Paradigmas Orientado a Aspectos (POA)

Giovani Zanella da Maia
Instituto Federal Catarinense
Blumenau–SC, Brasil
giovanizanelladamaia123@gmail.com

Resumo

O POA foi formalizada por Gregor Kiczales e sua equipe no PARC em 1997. Um paradigma que surgiu com o propósito de enfrentar a modularidade e manutenção de sistemas complexos, focando em preocupações transversais como login, segurança e gestão de transações.

O paradigma orientado a aspectos se diferencia de paradigmas tradicionais como orientação a objetos, procedural e funcional, ela separa os interesses transversais em módulos que chamamos de aspectos, que além de enfrentar os desafios, também deixa um código limpo e coeso. Será abordado como são sistemas sem a programação orientada a aspectos, fazendo uma comparação, para visualizar melhor os desafios e como seria a proposta de minimizá-los.

Será abordado cada fundamento deste paradigma, com aspectos que ajudam a organizar comportamentos comuns, como login e segurança, em um só lugar. Os pointcuts indicam exatamente onde esses aspectos devem ser aplicados no código, e os advices dizem o que deve ser feito nesses pontos. Isso tudo faz com que os desenvolvedores possam focar nas funcionalidades principais do sistema, deixando o código mais limpo e fácil de manter.

Palavras-chave

Paradigmas Orientação a Aspectos, Separação de Interesses, Modelagem Orientada a Aspectos.

Introdução

Com o avanço constante da tecnologia e o aumento da complexidade dos sistemas de software, os paradigmas de programação evoluíram para atender às novas demandas do mercado, continuando a evoluir cada vez mais. A Programação Orientada a Aspectos (POA) é um paradigma relativamente novo que surgiu para enfrentar os desafios de modularidade, que consiste na divisão de um sistema em partes menores (módulos), e de manutenção, sendo a facilidade com que um sistema pode ser modificado, corrigido ou melhorado. Isso é especialmente importante em sistemas de software complexos que possuem diversas preocupações transversais.

Preocupações transversais, como login, segurança e gestão de transações, são aspectos que se espalham por múltiplos módulos de um sistema, dificultando a modularidade e a manutenção do código. A POA facilita a manutenção e evolução do código, reduzindo a complexidade e melhorando a legibilidade e reutilização. Com a evolução dos paradigmas de programação, como as que foram citadas no documento de aula, a programação procedural, a orientação a objetos e a programação funcional, houve avanços significativos na forma como desenvolvemos e gerenciamos o software. No entanto, os paradigmas tradicionais possuem dificuldades em lidar com preocupações transversais, o que levou ao surgimento da POA.

A POA oferece uma abordagem que expande os paradigmas tradicionais, como a Programação Orientada a Objetos (POO), proporcionando uma maneira mais eficaz de lidar com a modularidade e manutenção em sistemas complexos. Ao modularizar preocupações transversais em aspectos, a POA permite que desenvolvedores foquem mais nas funcionalidades principais do sistema, resultando em um código mais limpo e fácil de gerenciar. Assim, a POA representa uma evolução significativa na forma como abordamos o desenvolvimento de software, oferecendo soluções práticas para problemas que os paradigmas tradicionais ainda lutam para resolver.

1.0 O que é a Programação Orientada a Aspectos (AOP)?

1.1 História

A Aspect-Oriented Programming foi desenvolvida como resultado do trabalho de pesquisa realizado pelos pesquisadores do Centro de Pesquisa de Palo Alto da Xerox durante as décadas de 1980 e 1990. A necessidade de introduzir um novo paradigma de programação tornou-se óbvia porque os novos paradigmas herdam sistematicamente as limitações intransponíveis do anterior, sendo impedido pela incapacidade de modular o interesse cruzado. A Aspect-Oriented Programming foi formalizada por Gregor Kiczales e sua equipe no PARC em 1997. AspectJ, uma extensão orientada a aspectos do Java, foi a primeira implementação significativa do paradigma,

apresentada em 2001. Desde então, o AOP tem se exacerbado, vários frameworks e extensões apareceram para uma variedade de linguagens de programação populares, como C++, Python e Ruby.

1.2 Surgimento e desafios

Um sistema possui características, princípios, requisitos, algoritmos ou ideias. Esse conjunto de semelhanças se chama conjunto de interesses, foi com base nisso que a Programação Orientada a Aspectos (POA) surgiu, como uma resposta aos desafios enfrentados no desenvolvimento de software, particularmente em relação à manutenção e escalabilidade de sistemas complexos. Dois problemas fundamentais que a POA visa abordar são o entrelaçamento (tangling) e o espalhamento (scattering). Mas quando esses desafios ocorrem?

