaração dos conjuntos de junção. Nessa estrutura, o conjunto de junção `methodExecution` (linhas 12 a 17) permite agrupar vários pontos que representam a execução de métodos. É também mostrado dois novos conjuntos de junção, o `constructorExecution` (linhas 5 a 8), relacionado à execução dos construtores de uma classe, e o `constructorCascade` (linhas 9 e 10), que representa os pontos onde há execução dos métodos e construtores chamados em cascata a partir do conjunto de junção `constructorExecution`.

```
1  public aspect [ASPECTO] {
2    private static Logger [CLASSE].[MEMBRO_LOGGER] =
3            LoggerInitializer.GET.getLogger([NOME_LOGGER]);
4
5    private pointcut constructorExecution() :
6      execution([CLASSE].new([ASSINATURA_CONSTRUTOR_1])) ||
7                ...
8      execution([CLASSE].new([ASSINATURA_CONSTRUTOR_N]));
9    private pointcut constructorCascade(): cflow(constructorExecution()) &&
10     (execution(* *(..)) || execution(new(..)));
11
12   private pointcut methodExecution() :
13     execution(* [CLASSE].[METODO_1]([ASSINATURA_METODO_1])) ||
14               ...
15     execution(* [CLASSE].[METODO_M]([ASSINATURA_METODO_M]));
16   private pointcut methodCascade(): cflow(methodExecution()) &&
17     (execution(* *(..)) || execution(new(..)));
18
19   before() : constructorExecution() || methodExecution() {
20     if([CLASSE].[MEMBRO_LOGGER].isEnabledFor(Level.[NIVEL])) {
21       String customizedMessage = [MENSAGEM_CUSTOMIZADA];
22       PatternLayout layout = new PatternLayout(customizedMessage);
23       LoggerInitializer.GET.[NIVEL]([CLASSE].[MEMBRO_LOGGER], layout);
24     }
25   }
26   after(): constructorExecution() {
27     if([CLASSE].[MEMBRO_LOGGER].isEnabledFor(Level.[NIVEL])) {
28       String customizedMessage = [MENSAGEM_CUSTOMIZADA];
29       PatternLayout layout = new PatternLayout(customizedMessage);
30       LoggerInitializer.GET.[NIVEL]([CLASSE].[MEMBRO_LOGGER], layout);
31     }
32   }
33   after() returning(Object o): methodExecution() {
34     if([CLASSE].[MEMBRO_LOGGER].isEnabledFor(Level.[NIVEL])) {
35       String customizedMessage = [MENSAGEM_CUSTOMIZADA];;
36       PatternLayout layout = new PatternLayout(customizedMessage);
37       LoggerInitializer.GET.[NIVEL]([CLASSE].[MEMBRO_LOGGER], layout);
38     }
39   }
40   before() : constructorCascade() || methodCascade() {
41     if([CLASSE].[MEMBRO_LOGGER].isEnabledFor(Level.[NIVEL])) {
42       String customizedMessage = [MENSAGEM_CUSTOMIZADA];
43       PatternLayout layout = new PatternLayout(customizedMessage);
44       LoggerInitializer.GET.[NIVEL]([CLASSE].[MEMBRO_LOGGER], layout);
45     }
46   }
47   after() throwing: constructorExecution() || methodExecution() {
48     if([CLASSE].[MEMBRO_LOGGER].isEnabledFor(Level.[NIVEL])) {
49       String customizedMessage = [MENSAGEM_CUSTOMIZADA];
50       PatternLayout layout = new PatternLayout(customizedMessage);
51       LoggerInitializer.GET.[NIVEL]([CLASSE].[MEMBRO_LOGGER], layout);
52     }
53   }
54 }
```

Listagem 4.6 - Estrutura do aspecto relacionado ao *logging* seletivo de métodos.
Fonte: Elaborada pelo autor.

Como um construtor não possui valor de retorno é necessária a criação de dois adendos para representar a opção *After* do agrupamento *Log Options*. O adendo do tipo `after` (linhas 26 a 32) controla o *logging* referente à execução de construtores (`constructorExecution`), enquanto que o adendo `after returning` (linhas 33 a 39) acompanha a execução de métodos (`methodExecution`), permitindo que o retorno de um método possa ser capturado e adicionado à mensagem de *logging*. O adendo referente à opção *Before* (linhas 19 a 25) e o adendo referente à opção *Exception* (linhas 47 a 53) se referem aos conjuntos de junção `constructorExecution` e `methodExecution`. Já o adendo relativo à opção *Cascade* se refere aos conjuntos de junção conStructorCascade e `methodCascade`.

