

Programação Orientada a Aspectos como Mecanismo para o Desenvolvimento de Aplicações de Log com Baixo Acoplamento.

Rafaell Pinheiro Sousa
Ygor de Oliveira de Carvalho
Nielson José Pontes da Silva Júnior
Herbert Anderson de Vasconcelos Dantas
Edemberg Silva da Rocha
Heremita Brasileiro Lira
Thiago Moura
Crishane Azevedo Freire

Centro Federal de Educação Tecnológica da Paraíba (CEFET-PB)

rafaellps@gmail.com
ygor.oliveria.carvalho@gmail.com
nielsondasilvajr@gmail.com@gmail.com
herbert.anderson@gmail.com
edemberg@cefetpb.edu.br
heremita@cefetpb.edu.br
thiagomoura@cefetpb.edu.br
crishane@cefetpb.edu.br

Resumo: *Soluções disponíveis oferecidas através de frameworks para implementação de registro de logs em Java, não permitem o desenvolvimento de um módulo independente, o que gera acoplamento, comprometendo a reusabilidade do sistema. O espalhamento e entrelaçamento de código impactam negativamente na qualidade do software, dado o acoplamento gerado. Neste contexto, a Programação Orientada a Aspectos (POA) nos oferece meios que viabilizam a implementação de um sistema de logs, sem espalhamento e entrelaçamento de código, permitindo que o sistema possa ser organizado e decomposto em assuntos gerenciáveis e de fácil manipulação. No núcleo de pesquisa e desenvolvimento do Projeto SIEP do CEFET-PB foi realizado um estudo de tecnologias que viabilizassem a implementação de logs de forma desacoplada ao sistema, sem a necessidade de recodificação. Como resultado, identificamos a POA como uma alternativa adequada para esta tarefa.*

Palavras-chave: *modularidade, auditoria, log, Programação Orientada a Aspectos, Programação orientada a objetos, siep.*

Abstract: *The available framework solutions for software logging in Java do not permit the development of a independent module, generating coupling, witch compromises the system reusability. The code spreading and interlacing have a negative impact over the software quality, given the coupling created. In this context, the Aspect Oriented Programming (AOP) offers means witch enables the implementation of a logging system, without code spreading and interlacing, allowing the system to be organized and decomposed in manageable and easily handled subjects. At the core of Project SIEP was made a study of technologies that permit the implementation of logs uncoupled from the system, without rewriting the code. As a result, AOP was identified as a suitable alternative for this task.*

Key-words: *modularity, auditing, logging, aspects oriented programming, object-oriented programming, siep.*

1.Introdução

Auditoria de um sistema é o processo que rastreia as atividades do usuário persistindo tipos selecionados de eventos no *log* de segurança, seja em banco de dados, arquivo, ou outra forma de armazenamento. *Logs* de segurança contêm várias entradas de auditoria, que devem fornecer informações como: a ação que foi executada, o usuário que executou a ação, o êxito ou falha do evento, quando ele ocorreu e informações adicionais.

Uma política de segurança é determinada de acordo com as necessidades de uma organização, e esta tem relação direta com o nível de auditoria a ser utilizado. Embora a auditoria possa oferecer informações valiosas, o excesso de registros pode preencher o *log* com informações desnecessárias, além de impactar negativamente no desempenho da aplicação. Por isso, faz-se necessário a elaboração de todo um levantamento de requisitos, onde serão mapeados os eventos de interesse passíveis de registro de *log*, bem como, que tipo de informações o sistema de *log* deverá prover e como essas informações serão disponibilizadas. Os tipos mais comuns de eventos a serem submetidos à auditoria ocorrem quando: estados de objetos são alterados, atividades em banco são realizadas, usuários fazem *logon* e *logoff* no sistema.

Para a implementação de um sistema de *logs* faz-se necessária a utilização de recursos que facilitem o desenvolvimento desse tipo de aplicação. Diversas tecnologias são oferecidas para este fim, porém é preciso analisar uma boa alternativa, o menos intrusiva possível, que ofereça artifícios para que o sistema de *log* represente um módulo desacoplado ao sistema de fato. Este trabalho tem o objetivo de apresentar uma alternativa para a implementação deste, através do uso da programação orientada a aspectos, sem a necessidade de recodificação do sistema legado, e de forma que o sistema de *log* seja um módulo independente do sistema, para promover a reusabilidade do mesmo.

