Mecanismo Visual Baseado em Aspectos para Automatização de Logging

Pedro Augusto Felipe Machado Gazolla[1] , Vladimir Oliveira Di Iorio[1], e João Gabriel Felipe Machado Gazolla[1]

[1] Departamento de Informática, Universidade Federal de Viçosa (UFV), Av. P.H. Rolfs, s/n, Viçosa-MG, Brasil
{pgazolla, vladimir, gazolla}@dpi.ufv.br

Resumo. A Programação Orientada a Objetos 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 se destacou. Entretanto, as linguagens orientadas a aspectos possuem deficiências que dificultam sua maior popularização: sintaxe e conceitos complexos. Este trabalho apresenta um mecanismo visual para geração automática de aspectos relacionados ao interesse transversal de *logging*, que permite ser manipulado de forma interativa e transparente quanto ao uso de aspectos, possibilitando a modularização desse interesse.

Palavras-chave: *Logging*. Programação Orientada a Aspectos. Ferramenta Visual.

1 Introdução

Em geral, independente da qualidade do processo de desenvolvimento, os sistemas apresentam defeitos, ou seja, não-conformidades em relação aos requisitos desejados. 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 [5]. 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. Normalmente, as estratégias de *logging* se baseiam no paradigma de Programação Orientada a Objetos (POO).

É notável o avanço proporcionado pela POO. A partir dela, os problemas passaram a ser vistos de uma forma mais natural, onde cada classe deveria ser responsável por modularizar um único requisito do sistema. Porém, na prática, isso acontece apenas parcialmente, devido à existência de interesses transversais (de importância sistêmica), como por exemplo, o *logging*. Isso acarreta em entrelaçamento (uma classe tratando de mais de um interesse) e espalhamento de código (o mesmo interesse sendo tratado por várias classes), o que, conseqüentemente, contribui para a

diminuição da coesão e aumento do acoplamento no sistema, uma vez que os níveis de modularização são prejudicados.

A Programação Orientada a Aspectos (POA) [12] 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 [13]. Com a POA, interesses transversais são implementados em aspectos, ao invés de ficarem espalhados dentro dos outros módulos. No entanto, as linguagens orientadas a aspectos apresentam níveis altos de dificuldade para sua utilização, com conceitos e sintaxes novos e complexos, ou seja, essas linguagens possuem uma longa curva de aprendizado.

Este artigo tem por objetivo apresentar um mecanismo denominado Asplog (*Aspect Logger*) que possibilita usufruir de forma transparente dos benefícios proporcionados pela POA para a modularização do interesse transversal de *logging* através da geração automática de aspectos. Para isso, o Asplog disponibiliza um conjunto de recursos visuais, interativos e mnemônicos, que facilitam o gerenciamento desse interesse. Como forma de simplificar o processo de instalação, utilização e adoção desse mecanismo, ele foi desenvolvido no formato de um *plugin* para o ambiente de desenvolvimento Eclipse [3].

A Seção 2 discute a importância do *logging*. A Seção 3 trata da POA e apresenta a linguagem orientada a aspectos AspectJ. A Seção 4 discute sobre trabalhos relacionados a este. A Seção 5 apresenta as principais funcionalidades do mecanismo desenvolvido, juntamente com suas interfaces gráficas, e a ferramenta de geração de aspectos. As conclusões finais são apresentadas na Seção 6.

2 *Logging*

O *logging* se tornou uma boa prática na engenharia de software, dado que está intimamente ligado à qualidade do sistema, e está sendo largamente utilizado [7]. Nessa perspectiva, Gupta [6] afirma que toda aplicação comercial necessita de *logging*, caso contrário, essa aplicação será difícil de depurar e perderá valor de mercado. Por conseguinte, os desenvolvedores começaram a visualizá-lo como um interesse que necessita ser modularizado, facilitando o seu gerenciamento.

Algumas decisões são necessárias para se atingir uma estratégia eficiente de *logging*, e isso requer, por parte do projetista, planejamento e disciplina. A tecnologia a ser utilizada pode facilitar bastante esse processo, e 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 [5, 6].

O uso do logging proporciona vários benefícios para um sistema. É possível diagnosticar problemas mais cedo e com mais precisão. Essa contextualização dos problemas simplifica as tarefas de manutenção. Quando o depurador (*debugger*) não é uma opção, seja pela velocidade que as atividades ocorrem ou a natureza distribuída das aplicações, ele possibilita o monitoramento da execução do sistema durante o desenvolvimento. 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. Com base nesses benefícios, ficam claros os ganhos de tempo e de custos no ciclo de vida de uma aplicação.