4.4.3.3 Relação com o Mecanismo de Configuração de *Logging* por Filtro para Métodos

Assim que uma configuração é criada pelo mecanismo de configuração de *logging* por filtro para métodos, o gerador de código é acionado para gerar o aspecto correspondente. Esse aspecto selecionará a partir de um conjunto de filtros os métodos e construtores de uma classe que serão monitorados por *logging*. A estrutura desse aspecto, mostrada na Listagem 4.7, se difere da apresentada na Subseção 4.4.3.2 nos seus conjuntos de junção, que serão detalhados abaixo. Os elementos variáveis dessa estrutura podem ser verificados no Quadro 4.4.

Todos esses elementos variáveis estarão relacionados à configuração que for criada a partir da interface gráfica do mecanismo de configuração de *logging* por filtro para métodos, mostrada na Figura 4.17. Os métodos e construtores selecionados serão aqueles que passarem pelo filtro do agrupamento *Methods Selection Filter* dessa interface. Os métodos serão filtrados por nome (*method name*), visibilidade (*visibility*), modificadores abstract e synchrozined (*modifiers*) e tipo de retorno (*return type*), como pode ser observado no conjunto de junção `methodExecution` (linhas 10 a 12). Um construtor só será monitorado se o valor do filtro por nome condisser com o nome da classe e o valor do filtro de visibilidade for o mesmo do construtor, como ilustrado no conjunto de junção `constructorExecution` (Linhas 5 e 6). Se mais algum filtro for utilizado no momento de criação da configuração, o aspecto não terá nenhum conjunto de junção

que faça referência a construtores. Logo, os adendos somente farão referência aos conjuntos de junção relacionados a métodos.

```
1  public aspect [ASPECTO] {
2    private static Logger [CLASSE].[MEMBRO_LOGGER] =
3            LoggerInitializer.GET.getLogger([NOME_LOGGER]);
4
5    private pointcut constructorExecution() :
6      execution([VISIBILIDADE] [CLASSE].new(..));
7    private pointcut constructorCascade(): cflow(constructorExecution()) &&
8      (execution(* *(..)) || execution(new(..)));
9
10   private pointcut methodExecution() :
11     execution([VISIBILIDADE] [ABSTRACT] [SYNCHROZINED] [TIPO_RETORNO]
12               [CLASSE].[FILTRO_NOME_METODO](..));
13   private pointcut methodCascade(): cflow(methodExecution()) &&
14     (execution(* *(..)) || execution(new(..)));
15
16   before() : constructorExecution() || methodExecution() {
17     if([CLASSE].[MEMBRO_LOGGER].isEnabledFor(Level.[NIVEL])) {
18       String customizedMessage = [MENSAGEM_CUSTOMIZADA];
19       PatternLayout layout = new PatternLayout(customizedMessage);
20       LoggerInitializer.GET.[NIVEL]([CLASSE].[MEMBRO_LOGGER], layout);
21     }
22   }
23   after(): constructorExecution() {
24     if([CLASSE].[MEMBRO_LOGGER].isEnabledFor(Level.[NIVEL])) {
25       String customizedMessage = [MENSAGEM_CUSTOMIZADA];
26       PatternLayout layout = new PatternLayout(customizedMessage);
27       LoggerInitializer.GET.[NIVEL]([CLASSE].[MEMBRO_LOGGER], layout);
28     }
29   }
30   after() returning(Object o): methodExecution() {
31     if([CLASSE].[MEMBRO_LOGGER].isEnabledFor(Level.[NIVEL])) {
32       String customizedMessage = [MENSAGEM_CUSTOMIZADA];;
33       PatternLayout layout = new PatternLayout(customizedMessage);
34       LoggerInitializer.GET.[NIVEL]([CLASSE].[MEMBRO_LOGGER], layout);
35     }
36   }
37   before() : constructorCascade() || methodCascade() {
38     if([CLASSE].[MEMBRO_LOGGER].isEnabledFor(Level.[NIVEL])) {
39       String customizedMessage = [MENSAGEM_CUSTOMIZADA];
40       PatternLayout layout = new PatternLayout(customizedMessage);
41       LoggerInitializer.GET.[NIVEL]([CLASSE].[MEMBRO_LOGGER], layout);
42     }
43   }
44   after() throwing: constructorExecution() || methodExecution() {
45     if([CLASSE].[MEMBRO_LOGGER].isEnabledFor(Level.[NIVEL])) {
46       String customizedMessage = [MENSAGEM_CUSTOMIZADA];
47       PatternLayout layout = new PatternLayout(customizedMessage);
48       LoggerInitializer.GET.[NIVEL]([CLASSE].[MEMBRO_LOGGER], layout);
49     }
50   }
51 }
```