2.Desenvolvimento

A funcionalidade de *log* representa um requisito não funcional do sistema. Requisitos não funcionais, como o nome sugere, são aqueles não diretamente relacionados as funções específicas fornecidas pelo sistema(SOMMERVILLE, 2007, p.82). Partindo desta concepção, torna-se

necessário estruturar o sistema de *log* como um módulo a parte, não inerente ao negócio real da aplicação. A modularização tem a vantagem de reduzir a complexidade do problema, dividindo-o em sub-problemas mais simples. Com base nos requisitos levantados que definem as responsabilidades do sistema de *log*, temos respaldo para a implementação.

Para este propósito, as soluções oferecidas por paradigmas existentes, a exemplo da Programação Orientada a Objetos (POO) não oferecem boas alternativas para codificação do sistema em foco de forma desacoplada, por mais que se faça o uso de padrões de projeto específicos. A POO conta com *frameworks* variados voltados especificamente para oferecer suporte a *logs*. Um *framework* (ou *framework* de aplicação) é um projeto de subsistema composto por um conjunto de classes abstratas e concretas, e as interfaces entre elas (WIRFS-BROCK e JOHNSON, 1990). A utilização da maioria destes *frameworks* em JAVA, a exemplo do *Log4J*, exigem a instanciação local de métodos deste em cada classe que deseja ter suas operações registradas em *log*. Além disso, cada método deverá realizar as chamadas de registro de *log* internamente. Se não for utilizada nenhuma espécie de *framework*, é necessária a implementação de uma classe responsável pelo *log* do sistema e como praticamente todo método necessita que alguns dados sejam registrados em *log*, as chamadas a essa classe seriam espalhadas por toda a aplicação.

Modularizar requisitos periféricos que atravessam múltiplos módulos, os requisitos transversais, utilizando somente POO pode ser uma tarefa complicada. exemplos: *logging*, integridade de transações, autenticação, segurança, desempenho, distribuição, persistência e *profiling* [Elrad et al., 2001]. Quando é necessário a implementação destes tipos de requisitos, as soluções que utilizam puramente POO, geram espalhamento de código, e entrelaçamento (requisito transversal misturado com regras de negócio), o que torna o código não reusável de difícil manutenção, acoplado, menos coeso e com muitas linhas de código adicionais.

2.1 - Descrição geral sobre POA

A POA surgiu a partir da identificação de uma dificuldade na programação orientada a objetos em modularizar certos tipos de interesses. Seu objetivo é romper os limites alcançados pelo

paradigma da Programação Orientada a Objetos (RESENDE; SILVA, 2005, p. 41). Esses limites, não permitem a redução dos níveis atuais de complexidade de desenvolvimento de software, impedindo que os processos e produtos se tornem mais gerenciáveis e flexíveis (RESENDE; SILVA, 2005, p. 01). Esta técnica de programação permite a separação de interesses de um sistema, tornando possível o agrupamento de requisitos em trechos bem definidos, denominado *separation of concerns*. O princípio de separação é importante para diminuir a complexidade em certos momentos, permitindo o entendimento e análise de uma única característica por vez. Entretanto, um sistema é naturalmente composto de diferentes características, e entendê-las em conjunto é igualmente importante. Se por um lado, precisamos separar as características, por outro precisamos compô-las. Muitas vezes estas características são fortemente relacionadas, entrelaçadas e/ou sobrepostas, influenciando ou restringido umas às outras, sendo chamadas de características transversais (TEKINERDOĐAN, 2004). Esse tipo de comportamento é conhecido como um comportamento que *crosscutting* (atravessa) o sistema . Na POO a unidade natural de modularização é a classe, e um comportamento do tipo *crosscutting* está espalhado em varias classes. Trabalhar com código que aponta para responsabilidades que atravessam o sistema gera problemas que resultam na falta de modularidade. Nesse ponto a POA ajuda a manter a consistência do projeto, apresentando um novo nível de abstração, os aspectos. Prezar a separação de responsabilidades de uma aplicação viabiliza a modularização do sistema a ser desenvolvido, pois cada módulo terá suas atribuições e responsabilidades bem definidas, evitando espalhamento de código. O aumento da modularidade implica em sistemas mais legíveis e reutilizáveis, os quais são mais facilmente projetados e mantidos.