O entrelaçamento ocorre quando vários interesses foram implementados na mesma classe. Acredito que os sistemas são imprevisíveis, não sabemos o que será implementado ou alterado futuramente, e conforme o sistema cresce será mais difícil separar esses interesses. Para facilitar a compreensão, tome como exemplo a imagem abaixo, podemos dizer que a implementação em vermelho seria de segurança e a verde de log.

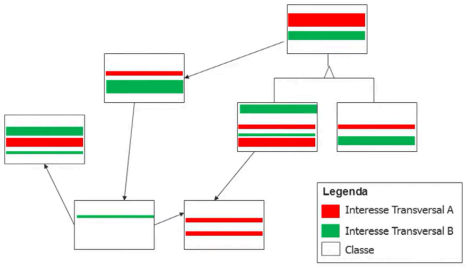

Figura 1: Diagrama de um sistema com entrelaçamento, sem POA[4]

Por outro lado, o espalhamento acontece quando um único interesse está espalhado por várias classes do sistema.

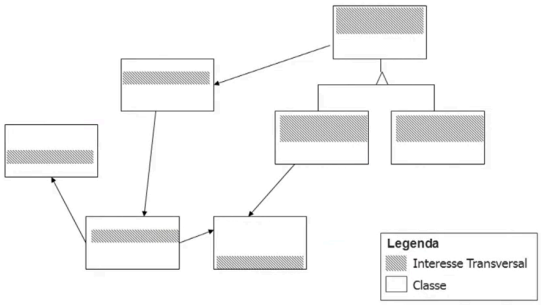

Figura 2: Diagrama de um interesse espalhado em várias classes, sem POA. [4]

Uma solução com aspectos seria encapsular de uma forma modular esses interesses, como no exemplo acima, separamos em dois aspectos diferentes o código de log e segurança, que serão aplicados aos pontos de junção especificados, assim como feito na imagem abaixo.

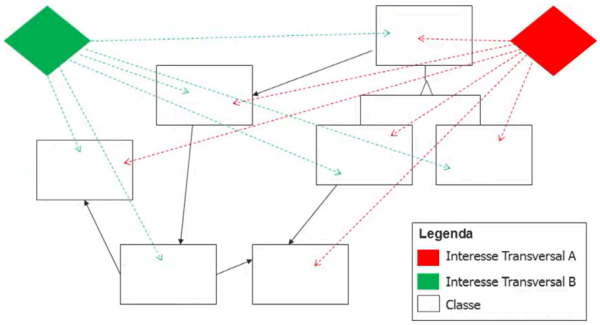

Figura 3: Diagrama de interesses, em um sistema utilizando POA.[4]

Como resultado, a POA reduziu significativamente o entrelaçamento e o espalhamento, melhorando a modularidade e a manutenção do código, é importante notar que a POA não elimina completamente o entrelaçamento e o espalhamento em todos os casos. Ela oferece uma estrutura para gerenciar melhor esses desafios, permitindo que os desenvolvedores organizem o código de uma maneira que separe mais claramente os diferentes interesses do sistema.

2.0 Como os aspectos se diferenciam de classes e objetos na OOP

Como já introduzido, os aspectos são comumente utilizados em conjunto com a programação orientada a objetos e outras tecnologias, é uma abordagem interessante, pois o que você não consegue modularizar com aspectos, poderá

utilizar a orientação a objetos, eles podem ser usados para complementar mutualmente.

A diferença entre ambos os paradigmas é que na POO, o código é organizado em instâncias de classes que encapsulam o estado e comportamento relacionado a uma entidade, utilizando os 4 pilares da POO, que seria o encapsulamento, abstração, herança e polimorfismo. Já a POA possui uma abordagem diferenciada, onde identificamos os interesses transversais e separamos em módulos chamados aspectos, utilizamos os aspectos no código de uma forma mais coesa, esse conceito de aspecto é utilizado em conjunto com pointcuts (pontos de corte) e advices (conselhos) que definem onde e como os aspectos devem ser aplicados no código, veremos com mais detalhes esses conceitos.

3.0 Como os aspectos se diferenciam de classes e objetos na OOP

3.1 Aspecto

Resumiria aspectos como uma unidade modular que encapsula comportamentos transversais. Pense em aspectos como um conceito básico, assim como uma instância de classe é um conceito fundamental.

> *"Aspecto é uma parte de um programa POA que separa os interesses ortogonais dos interesses principais"* [12].

