UNIVERSIDADE FEDERAL DO RIO GRANDE
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

NICOLAS BENTO

Framework AOP utilizando técnicas de Bytecode Engineering

Trabalho de Conclusão apresentado como requisito parcial para a obtenção do grau de Tecnólogo em Análise e Desenvolvimento de Sistemas

Prof. Márcio Torres
Orientador

Rio Grande, fevereiro de 2014

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Bento, Nicolas

Framework AOP utilizando técnicas de Bytecode Engineering / Nicolas Bento. – Rio Grande: TADS/FURG, 2014.

49 f.: il.

Trabalho de Conclusão de Curso (tecnólogo) – Universidade Federal do Rio Grande. Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, Rio Grande, BR–RS, 2014. Orientador: Márcio Torres.

1. AOP, Bytecode Engineering, Meta-programação, Framework, SoC. I. Torres, Márcio. II. Título.

FOLHA DE APROVAÇÃO

Monografia sob o título *"Framework AOP utilizando técnicas de Bytecode Engineering"*, defendida por Nicolas Dias Bento e aprovada em ?? de ?? de ???, em Rio Grande, estado do Rio Grande do Sul, pela banca examinadora constituída pelos professores:

Prof. Márcio Torres
Orientador

Prof. NOME
IFRS - Campus Rio Grande

Prof. NOME
IFRS - Campus Rio Grande

"A mente que se abre a uma nova ideia jamais volta ao seu tamanho original."
— ALBERT EINSTEIN

AGRADECIMENTOS

Agradecimentos ...

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

AOP	*Aspect Oriented Programming* (Programação Orientada a Aspectos)
SoC	*Separation of concerns* (Separação de Interesses)
OOP	*Object Oriented Programming* (Programação Orientada a Objetos)
PARC	*Palo Alto Research Center* (Centro de Pesquisa Palo Alto)
SQL	*Structured Query Language* (Linguagem de Consulta Estruturada)
YAGNI	*You aren't gonna need it* (Você não vai precisar dele)
XP	*Extreme Programming* (Programação Extrema)
Javassist	*Java Programming Assistant* (Assistente de Programação Java)
BCEL	*Byte Code Engineering Library*
API	*Application Programming Interface* (Interface de Programação de Aplicativos)

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Resumo ...

Palavras-chave: AOP, Bytecode Engineering, Meta-programação, Framework, SoC.

1 INTRODUÇÃO

Atualmente o setor de desenvolvimento de software é um dos setores que mais cresce em todo o mundo, devido a esse grande crescimento a preocupação das empresas em desenvolver software de qualidade têm aumentado constantemente. A grande maioria das empresas atuantes no mercado, fazem uso do paradigma de Programação Orientada a Objetos (*Object Oriented Programming* - OOP) no desenvolvimento de seus softwares, com intuito de ter um produto manutenível e que se adapte a mudanças, mas com uma qualidade permanente.

Um dos principais motivos que fez com que OOP se tornasse um dos paradigmas mais utilizados foi a sua capacidade de encapsulamento, tornando o projeto de software mais claro, aumentando a reusabilidade de módulos, implementação simplificada e redução nos custos de manutenção. Porém existem interesses que não se aplicam à apenas um módulo, por exemplo o tratamento de exceções deveria estar encapsulado em um módulo distinto, mas na prática ele esta espalhado por diversas partes da aplicação (diversos módulos). Estes interesses são chamados de interesses transversais (*crosscuting concerns*).

Para encapsular um interesse transversal em um módulo único surgiu o paradigma de Programação Orientada a Aspectos (*Aspect Oriented Programming* - AOP)[1] que é uma extensão de OOP. Quando AOP é utilizado juntamente com OOP, pode-se garantir que independentemente do tipo de interesse (interesse central ou interesse transversal) tem-se um sistema altamente modularizado, sem espalhamento ou entrelaçamento de código.

A maioria das linguagens orientada à objetos, possuem ferramentas ou até outras linguagens que dão suporte a programação orientada a aspectos. A linguagem mais popular e madura que dá suporte a AOP para a linguagem Java é o AspectJ, porém o seu maior ponto fraco na opinião de diversos desenvolvedores é a complexidade no aprendizado,

[1] Em AOP interesses transversais recebem o nome de "aspectos".

por ser uma linguagem que possui uma sintaxe própria.