3 Programação Orientada a Aspectos

O objetivo da Programação Orientada a Aspectos (POA) [12] 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 [4].

Para isso, foram definidos, em [12], 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, isto é, os interesses transversais.

A modularização total em nível de interesses é possível, na filosofia orientada a aspectos, porque as estruturas responsáveis por encapsular os interesses de negócio (como as classes) não têm consciência da existência dos interesses transversais. Em contrapartida, os aspectos precisam saber os pontos onde esses interesses transversais devem se relacionar com os interesses de negócio. Percebe-se, então, que ocorre uma inversão de controle.

A POA envolve basicamente três etapas distintas de desenvolvimento. Na *Decomposição Aspectual*, os interesses do sistema são decompostos em interesses de negócio e interesses transversais. Na *Implementação dos Interesses*, cada interesse deve ser implementado separadamente. Por fim, ocorre a *Recomposição Aspectual*, onde as regras de recomposição devem ser especificadas nos aspectos para que o processo de recomposição (*weaving*) componha o sistema final.

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 [13].

3.1 AspectJ

A linguagem AspectJ [11] é uma extensão orientada a aspectos da linguagem Java e 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 realizada 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 [4].

AspectJ oferece os mesmos recursos de Java, tais como: classes, métodos, atributos e outros. Além disso, são fornecidos novos conceitos e construções. Um ponto de junção (*point cut*) é um ponto bem definido na execução de um programa. Um conjunto de junção (*join point*) é um conjunto de pontos de junção. Um adendo (*advice*) define o momento (antes, durante ou depois) e a ação a ser executada quando um ponto de junção é atingido. Elementos com suporte a reflexão também são disponibilizados com o intuito de obter informações do contexto de execução de um ponto de junção. Por último, o aspecto (*aspect*) é a unidade modular que encapsula os demais elementos para a implementação de requisitos transversais.

4 Trabalhos Relacionados

Durante a investigação de trabalhos relacionados, não foi encontrado nenhum que trate especificamente de *logging* por meio de uma abordagem orientada a aspectos. O trabalho mais correlato encontrado foi o Log4E [14]. Esse trabalho tem como objetivo simplificar a criação de *logging* por meio de uma abordagem visual e orientada a objetos, não resolvendo os problemas de entrelaçamento e espalhamento de código, ou seja, ele não é capaz de modularizar o *logging*.

Porém, foi verificado que existem trabalhos que visam abstrair a complexidade das linguagens orientadas a aspectos para o tratamento de outros interesses transversais.

Em [2], é descrito 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.

Um *framework* usando POA e mecanismos de reflexão, cujo objetivo é simplificar o teste de sistemas distribuídos, foi proposto em [8]. Esse *framework* possui uma estrutura de modelos (*templates*) para a criação de aspectos em AspectJ. Posteriormente, baseado nessa *framework*, foi desenvolvido, em [9], um ambiente para possibilitar a inserção e remoção automática de cenários de teste em sistemas distribuídos.

Em [1], foi proposto um *framework* orientado a aspectos para tratar do interesse transversal de persistência. 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. Ainda nesse trabalho, foi desenvolvida uma ferramenta para geração automática dos aspectos concretos que especializam os aspectos abstratos do *framework*.

5 Asplog

O Asplog é um mecanismo que foi desenvolvido com a finalidade de resolver o problema da modularização do interesse transversal de *logging* e de facilitar a criação de seus trechos de código, auxiliando o desenvolvedor na construção de sistemas. Isso

é alcançado por meio de um processo de criação completamente visual e flexível que possui como resultado final aspectos e classes geradas automaticamente, que são responsáveis pela modularização desse interesse. Como forma de atingir seu objetivo, o Asplog foi dividido em quatro partes:

- *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 ou estado das classes devem ser monitorados através de *logging*.
- *Mecanismo de Geração de Código* - baseia-se nas configurações definidas para criar os aspectos (em AspectJ) e classes responsáveis por modularizar o *logging*.