> *"Um aspecto, como uma classe Java, pode definir membros (atributos e métodos) e uma hierarquia de aspectos, através da definição de aspectos especializados"[3]*

A anotação @**Aspect** define uma classe como um aspecto em java. E seu bloco de código abriga todos os elementos da programação orientada a aspecto.

```java
// Definição de um aspecto
@Aspect
public class LoggingAspect {

    // Definição de um advice
    @Before("execution(* com.example.service.*.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("Log Before : " + joinPoint.getSignature().getName());
    }
}
```

Figura 4: definindo um aspecto

3.3 Join point ou ponto de junção

Join Points são locais específicos onde os Aspectos podem ser aplicados. Esses pontos definidos de execução são onde o código do aspecto e o código da aplicação principal podem se unir. Isso pode ocorrer de algumas formas, como a chamada e a execução de métodos, construtores, a acessibilidade de parâmetros, a inicialização de objetos, a manipulação de exceções, etc. Eles fornecem ao aspecto um meio de chamar e modificar o comportamento do programa em locais particulares de execução sem alterar o código principal.

"O ponto de junção é um ponto bem definido na execução de um sistema" [12]

No exemplo abaixo será mostrado o funcionamento, e a pilha de execução. Temos uma classe chamada Exemplo e ela possui um método tarefa.

```java
public class Exemplo {
    public void tarefa() {
        System.out.println("Executa tarefa");
    }
}
```

Figura 5:

Após criar a classe LoggingAspect que é um aspecto, o Pointcut define qual Join Points serão interceptados pelo aspecto, nesse exemplo estamos interceptando o join point Exemplo.tarefa(). Com o Advice**,** um join point pode ser executado antes e depois da execução desse método.

```java
import org.aspectj.lang.annotation.*;

@Aspect
public class LoggingAspect {

    // Definição do Pointcut
    @Pointcut("execution(* Exemplo.tarefa(..))")
    public void executaTarefaPointcut() {}

    // Before Advice
    @Before("executaTarefaPointcut()")
    public void logAntes() {
        System.out.println("log antes");
    }

    // After Advice
    @After("executaTarefaPointcut()")
    public void logDepois() {
        System.out.println("Log depois");
    }
}
```

Figura 6:

No código principal criamos um objeto de Exemplo e chamamos o método tarefa.

Fluxo de Execução ou como alguns programadores costumam chamar de pilha de execução do programa.

1. **Classe principal:** O método main chama o método tarefa que pertence à classe Exemplo.
2. **Interceptação:** intercepta a chamada ao método tarefa em Exemplo, como um Join Point.
3. **Advice (@Before)**: antes de executar o método tarefa, ele executa o logAntes, imprimindo "Log antes".
4. **Realização do Método Original:** O método da tarefa é executado, exibindo a mensagem "Tarefa em execução".
5. **Advice (@After)**: depois da execução do método tarefa, O logDepois é executado, imprimindo "Log depois".
6. **Continuação da Execução**: A execução retorna ao chamador original (main).

3.3 Pointcuts ou ponto de corte

Pointcuts são responsáveis por selecionar pontos específicos (join points) na execução de um programa onde aspectos devem ser aplicados. Eles determinam onde e quando a lógica adicional definida em aspectos será interceptada e executada. No Spring AOP, baseado em AspectJ, você pode definir pointcuts usando a anotação @Pointcut. Eles podem ser nomeados ou anônimos, para definir um pointcut nomeado deve usar a sintaxe 'pointcut <nome>(Argumentos): <corpo>;'.

Pointcuts podem ser encapsulados usando os modificadores de acesso private, public e protected. Além disso, pointcuts não podem ser sobrecarregados (overloaded), mas podem ser declarados como abstratos, desde que dentro de aspectos abstratos. Também podem ser compostos por operadores lógicos, como '!' (não), '||"(ou) e '&&'(E).

Um exemplo de um pointcut nomeado pode ser visto na figura 6 na a linha 8 do código.

3.3 Advice ou conselho

Os conselhos representam partes de código executadas em pontos específicos da execução do programa, definidos pelos pontos de conexão. Os conselhos permitem a implementação de comportamentos adicionais sem modificar o código-fonte principal, promovendo a separação de preocupações e melhorando a modularidade do software de maneira dramática. Existem três principais tipos de conselhos, cada um definido pelo momento em que o código é executado em relação ao ponto de junção:

Antes do Conselho: O antes do conselho é executado antes do ponto de junção. Isso significa que qualquer lógica contida no antes do conselho será processada antes que o ponto de conexão especificado pelo ponto de conexão seja alcançado. Esse tipo de conselho serve para verificar condições prévias complexas ou inicializar recursos essenciais antes que o comportamento principal ocorra de forma explosiva.