Para diminuir a complexidade na utilização de aspectos na linguagem Java, este trabalho propõe a criação de um *framework*[2] que implemente os principais conceitos de AOP, com o obejtivo de encapsular os interesses transversais, mantendo uma API simplificada, apenas utilizando as construções da linguagem Java, fazendo com que o desenvolvedor não precise aprender uma nova linguagem para a utilização do framework, apenas conheça a própria linguagem base.

1.1 Objetivo

Este trabalho tem como objetivo geral desenvolver um *framework* que aplique os conceitos básicos de Programação Orientada à Aspectos.

Os objetivos específicos deste trabalho são:

- Entender os principais conceitos de AOP.
- Aplicar os conceitos fundamentais de AOP no desenvolvimento de um *framework*.
- Utilizar técnicas de *Bytecode Engineering*.
- Desenvolver um estudo de caso para demostrar as funcionalidades do *framework* desenvolvido.

1.2 Motivação

Existe uma variedade de ferramentas que implementam AOP, porém estas ferramentas na maioria das vezes se tornam complexas na resolução de um simples problema. Esta complexidade durante o uso da ferramenta, acarreta em um tempo maior de aprendizagem, e consequentemente desencoraja o seu uso pelos usuários. Por este motivo, este trabalho propõe o desenvolvimento de um *framework* que resolva a maioria dos problemas que AOP se responsabiliza em resolver, porém a complexidade do *framework* desenvolvido será reduzida consideravelmente, pois será implementado apenas os conceitos necessários.

[2] Uma abstração entre classes comuns entre vários projetos, provendo uma funcionalidade genérica.

1.3 Resumo dos capítulos

Este trabalho está dividido em 9 capítulos, abaixo uma pequena descrição de cada um deles:

Capítulo 1 este capítulo traz uma visão geral do problema, também descreve os objetivos e a motivação encontrada para a realização deste trabalho.

Capítulo 2 este capítulo aborda os principais conceitos de Programação Orientada a Aspectos, e os principais benefícios no uso deste paradigma.

Capítulo 3 este capítulo descreve os conceitos básicos de *bytecode* e *Bytecode Engineering*, introduz também as principais ferramentas utilizadas na aplicação da técnica de *Bytecode Engineering*.

Capítulo 4 este capítulo descreve os principais conceitos de metaprogramação, reflexão e metadados. Aborda de forma prática o uso de anotações (metadados) na linguagem Java.

Capítulo 5 este capítulo fala de forma superficial sobre as tecnologias e ferramentas utilizadas na realização deste trabalho.

Capítulo 6 este capítulo descreve de forma clara e objetiva as soluções já existentes, comentando as vantagens e desvantagens de cada solução.

Capítulo 7 este capítulo descreve todas as características do projeto, desde a parte de análise até a implementação.

Capítulo 8 neste capítulo será desenvolvido um estudo de caso para comprovar as funcionalidades do framework.

Capítulo 9 este capítulo descreve os pontos fortes e fracos do trabalho, trabalhos futuros e outras conclusões sobre o projeto.

2 ASPECT ORIENTED PROGRAMMING

Neste capítulo será abordado de forma geral o paradigma de programação orientada a aspectos. Será falado um pouco da história deste paradigma, passando de forma objetiva pelos principais conceitos e benefícios, trazendo as informações necessárias para a compreensão superficial deste paradigma.

2.1 Definição

AOP é um paradigma de programação que foi construído tomando como base outros paradigmas (OOP e *procedural programming*) , cujo o principal objetivo seria a modularização de interesses transversais, utilizando um dos paradigmas base na implementação dos interesses centrais. A forma como AOP e o paradigma base se integram se dá com a utilização de aspectos que determinam a forma como os diferentes módulos se relacionam entre si na formação do sistema final (LADDAD, 2003).

2.2 História

Após um grande período de estudos, pesquisadores chegaram a conclusão que para desenvolver um software de qualidade era fundamental separar os interesses do sistema, ou seja, deveria então ser aplicado o princípio de *Separation of Concerns* (SoC)[1]. Em 1972, David Parnas escreveu um artigo, que tinha como proposta aplicar SoC através de um processo de modularização, onde cada módulo deveria esconder as suas decisões de outros módulos. Passado alguns anos, pesquisadores continuaram a estudar diversas formas de separação de interesses. OOP foi a melhor, se tratando de separação de interesses centrais, mas quando se tratava de interesses transversais, acabava deixando a

[1]Para saber mais sobre SoC consulte o Glossário ao final deste texto.

desejar. Diversas metodologias — *generative programming, meta-programming, reflective programming, compositional filtering, adaptive programming, subject-oriented programming, aspect oriented programming,* e *intentional programming* — surgiram como possíveis abordagens para modularização de interesses transversais. AOP acabou se tornando a mais popular entre elas (LADDAD, 2003).