A POA não tem o intuito de ser um novo paradigma de programação, mas uma nova técnica que deve ser utilizada em conjunto com linguagens de programação para construção de sistemas de *software* de melhor arquitetura, auxiliando na manutenção dos vários interesses e a compreensão do *software*. Entretanto, não é um antídoto para um *design* ruim ou insuficiente (ELRAD et al.,2001).Sua principal contribuição é permitir que o desenvolvedor modifique um modelo OO para criar um sistema que pode crescer e cumprir novos requerimentos. Assim como os objetos do mundo real podem mudar de estado durante seu ciclo de vida, uma aplicação pode adotar novas características seguindo a

evolução. A POA nos ajuda a implementar funcionalidades sem realizar intervenções diretas no código pré-existente. Através desta, podemos monitorar todo o funcionamento de uma aplicação e suas respectivas chamadas de métodos, o que nos possibilita realizar tratamentos adicionais ou incrementos para atender a novos requisitos.

Para utilização de POA em Java, é necessário o uso de algumas extensões que oferecem suporte para tal, a exemplo o *AspectJ*. *AspectJ* (KICZALES et al., 2001) é uma extensão orientada a aspectos de propósito geral da linguagem Java. Além dos elementos oferecidos pela POO como classes, métodos, atributos e etc, são acrescentados novos conceitos e construções ao *AspectJ*, tais como: aspectos (*aspects*), conjuntos de junção (*pointcuts*), pontos de junção (*join points*), advices (*advice*s) e declarações inter-tipos (*inter-type declarations*)¹.

Aspectos: através da palavra reservada *aspect*, definimos uma classe de aspectos, que deve reunir toda estrutura necessária a um aspecto: *advice*s e *pointcuts*. Um aspecto é um componente que contém uma estrutura de *pointcuts* (um conjunto de pontos de junção declarados) que podem convenientemente executar *advice*s (outra função que não pertence ao fluxo normal do programa). (RESENDE; SILVA, 2005, p. 42).

Pointcuts: representa uma variável que armazena uma lista contendo as assinaturas de métodos que se tem interesse em interceptar. Esta interceptação é feita com o objetivo de alterar o fluxo original do programa e inserir novas chamadas, *advice*s. Um *pointcut* está para atributos assim como *aspect* está para uma classe. (RESENDE; SILVA, 2005, p. 45).

*Advice*s: equivalem a métodos ou funções e são formados por um conjunto de instruções que poderão ser executados no momento que um *joinpoint* for alcançado.

JoinPoint: É um ponto durante a execução do programa que será afetado pelo aspecto. Este ponto pode ser a execução de um método, a modificação do valor de uma variável ou outros. Diferentes linguagens orientadas a aspectos implementam diferentes tipos de *joinpoint*.

Weaving: É o mecanismo responsável por ligar o aspecto aos objetos da aplicação. Isto pode ocorrer em tempo de compilação ou em tempo de execução. No *AspectJ*, o processo de *weaving* ocorre sempre em tempo de compilação.

Neste contexto, a aplicação da POA para a implementação de um sistema de *log*, nos oferece meios para intervir em classes e métodos, sem codificação direta, não exigindo nenhuma intervenção no código legado. Isso é possível através da implementação de camadas de

aspectos, o que possibilita a organização e decomposição do sistema em assuntos gerenciáveis e de fácil manipulação. Para um conjunto de requisitos passíveis a geração de *logs* de uma classe, é definido um aspecto. Este aspecto irá funcionar como uma espécie de filtro, que é ativado toda vez que determinado método ou operação for executada. A interceptação ocorre em tempo de execução, e dinamicamente podemos alterar o fluxo da aplicação se desejarmos. Uma interceptação pode trazer consigo, se assim definido no *advice*, todo o contexto do método, oferecendo ao programador a possibilidade de captura dos parâmetros recebidos, retornos, exceções lançadas, bem como meios para intervenção direta, que acarretam na alteração do código original.