Em torno de conselho: O em torno de conselho envolve o ponto de junção, permitindo que o código seja executado tanto antes quanto depois do ponto de junção. Ele é o tipo mais poderoso de conselho, pois pode controlar a execução do método alvo, decidir se o método alvo deve ser chamado ou não, e modificar os valores de retorno ou exceções lançadas.

Após o conselho: O após o conselho é executado após a conclusão do ponto de junção, independentemente do método alvo ter sido concluído com sucesso ou ter lançado uma exceção. Este tipo de conselho é útil para liberar recursos ou executar ações de limpeza.

Além desses, existem variações como o conselho após o retorno, executado somente se o método alvo retornar normalmente, e o conselho após o lançamento, que é executado se o método alvo lançar uma exceção.

Conclusão

O paradigma da Programação Orientada a Aspectos se mostra uma ótima ferramenta na construção de sistemas de vários tipos, e sua versatilidade na forma que lida com preocupações transversais, se mostrando um ótimo paradigma, para trabalhar programas que demandam esse mesmo método. Foi abordado os conceitos fundamentais da POA, como seus conceitos, entendendo então como funcionam os aspectos, os pontos de corte e as junções e os conselhos, sendo as partes mais importantes de um sistema que usa essa estrutura. Foi demonstrado exemplos, de como cada fundamento se comporta no código, como podemos visualizar na pilha de execução. Além disso, a POA oferece uma maneira avançada de organizar um projeto, separando preocupações transversais do núcleo do programa, o que permite um funcionamento mais independente e eficiente. Isso facilita o desenvolvimento e a manutenção de um projeto com um planejamento inteligente. Em resumo, a Programação Orientada a Aspectos é muito importante para a modernização e inovação na engenharia de software, permitindo que desenvolvedores criem programas cada vez maiores e inovadores, sem perder o que já temos. Aprender e aplicar os princípios desse paradigma é essencial para um desenvolvedor que busca um projeto com eficiência.

Referencias

[1] T. Treinaweb. "O que é AOP: Programação Orientada a Aspectos". [Online].Available:https://www.treinaweb.com.br/blog/o-que-e-aop-programacao-orientada-a-aspectos. [Accessed: Jan. 4, 2016].

[2] G. Kiczales et al., "Aspect-Oriented Programming". Presented at the European Conference on Object-Oriented Programming (ECOOP), 1997. [Online].Available:https://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf. [Accessed: Jan. 4, 2016].

[3] AOPJ Group, "Aspect-Oriented Programming: Relatório Técnico". Universidade Federal de Santa Catarina (UFSC), 2003. [Online]. Available: https://lisha.ufsc.br/teaching/sce/ine6511-2003-2/work/aopj/relatorio.pdf. [Accessed: Jan. 4, 2016].

[4] Diariodeumacdf, "Programação Orientada a Aspectos". YouTube, 2020. [Online]. Available: https://www.youtube.com/watch?v=QJ4_nHbDr6Y. [Accessed: Jan. 4, 2016].

[5] DevMedia, "Programação Orientada a Aspectos". DevMedia, 2010. [Online].Available: https://www.devmedia.com.br/artigo-engenharia-de-software-10-programacao-orientada-a-aspectos/11912. [Accessed: Jan. 4, 2016].

[6]João Roberto Silva de Almeida, "Título da Monografia". Universidade Federal de Juiz de Fora (UFJF), 2010. [Online]. Available: http://monografias.ice.ufjf.br/tcc-web/exibePdf?id=8. [Accessed: Jan. 4, 2016].

[7] Autor Desconhecido, "AspectJ Android with Example". SlideShare, 2015. [Online]. Available: https://pt.slideshare.net/slideshow/aspectj-android-with-example/40759881#15. [Accessed: Jan. 4, 2016].

[8] C. Larman, Utilizando UML e Padrões: Uma Introdução à Análise e ao Projeto Orientados a Objetos e ao Processo Unificado, 3rd ed. Porto Alegre: Bookman, 2007.

[9] B. Meyer, Object-Oriented Software Construction, 2nd ed. New York: Prentice Hall, 1997.

[10] G. Kiczales et al., "Aspect-Oriented Programming". In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1997.

[11] E. Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley, 1995.

[12] R. Safonov, Using Aspect-Oriented Programming for Trustworthy Software Development. Wiley-Interscience, New Jersey, 2008.

[13] Herder, J. (2002). *Aspect-Oriented Programming with AspectJ*.

[14] Herder, J. *Aspect-Oriented Programming with AspectJ*. 2003. Disponível em: https://ulbra-to.br/encoinfo/wp-content/uploads/2020/03/aspectj-encoinfo2003.pdf

[15] Kiselev, I. (2002). *Aspect-Oriented Programming with AspectJ*. Ed. Sams Publishing.