Listagem 4.7 - Estrutura do aspecto relacionado ao *logging* por filtro de métodos.
Fonte: Elaborada pelo autor.

4.4.3.4 Relação com o Mecanismo de Configuração de *Logging* Seletivo para Membros

O aspecto construído pelo gerador de código a partir de uma configuração feita no mecanismo de configuração de *logging* seletivo para membros é capaz de monitorar por *logging* os membros de uma classe que sofrem alteração do seu valor ou que são alvos de leitura em alguma parte da aplicação. A estrutura geral desse aspecto pode ser observada na Listagem 4.8. Os elementos variáveis do aspecto, que estão entre colchetes, podem ser verificados no Quadro 4.4.

```
1  public aspect [ASPECTO] {
2    private static Logger [CLASSE].[MEMBRO_LOGGER] =
3            LoggerInitializer.GET.getLogger([NOME_LOGGER]);
4
5    private pointcut fieldGet():
6      get(* [CLASSE].[MEMBRO_1]) ||
7                  ...
8      get(* [CLASSE].[MEMBRO_N]);
9    private pointcut fieldSet(Object o):
10     (set(* [CLASSE].[MEMBRO_1]) ||
11                 ...
12      set(* [CLASSE].[MEMBRO_N])) && args(o);
13
14   Object around() : fieldGet() {
15     Object o = proceed();
16     if([CLASSE].[MEMBRO_LOGGER].isEnabledFor(Level.[NIVEL])) {
17       String customizedMessage = [MENSAGEM_CUSTOMIZADA];
18       PatternLayout layout = new PatternLayout(customizedMessage);
19       LoggerInitializer.GET.[NIVEL]([CLASSE].[MEMBRO_LOGGER], layout);
20     }
21     return o;
22   }
23   before(Object o) : fieldSet(o)
24   {
25     if([CLASSE].[MEMBRO_LOGGER].isEnabledFor(Level.[NIVEL]))  {
26       String customizedMessage = [MENSAGEM_CUSTOMIZADA];
27       PatternLayout layout = new PatternLayout(customizedMessage);
28       LoggerInitializer.GET.[NIVEL]([CLASSE].[MEMBRO_LOGGER], layout);
29     }
30   }
31 }
```

Listagem 4.8 - Estrutura do aspecto relacionado ao *logging* por filtro de membros.
Fonte: Elaborada pelo autor.

Todos esses elementos variáveis estarão relacionados à configuração que for criada a partir da interface gráfica do mecanismo de configuração de *logging* seletivo para membros, destacada na Figura 4.19. Os membros monitorados serão aqueles presentes da tabela *Members in Use* do agrupamento *Members Selection* dessa interface. Para representar os pontos onde há operações de leitura sobre os membros selecionados, é criado o conjunto de junção `fieldGet` (linhas 5 a 8). Enquanto

isso, o conjunto de junção `fieldSet` (linhas 9 a 12) representa os pontos onde há alterações no valor desses membros.

O adendo do tipo `around` (linhas 14 a 22) se relaciona com o conjunto de junção `fieldGet`. Esse tipo de adendo permite o uso do elemento `proceed`, que representa a execução completa do ponto de junção monitorado. Neste caso, ele representa o retorno do valor de um membro, ou seja, a sua leitura. Esse valor é armazenado em uma variável (linha 15), pois pode ser usado na mensagem de *logging*. Já o adendo do tipo `before` (linhas 23 a 30) tem a função de gerar o *logging* para os pontos definidos no conjunto de junção `fieldSet`.