5.1 Esquema de Funcionamento

O esquema de funcionamento do Asplog é mostrado na Figura 1. Nele, 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 L*oggers*. 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 (*XML data binding*) ocorre usando JAXB [10]. As classes do sistema passam a ser entrecortadas pelos aspectos gerados. Verifica-se que a API Java Log4j [15], que possibilita a criação de códigos de *logging*, foi adicionada ao projeto e tem papel fundamental no seu funcionamento. Do Log4j, foram aproveitadas suas estruturas, como o *logger*, que controla as requisições de *logging*, o *appender*, que determina o destino de saída das mensagens; e o *layout*, que permite definir o formato das mensagens. Por fim, é mostrado que o Asplog foi incorporado ao Eclipse como um *plugin*.

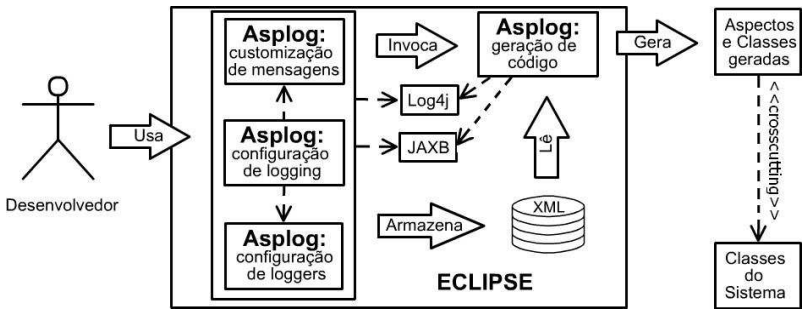

Fig. 1. Esquema de funcionamento do Asplog.

5.2 Mecanismo de Customização de Mensagens

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 a criticalidade dos pontos que estão sendo monitorados e a necessidade da pessoa responsável por analisá-las.

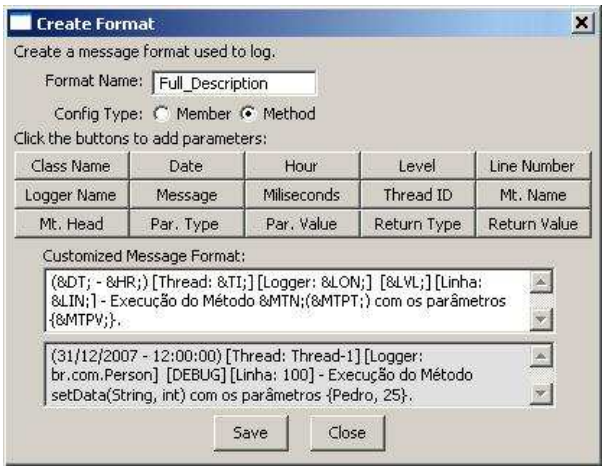

Fig. 2. Interface gráfica para criação de um modelo de mensagem reutilizável.

O Mecanismo de Customização de Mensagens simplifica o processo de personalização de mensagens de *logging*, permitindo a criação de modelos de mensagem reutilizáveis, ou seja, uma vez criado um modelo, basta fazer uma referência a ele para utilizá-lo em qualquer lugar onde o *logging* for necessário. Várias informações (estáticas e dinâmicas) sobre o contexto dos pontos monitorados podem ser adicionadas em um modelo, por exemplo, o nome, valores dos parâmetros, e o momento de execução (data e hora) de um método. Na Figura 2, é apresentada a interface gráfica que permite a criação de um modelo de mensagem reutilizável. Na caixa de texto inferior dessa interface, é possível acompanhar um exemplo de como ficaria um registro de *logging* a partir do modelo em desenvolvimento.

5.3 Mecanismo de Configuração de *Loggers*

Em Log4j, o *logger* é a estrutura responsável por gerenciar requisições de mensagens de *logging*. O Asplog incorporou essa estrutura como parte de sua solução. Em geral, um *logger* precisará estar vinculado a cada configuração de *logging* (Seção 5.4) criada para fazer o controle de requisições.

O Mecanismo de Configuração de *Loggers* tem a função de concentrar em um único local todos os *loggers* criados de um projeto, tornando-os mais fáceis de

visualizar e configurar. Na Figura 3, é mostrada a interface gráfica referente a esse mecanismo. O conceito de hierarquia de *loggers* definido em Log4j, que se baseia no nome de cada *logger*, foi adotado na visualização deles a partir de um modelo em árvore. Algumas configurações de um *logger* podem ser editadas, tais como: o nível de relevância das mensagens que ele é capaz de tratar; os destinos das mensagens de saída – console e arquivo; e, no caso de haver saída no arquivo, o nome desse arquivo.