Em 1997 Gregor Kiczales e sua equipe descreveram de forma sólida o conceito de Programação Orientada a Aspectos, durante um trabalho de pesquisa realizado pelo PARC, uma subsidiária da *Xerox Corporation*. O documento descreve uma solução complementar a OOP, ou seja, seriam utilizados aspectos para encapsular as preocupações transversais, de forma a garantir a reutilização por outros módulos de um sistema. Sugeriu também diversas implementações de AOP, servindo como base para a criação do AspectJ[2], uma linguagem AOP muito difundida nos dias de hoje (GROVES, 2013).

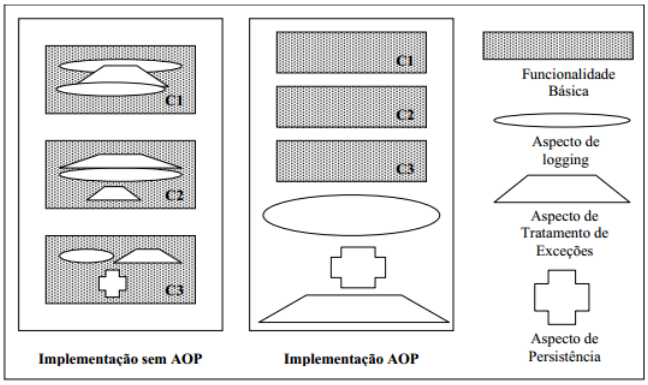

Figura 2.1: Implementação de um sistema com e sem a utilização de aspectos.
Fonte: (STEINMACHER, 2003).

2.3 Principais conceitos

Para entender melhor o funcionamento de AOP, é preciso compreender seus principais conceitos.

Os 6 principais conceitos de AOP são:

Aspecto (*aspect*) - pode ser definido como um interesse transversal, ou seja, interesse onde sua implementação é espalhada por diversos módulos ou componentes de um sistema.

[2]No final dos anos 90, a Xerox Corporation, transferiu o projeto AspectJ para a comunidade *Open Source* em eclipse.org.

Ponto de junção (*joinpoint*) - pode ser definido basicamente como pontos bem definidos na execução de um programa.

Ponto de corte (*pointcut*) - em AOP um *pointcut* pode ser representado como um agrupamento de pontos de junção.

Conselho (*advice*) - é a implementação de determinado interesse transversal, sendo executado quando determinado *pointcut* é ativado, podendo ser executado antes (*before*), em torno (*around*) ou depois (*after*) da execução de um ponto de junção.

Introdução (*introduction*) - é quando um aspecto introduz algumas mudanças em classes, interfaces , métodos. Por exemplo, pode-se acrescentar uma variável ou método a uma classe existente (LADDAD, 2003).

Combinador (*weaver*) - é a ferramenta responsável pela combinação dos interesses centrais com os interesses transversais do sistema, ou seja, é o momento de integração destes interesses na formação do sistema final.

Pode-se comparar também a complexidade de alguns conceitos de AOP com os principais conceitos de OOP:

Tabela 2.1: Comparação entre conceitos de AOP e OOP (JACOBSON; NG, 2004).

AOP	**OOP**
Aspecto	Classe
Conselho	Método
Ponto de junção	Atributo

Pointcut não se encaixa em termos de complexidade em nenhum conceito de orientação a objetos, mas podemos comparar um *pointcut* com um gatilho (*trigger*) da linguagem SQL (*Structured Query Language*) (JACOBSON; NG, 2004).

2.3.1 Tipos de combinação

Uma importante decisão de projeto que deve ser tomada, ao planejar uma ferramenta ou linguagem AOP, é o tipo de combinação que será utilizada na integração dos interesses. Existem dois tipos de combinação:

Estática consiste na modificação de bytecodes em tempo de compilação, para integração dos interesses centrais e transversais. Possui um desempenho maior em relação a

combinação dinâmica, por ser um processo que necessita de menos passos para ser concluído (STEINMACHER, 2003).