4.5 Benefícios do Asplog

São perceptíveis os ganhos de tempo e de custo (aumento de produtividade) quando se compara o processo visual e automatizado utilizado no Asplog para o gerenciamento de *logging* através do uso de aspectos, apresentado na Figura 4.21, com um processo manual e clássico (uso de POO), mostrado na Figura 4.20. No processo manual, é necessário um esforço considerável do desenvolvedor nas etapas de vistoria (1 e 5), inserção (2) e remoção (6) dos códigos de *logging*. Enquanto isso, no processo visual, o trabalho do desenvolvedor se concentra apenas na seleção (1) e desativação (5) visual dos pontos monitorados por *logging*. Além disso, o Asplog possibilita a modularização do interesse transversal de *logging*, eliminando o entrelaçamento e espalhamento que ocorre quando uma abordagem orientada a objetos é utilizada.

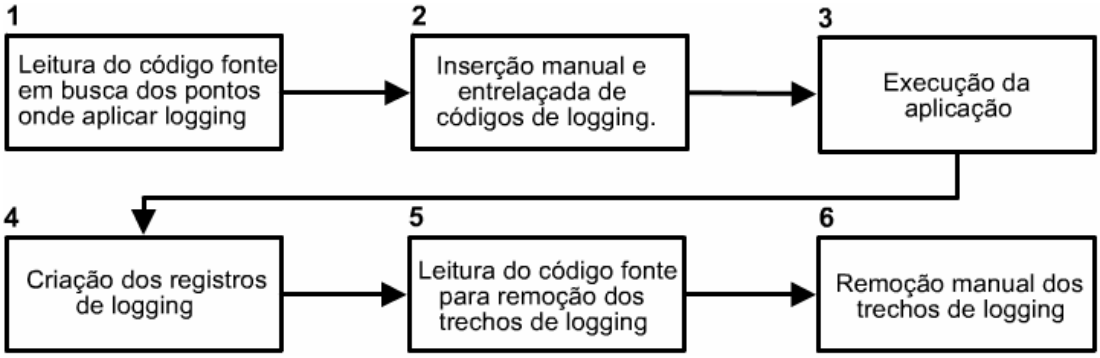

Figura 4.20 – Processo clássico de gerenciamento manual do *logging*.
Fonte: Elaborada pelo autor.

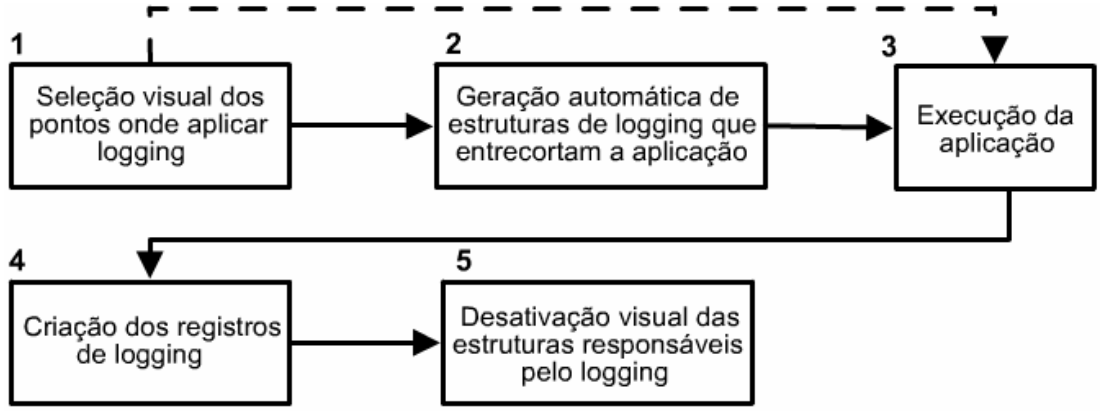

Figura 4.21 – Processo de gerenciamento visual do *logging* utilizado no Asplog.
Fonte: Elaborada pelo autor.

4.6 Limitação do Asplog

Durante o desenvolvimento de um sistema, é comum acontecer mudanças nas estruturas responsáveis pela implementação dos requisitos e operações de refatoração que visam deixar o código mais claro, tais como:

- Alteração nos nomes de classes, métodos e membros;
- Alteração da assinatura dos métodos e do tipo dos membros;
- Alteração das operações de um método;
- Alteração na hierarquia de classes;
- Mudança do pacote a qual uma classe pertence;
- Inserção de novos métodos e membros em uma classe;
- Remoção de classes, métodos e membros.