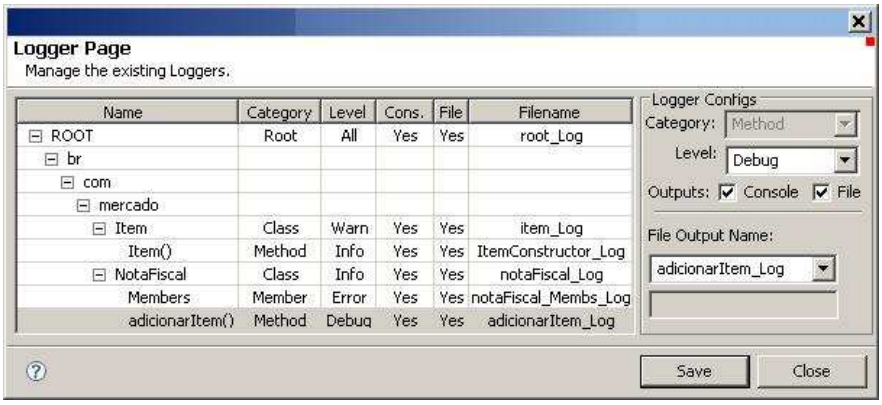

Fig. 3. Interface gráfica para centralização dos *loggers* de um projeto.

Uma das principais vantagens desse mecanismo é a de simplificar o processo de ativação e desativação de uma configuração de *logging*. Isso se resume a alterar, através da interface, o nível das mensagens suportadas pelo *logger* relacionado a uma determinada configuração. Dessa forma, essa funcionalidade agrega recursos importantes para que o Asplog atinja seus objetivos.

5.4 Mecanismos de Configuração de *Logging*

Dentre os mecanismos oferecidos pelo Asplog, os Mecanismos de Configuração de *Logging* possuem o papel mais importante. É a partir deles que o desenvolvedor pode criar suas próprias configurações de *logging*, que informam como os aspectos deverão ser criados pelo gerador de código.

Os mecanismos encarregados de oferecer essa funcionalidade são:

- *Mecanismo de Configuração de Logging Seletivo para Métodos* - permite configurar de forma seletiva e visualmente os métodos de uma classe que se deseja monitorar por *logging*.
- *Mecanismo de Configuração de Logging por Filtro para Métodos* - permite configurar por meio de filtros e visualmente os métodos de uma classe que se deseja monitorar por *logging* (Figura 4).

- *Mecanismo de Configuração de Logging Seletivo para Membros* - permite configurar de forma seletiva e visualmente os membros de uma classe que se deseja monitorar por *logging*.

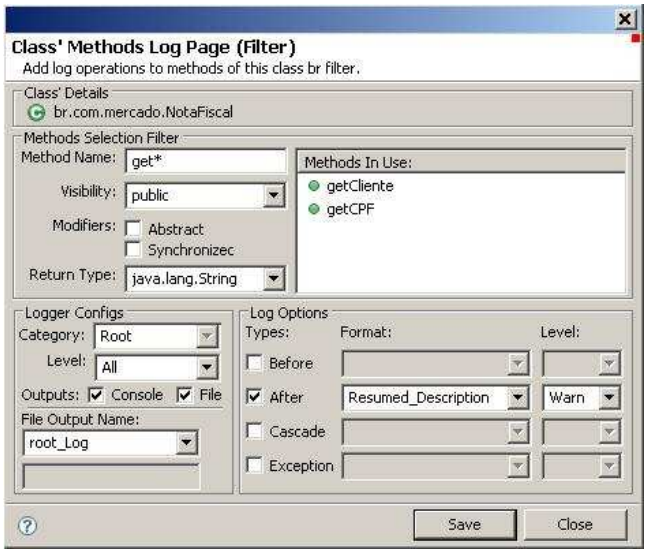

Fig. 4. Interface gráfica para criação de configuração de *logging* por filtro para métodos.

A fim de ilustrar um desses mecanismos, é apresentada a Figura 4. No agrupamento *Methods Selection Filter*, pode-se fitrar os métodos que serão monitorados a partir do nome, da visibilidade, dos modificadores e do tipo de retorno. O agrupamento *Logger Configs* define o *logger* responsável por essa configuração. Os pontos que serão monitorados em relação à execução de um método são definidos no agrupamento *Log Options*, podendo ser antes (*before*), depois (*after*), em cascata (*cascade*), que permite monitorar a execução de um método e de todos os outros chamados recursivamente a partir dele, ou quando uma exceção (*exception*) ocorrer. A definição desses pontos se baseia nos recursos disponibilizados pelo elemento *adendo* de AspectJ. Para cada um dos pontos selecionados, deve-se associar um modelo de mensagem reutilizável (*format*) e um nível de prioridade (*level*).