2.2 - Implementação do sistema de LOG baseado em POA

Identificada a tecnologia e levantados os requisitos e modelagem do sistema de *log*, a implementação deste pode ser realizada. No tocante a POA, serão necessárias a criação de aspectos para as classes que possuem métodos que requeiram registros após interação e os *advices* associados.

Os *advices* irão definir a ação que deverá ser tomada ao atingir certo *pointcut*, para o caso específico, são os métodos responsáveis por identificar natureza da operação, e estado de objetos, para o registro do *log* correspondente. Nenhuma parte do código legado sofre nenhum tipo de alteração, que caracteriza a boa modularidade e não invasividade propiciada pelo uso da POA. Entre os principais requisitos identificados para o sistema de *log*, merecem destaque: persistência em banco de dados, interface web para consulta dos *logs*, identificar quem executou a operação, qual a operação, o que mudou e quando alteração foi executada, registrar em *log* toda atividade que resultasse em registro permanente em banco de dados (*insert*, *update*, *remove*).

Para a persistência dos registros em banco, identificamos como melhor alternativa a utilização do *Java Persistence Api* (JPA,) um padrão que especifica a persistência de objetos com mapeamento Objeto-Relacional na plataforma JavaEE (EJB3) e JavaSE, além de uma linguagem de consulta, o JPQL. Utilizando uma de suas implementações (*TopLink Essentials*), temos a possibilidade modularizar a funcionalidade de registro em banco de dados. Para a visão, camada responsável por oferecer uma interface para a consulta dos *logs* gerados,

com variados filtros, utilizamos o JSF (*Java Server Faces*). O JSF é um framework da linguagem Java que oferece ao programador toda infra-estrutura necessária para auxílio a produtividade e modularidade na construção de páginas web. Para a implementação dos requisitos de identificação de usuários que efetuaram a operação, qual operação foi realizada e o que mudou, identificando estados do objeto anteriores e posteriores a alteração, foi utilizada a POA.

Com exceção da POA, todas as outras tecnologias adotadas para o sistema de *log*, já eram as utilizadas na aplicação que iria agregar esta funcionalidade, o que trouxe clareza ao sistema como um todo, dado a padronização.

3. Conclusão

Para o desenvolvimento de um sistema de *log*, é necessário identificar várias interações relevantes realizadas no contexto da aplicação. Para registrar essas interações o uso de uma boa estratégia de implementação é muito importante, para manter a clareza e modularidade do sistema. A POA, se mostra como uma ótima alternativa, no tocante a desenvolvimento de novos requisitos, pois provê um conjunto de artifícios que possibilitam a implementação destes, mantendo a coesão e divisão de responsabilidades da aplicação. No núcleo de pesquisa e desenvolvimento do Projeto SIEP do CEFET-PB necessitávamos implementar um sistema de registro de *logs*, com fins de auditoria. Desta forma, foi realizado um estudo tomando como base a Programação Orientada a Objetos para a implementação de um sistema de *log* de forma desacoplada ao sistema, sem a necessidade de recodificação. Como resultado deste estudo identificamos a POA como uma alternativa adequada para esta tarefa.

4. Referências

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; Palm, J.; Griswold, W.G. **An overview of AspectJ**, In Knudsen, J. L., editor, proceedings of the European Conference on Object-Oriented Programming (ECOOP), Berlin, pg 327-353, Springer-Verlag, 2001.

KISELEV, I. **Aspect-Oriented Programming with AspectJ**, Ed. Sams Publishing, 2002.

RESENDE, A.; SILVA, C. **Programação orientada a aspectos em Java**, Ed. Brasport, Rio de Janeiro, 2005.

SOARES, S.; BORBA, P. *AspectJ* - **Programação orientada a aspectos em Java**, In: VI Simpósio Brasileiro de Linguagens de Programação, Rio de Janeiro, 2002.

SOMMERVILLE, I. **Engenharia de Software**. 8ª Ed. Tradução: MELNIKOFF, S. S. S.; ARAKAKI, R.; BARBOSA, E. A. São Paulo, Pearson Addison-Wesley, 2007.

TEKINERDOĐAN, A. MOREIRA, J. ARAÚJO, P. CLEMENTS, **Early aspects: aspect-oriented requirements engineering and architecture design**, Report Early Aspects Workshop at AOSD, England, 2004.

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