Infelizmente, o Asplog não tem a capacidade de suportar tais tipos de operações, ou seja, se houver alguma mudança como as enumeradas anteriormente, as configurações e aspectos já criados não irão refleti-la automaticamente. Para resolver essa questão, será necessário que o usuário do Asplog remova do projeto as classes e aspectos gerados que estão relacionados às configurações afetadas pela mudança. Posteriormente, é preciso refazer as configurações de modo a refletir as alterações ocorridas na estrutura do sistema. Assim, novas classes e aspectos serão gerados automaticamente pelo Asplog para tratar do interesse transversal de *logging*.

Essa limitação do Asplog pode ser contornada por meio de um certo esforço de programação. Contudo, como esse não é um problema crítico que invalida o que foi desenvolvido, optou-se por descartar, para o momento, uma solução para essa limitação. Dessa forma, pode-se concentrar tempo e esforço no desenvolvimento de características mais importantes para o funcionamento do mecanismo visual criado.

5 CONCLUSÕES E TRABALHOS FUTUROS

Nesta seção, é feita uma avaliação geral do trabalho desenvolvido. Para isso, os objetivos propostos inicialmente são considerados, assim como os meios utilizados para alcançá-los. Dessa forma, é possível apresentar as principais conclusões e contribuições relacionadas a esse trabalho. Em seguida, são enumeradas algumas sugestões de trabalhos futuros com a intenção de contribuir com o desenvolvimento dessa linha de pesquisa.

5.1 Conclusões

A POO é um paradigma de programação capaz de modularizar interesses de negócio, porém não permite gerenciar eficientemente interesses transversais, o que acarreta em problemas de entrelaçamento e espalhamento de código. Logo, com o passar do tempo, foi verificada a necessidade de se resolver essa questão. Conseqüentemente, novas abordagens surgiram propondo soluções cuja principal intenção era a utilização de novas estruturas e perspectivas para modularizar esses interesses transversais.

Dentre essas abordagens, a de maior destaque é a Programação Orientada a Aspectos, que sugere a utilização de novas estruturas – os aspectos – para a modularização de interesses transversais. Porém, de forma contraditória ao que a maioria pode imaginar, as linguagens orientadas a aspectos possuem conceitos e sintaxes complexas. Isso acaba gerando certa rejeição e inércia quanto ao aprendizado e adoção por grande parte das pessoas, que vêem com receio a "escalada" da curva de aprendizado. Contudo, pode-se afirmar que o esforço no aprendizado é compensado através da elaboração de melhores projetos e de códigos mais limpos e legíveis, o que acaba acarretando em ganho de qualidade e na diminuição de custos nas fases do ciclo de vida de um sistema.

A fim de diminuir as barreiras que impedem uma maior popularização da POA e dos benefícios proporcionados por ela, vários trabalhos vêm propondo meios de abstrair a sua complexidade, tornando transparente para o desenvolvedor a sua utilização. Nesta dissertação, foram avaliados os trabalhos descritos em Davies *et al*.

(2003), Hughes e Greenwood (2003) , Hawkins e January (2006) e Couto (2006). Perceberam-se algumas características em comum nesses trabalhos, tais como: intensa utilização de soluções visuais, apelo ao uso de recursos mnemônicos, alto grau de interatividade, e principalmente, total abstração quanto ao uso de aspectos.

Seguindo a linha de pensamento desses trabalhos, verificou-se uma carência quanto à existência de soluções que oferecessem o mesmo tipo de tratamento para o interesse transversal de *logging*, que Gupta (2003) afirma ter um papel importante na qualidade final do sistema, pois permite encontrar problemas mais cedo e com mais precisão. Log4E (2008) é um projeto que propõe uma solução visual para o tratamento de *logging* por meio de uma abordagem orientada a objetos, o que, porém, não soluciona os problemas de espalhamento e entrelaçamento de código.