5.5 Mecanismo de Geração de Código

Os outros três mecanismos têm a função de criar as configurações relacionadas à *logging* e armazená-las em arquivos XML. Enquanto isso, o gerador de código é responsável por fazer a leitura desses arquivos para concretizar as configurações através da criação de aspectos e classes que cuidarão do *logging*.

Para possibilitar a geração dessas construções, foram definidos modelos (*templates*) dos aspectos e das classes que serão gerados. Cada modelo possui uma estrutura fixa e outra variável. Esta última possui *tags* personalizadas que são substituídas de acordo com as configurações criadas pelo usuário e armazenadas em XML. Por exemplo, a tag <METODO> seria substituída pelo nome do método escolhido.

Exemplo simplificado de um modelo de aspecto utilizado pelo gerador para concretização de uma configuração de *logging* por filtro para métodos.

```
public aspect <ASPECTO> {
  private static Logger <CLASSE>.<LOGGER> = ...
  private pointcut methodExec() : execution
    (<VISIBILIDADE> <TIPO_RETORNO> <CLASSE>.
     <FILTRO_NOME>(..)); ...
  before(): methodExec() {...} ...
}
```

6 Conclusões

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.

O Asplog disponibiliza um conjunto de características interessantes que dão suporte ao desenvolvedor no gerenciamento do *logging*:

- Utilização de recursos visuais, por meio de interfaces gráficas;
- Flexibilidade na criação de modelos de mensagem reutilizáveis para *logging*;
- Centralização dos *loggers* responsáveis por despachar os pedidos de *logging*, o que facilita configurá-los;
- Conjunto de opções para definir os pontos de uma aplicação a serem monitorados;
- Gerador automático de aspectos e classes responsáveis pelo *logging*.

Como consideração final em termos de contribuição deste trabalho, 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.

Finalmente, cabe destacar que essa é uma área de pesquisa ainda pouco explorada e que possui boas oportunidades, como por exemplo, abstrair a complexidade por detrás de outros interesses transversais através de uma abordagem orientada a aspectos. Assim, trabalhos futuros podem ser realizados como o propósito aprofundar os estudos científicos sobre o tema.

Referências

1. Couto, C.: Um Arcabouço Orientado por Aspectos Para Implementação Automatizada de Persistência. Master's Thesis, Department of Computer Science - Minas Gerais Federal University, Brazil (2006)
2. Davies, J., Huismans, N., Slaney, R., Whiting, S.: An Aspect Oriented Performance Analysis Environment. In: International Conference on Aspect-Oriented Software Development. Boston (2003)
3. Eclipse, http://www.eclipse.org
4. Elrad, T., Kiczales, G., Aksit, M., Lieberher, K., Ossher, H.: Discussing Aspects of AOP. Communications of the ACM, vol. 44, pp. 33--38. ACM Press, New York (2001)
5. Gulcu, C.: Log4j: The Complete Manual. QOS.ch, Lausanne (2004)
6. Gupta, S.: Logging in Java with the JDK 1.4 Logging API and Apache Log4j. Apress, New York (2003)
7. Happel, H. J., Schmidt, A.: Knowledge Maturing as a Process Model for Describing Software Reuse. In: 4th Conference Professional Knowledge Management - Experiences and Visions, 9th International Workshop on Learning Software Organizations, vol. 2, pp. 155--164. Potsdam (2007)
8. Hughes, D., Greenwood, P., Blair, L.: Aspect Testing Framework. In: Formal Methods for Open Object-Based Distributed Systems and Distributed Applications and Interoperable Systems. Paris (2003)
9. 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, pp. 262--263. IEEE Computer Society, Zurich (2004)
10. Java Architecture for XML Binding (JAXB), https://jaxb.dev.java.net
11. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G.: An Overview of AspectJ. In: Proceedings of the European Conference on Object-Oriented Programming. LNCS, vol. 2072, pp. 327--355. Springer-Verlag, Budapest (2001)
12. 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. LNCS, vol. 1241, pp. 220--242. Springer-Verlag, New York (1997)
13. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications, Greenwich (2003)
14. Log4E, http://log4e.jayefem.de
15. Log4j, http://logging.apache.org/log4j