Assim sendo, foi proposto como objetivo principal deste trabalho analisar o uso de técnicas de programação orientada a aspectos na modularização do interesse transversal de *logging* a fim de desenvolver um mecanismo visual para automatização desse interesse. Um estudo embasado na literatura permitiu planejar cada uma das etapas seguidas durante a realização deste trabalho para a concretização dos objetivos específicos. Conseqüentemente, o objetivo principal pôde ser alcançado com sucesso por meio do desenvolvimento do Asplog.

Com o Asplog, pode-se ratificar que o emprego de técnicas orientadas a aspectos são eficientes para solucionar o problema de entrelaçamento e espalhamento de código ocasionado pelo *logging*. Indo além, permitiu-se comprovar que é possível abstrair o uso de aspectos em prol da simplificação de sua complexidade para que seja possível usufruir de seus benefícios, principalmente o de permitir a modularização de interesses transversais.

A integração do Asplog ao Eclipse no formato de um *plugin* permite simplificar o processo de instalação, utilização e adoção por parte dos interessados, uma vez que o mecanismo se beneficia das vantagens de estar relacionado a uma IDE popular.

Por meio da extensão PDE do Eclipse, que fornece meios para a criação de *plugins*, dos recursos da linguagem orientada a aspectos AspectJ e da biblioteca de *logging* Log4J, foi possível adicionar características interessantes ao Asplog para os seus submecanismos, dentre elas:

- Utilização de recursos visuais, por meio de interfaces gráficas que facilitam a interatividade e memorização do usuário;
- Flexibilidade na criação de modelos de mensagem reutilizáveis para *logging*, além de um conjunto de opções de informações;
- Centralização dos *loggers* responsáveis por despachar os pedidos de *logging*, o que facilita configurá-los, inclusive para operações de (des)ativação dos trechos monitorados;
- Variedade de opções para definir os pontos de uma aplicação que devem ser monitorados. Opções essas que foram apoiadas nos conceitos de POA;
- Gerador automático de aspectos e classes responsáveis por gerenciar, de forma modular, o interesse transversal de *logging*, que oculta a complexidade por trás das estruturas geradas.

A questão *usabilidade* esteve em foco durante todas as etapas de projeto e desenvolvimento. Desse modo, apesar de não ter sido realizada uma pesquisa com um grupo de usuários do Asplog para avaliar essa questão, acredita-se que ele é de fácil manuseio e que simplifica e agiliza o processo de gerenciamento do interesse transversal de *logging* por meio de uma abordagem visual, que abstrai a orientação a aspectos.

Uma limitação no Asplog foi verificada no que diz respeito ao suporte de alterações nas estruturas responsáveis por implementar os requisitos de uma aplicação, seja por mudanças nos requisitos ou por operações de refatoração. Nesses casos, as alterações não irão refletir automaticamente nas estruturas criadas pelo gerador, sendo necessária a intervenção do usuário. Porém, essa limitação não impede e muito menos invalida o mecanismo. Portanto, optou-se por postergar a implementação da solução, que se resume apenas a esforço de programação.

Como consideração final em termos de contribuição dessa pesquisa, acredita-se que os resultados são relevantes, à medida que proporcionam uma solução diferenciada que melhora o gerenciamento do requisito de *logging*, necessário à grande maioria dos sistemas. Em termos teóricos, buscou-se simplificar a utilização da POA para esse requisito, pois apesar das linguagens orientadas a aspectos serem poderosas, ainda deixam a desejar na questão de simplicidade de sintaxes e conceitos, o que dificulta utilizá-las.

5.2 Trabalhos Futuros

Com a intenção de contribuir com a ramificação e expansão do horizonte desse trabalho, são apresentadas a seguir algumas recomendações e sugestões para trabalhos futuros nessa linha:

- Expandir e evoluir os recursos do Asplog, a fim de que seja oferecido um mecanismo cada vez mais completo para o tratamento de *logging*. Além disso, resolver a limitação relacionada ao suporte de mudanças nas estruturas responsáveis pela implementação dos requisitos e de operações de refatoração (ver Subseção 4.6);
- Quantificar, a partir da avaliação de um grupo de usuários, as contribuições proporcionadas pelo mecanismo através da análise de variáveis como: ganho de produtividade, ganho de qualidade, grau de modularização atingido, nível de simplificação do problema, grau de utilidade do mecanismo, nível de abstração da POA, dentre outras;
- Propor formas de abstrair a complexidade por detrás de outras linguagens orientadas a aspectos, como AspectC e AspectC++, e para outros interesses transversais carentes nesse sentido.

REFERÊNCIAS BIBLIOGRÁFICAS

AJDT - AspectJ Development Tools. Disponível em: <http://www.eclipse.org/ajdt/>. Acesso em: 01 mar. 2008.

AKSIT, M.; WAKITA, K.; BOSCH, J.; BERGMANS, L.; YONEZAWA, A. **Abstracting Object Interactions Using Composition Filters**. In: PROCEEDINGS OF THE ECOOP'93 WORKSHOP ON OBJECT-BASED DISTRIBUTED PROGRAMMING, 1993, Springer-Verlag, v. 791, p. 152–184.

ASPECTC++. Disponível em: <http://www.aspectc.org/>. Acesso em: 01 mar. 2008.

ASPECTJ. Disponível em: <http://www.eclipse.org/aspectj>. Acesso em: 01 mar. 2008.

BOOCH G.; RUMBAUGH, J.; JACOBSON I. **The Unified Modeling Language User Guide**. 1 ed. Reading, MA: Addison-Wesley, 1999. 512 p.

CAMARGO, V. V.; MASIERO, P. C. **Frameworks Orientados a Aspectos**. In: XIX SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE (SBES'2005), Uberlândia, MG, 2005.

CHAN, C. **Effective Logging Practices Ease Enterprise Development**. 2005. Disponível em: <http://www.ibm.com/developerworks/java/library/j-logging/>. Acesso em: 01 mar. 2008.

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 8TH EUROPEAN SOFTWARE ENGINEERING CONFERENCE held jointly with 9TH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, 2001, ACM Press, p. 88-98.

COUTO, C. **Um Arcabouço Orientado por Aspectos Para Implementação Automatizada de Persistência**. 109 f. Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Minas Gerais, Belo Horizonte, MG, 2006. Disponível em: <http://hdl.handle.net/1843/RVMR-6TJQAZ>. Acesso em: 01 mar. 2008.

DAVIES, J.; HUISMANS, N.; SLANEY, R.; WHITING, S. **An Aspect Oriented Performance Analysis Environment**. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT (AOSD'03), 2003, Boston, MA.

DIJKSTRA, E. W. **A Discipline of Programming**. Englewood Cliffs, NJ: Prentice-Hall, 1976. 217 p.

ECLIPSE. Disponível em: <http://www.eclipse.org>. Acesso em: 01 mar. 2008.

ELRAD, T.; KICZALES, G.; AKSIT, M.; LIEBERHER, K.; OSSHER, H. Discussing Aspects of AOP. **Communications of the ACM**, New York, NY, v. 44, n. 10, p. 33-38, 2001.

FILMAN, R.; ELRAD, T.; CLARKE, S.; AKSIT, M. **Aspect-Oriented Software Development**. 1. ed. Boston, MA: Addison-Wesley, 2005. 800 p.

GRADECK, J. D.; LESIECKI, N. **Mastering AspectJ: Aspect-Oriented Programming in Java**. 1. ed. Indianopolis, Indiana: Wiley, 2003. 456 p.

GULCU, C. **Log4j: The complete manual**. 2004. 206 p.

GUPTA, S. **Logging in Java with the JDK 1.4 Logging API and Apache Log4j**. New York, NY: Apress, 2003. 336 p.

HAPPEL, H. J.; SCHMIDT, A. **Knowledge Maturing as a Process Model for Describing Software Reuse**. In: WORKSHOP ON LEARNING SOFTWARE ORGANIZATIONS (LSO'07) held jointly with 4TH CONFERENCE PROFESSIONAL KNOWLEDGE MANAGEMENT - EXPERIENCES AND VISIONS (WM'07), 2007, Potsdam, GITO, p. 155-164.

HAWKINS, H; JANUARY, S. **Aspects For MDD: Aspect-Based Tracing and First Failure Data Capture in Rational Software Architect**. 2006. Disponível em: http://www-128.ibm.com/developerworks/rational/library/06/0711_hawkins_january/>. Acesso em: 01 mar. 2008.

HARRISON, W.; OSSHER, H. **Subject-Oriented Programming: A Critique of Pure Objects**. In: PROCEEDINGS OF THE ACM CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS (OOPSLA'93), 1993, v. 28, n. 10, p. 411-428.

HUGHES, D.; GREENWOOD, P; BLAIR L. **Aspect Testing Framework**. In: FORMAL METHODS FOR OPEN OBJECT-BASED DISTRIBUTED SYSTEMS AND DISTRIBUTED APPLICATIONS AND INTEROPERABLE SYSTEMS (FMOODS/DAIS) PHD WORKSHOP, Paris, France, 2003.

HUGHES, D.; GREENWOOD, P.; COULSON, G. **A Framework for Testing Distributed Systems**. In: PROCEEDINGS OF THE 4TH IEEE INTERNATIONAL CONFERENCE ON PEER-TO-PEER COMPUTING, Zurich, Switzerland, 2004.

IRSA - IBM Rational Software Architect. Disponível em: <http://www-306.ibm.com/software/awdtools/architect/swarchitect/>. Acesso em: 01 mar. 2008.

JAVA. Disponível em: <http://java.sun.com>. Acesso em: 01 mar. 2008.

JAVA LOGGING. Disponível em: <http://java.sun.com/j2se/1.4.2/docs/guide/util/logging>. Acesso em: 01 mar. 2008.

JAXB – Java Architecture for XML Binding. Disponível em: <https://jaxb.dev.java.net/>. Acesso em: 01 mar. 2008.

KICZALES, G; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C. V.; LOINGTIER, J.; IRWIN, J. **Aspect-Oriented Programming**. In: PROCEEDINGS OF THE EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP'97), 1997, Finland, Springer-Verlag, v. 1241, p. 220-242.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. G. **An overview of AspectJ**. In: PROCEEDINGS OF THE 15TH EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP'01), 2001, Berlin, Springer-Verlag, p. 327-353.

KICZALES, G. In: ELRAD, T.; KICZALES, G.; AKSIT, M.; LIEBERHER, K.; OSSHER, H. Discussing Aspects of AOP. **Communications of the ACM**, New York, NY, v. 44, n. 10, p. 33-38, 2001.

KULESZA, U.; SANT'ANNA, C; LUCENA, C. **Técnicas de Projeto Orientado a Aspectos**. In: XIX SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE (SBES), Uberlândia, MG, 2005.

LADDAD, R. **AspectJ in Action: Practical Aspect-Oriented Programming**. Greenwich, CT: Manning, 2003. 512 p.

LIEBERHERR, K. **Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns**. 1. ed. Boston, MA: PWS Publishing Company, 1995. 656 p.

LOG4E. Disponível em: <http://log4e.jayefem.de/>. Acesso em 01 mar. 2008.

LOG4J. Disponível em: <http://logging.apache.org/log4j/>. Acesso em 01 mar. 2008.

NETBEANS. Disponível em: <http://www.netbeans.org/>. Acesso em 01 mar. 2008.

ORT, E.; MEHTA, B. **Java Architecture for XML Binding (JAXB)**. 2003. Disponível em: <http://java.sun.com/developer/technicalArticles/WebServices/jaxb/>. Acesso em: 01 mar. 2008.

OSSHER, H.; TARR, P. **Hyper/J: Multi-Dimensional Separation of Concerns for Java**. In: PROCEEDINGS OF THE 22ND INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEENRING, 2000, ACM Press, p. 734-737.

OSSHER, H.; TARR, P. Using Multi-dimensional Separation of Concerns to (Re)Shape Evolving Software. **Communications of the ACM**, New York, NY, v. 44, n. 10, p. 43–50, 2001.

PARNAS, D. L. On the Criteria To Be Used in Decomposing Systems Into Modules. **Communications of the ACM**, New York, NY, v. 15, n. 12, p. 1053-1058, 1972.

PDE – Plug-in Development Environment. Disponível em: <http://www.eclipse.org/pde/>. Acesso em: 01 mar. 2008.

SCHILDT, H. **C Completo e Total**. 3. ed. São Paulo, SP: Makron Books, 1997. 827 p.

TIRELO, F; BIGONHA, R. S.; BIGONHA M. A. S.; VALENTE, M. T. O. **Desenvolvimento de Software Orientado por Aspectos**. In: XIII JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA (JAI'04) - SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 2004, Salvador, BA, v. 2, p. 57-96.