Dinâmica a combinação é feita em tempo de execução e oferece maior flexibilidade ao programador podendo alterar, modificar e remover aspectos. Porém o desempenho do sistema é afetado, podendo causar também erros durante a execução do programa (STEINMACHER, 2003).

2.4 Benefícios

De forma geral pode-se dizer que todo paradigma possui seus prós e contras, sendo assim, difícilmente será encontrada uma metodologia que resolva todos os problemas da melhor maneira. Com AOP não é diferente, mas deve-se considerar sua larga escala de benefícios:

- Separação de interesses e Alta modularização - com AOP pode-se separar o projeto em módulos distintos, possuindo assim um acoplamento mínimo que elimina o codigo duplicado, fazendo com que o sistema fique muito mais fácil de entender e manter (LADDAD, 2003).

- Fácil evolução - utilizando AOP o sistema fica muito mais flexível quando se trata da adição de novas funcionalidades, fazendo com que a resposta às exigências se tornem mais rápidas.

- Foco na prioridade - o arquiteto do projeto pode se concentar nos requisitos básicos atuais do sistema, os novos requisitos que abordam interesses transverssais podem ser tratados facilmente com a criação de novos aspectos. AOP trabalha em harmonia com metodos agéis, por exemplo apóia a prática YAGNI (*"You aren't gonna need it"*) [3], do *Extreme Programming* (XP)[4] (LADDAD, 2003).

- Maior reutilização de código - a chave para a reutização de código definitiva é uma implementação flexível, com AOP é possível pois cada aspecto é implementado como um módulo distinto, se tornando adaptavél a implementações equivalentes convencionais (LADDAD, 2003).

[3] Em português significa "Você não vai precisar dele". Para saber mais sobre YAGNI consulte o Glossário.

[4] É um dos métodos agéis mais utilizados nos dias atuais.

- Aumento na produtividade - o ciclo do projeto se torna mais rápido, a grande reutilização usada em AOP, faz com que o tempo de desenvolvimento seja reduzido, diminuindo também o tempo de implantação e o tempo de resposta às novas exigências de mercado (LADDAD, 2003).

- Redução de custos e aumento da qualidade - com a implementação dos interesses transversais em módulos distintos, o custo de implementação cai bastante, fazendo com que o desenvolvedor concentre-se mais nos interesses centrais, criando um produto de qualidade e com o custo reduzido.

2.5 Considerações finais

Neste capítulo foi definido os principais conceitos de AOP, que serão tomados como base para o desenvolvimento deste trabalho . Também foi possível perceber a importância em utilizar AOP como paradigma responsável pela modularização dos interesses transverssais.

3 BYTECODE ENGINEERING

Neste capítulo serão abordados alguns conceitos fundamentais de *bytecode*, também será definida a técnica de *Bytecode Engineering*, apresentando algumas vantagens em sua utilização e ferramentas que podem ser utilizadas para aplicar esta técnica.

3.1 Fundamentos de bytecode

Basicamente pode-se dizer que *bytecode* é a linguagem intermediária entre o código fonte e o código de máquina, que faz com que os programas Java possam ser executados em múltiplas plataformas. O documento responsável pela definição de *bytecode* é a Especificação da Máquina Virtual Java (*Java Virtual Machine Specification*)[1] que descreve também os conceitos da linguagem, formato dos arquivos de classes, os requisitos da *Java Virtual Machine* (JVM), entre outros (KALINOVSKY, 2004).

A JVM que executa sobre o sistema operacional, é responsável pelo ambiente de execução dos programas Java, sendo também a responsável pela conversão de instruções de *bytecode* Java em instruções nativas de máquina (STÄRK; SCHMID; BÖRGER, 2001).

3.2 Definição

Bytecode Engineering é uma técnica normalmente utilizada para criação, manipulação e modificação de classes Java compiladas à nível de *bytecode*. Muitas tecnologias utilizam esta técnica para otimizar, ou melhorar classes já existentes.

[1] Pode ser acessada em : http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

3.3 Vantagens

Dentre as principais vantagens em utilizar *Bytecode Engineering* estão:

- Consegue-se manipular o *bytecode* sem a necessidade de recompilação, ou obtenção do código fonte original.

- O *bytecode* pode ser gerado ou instrumentado por *class loader* em tempo real, enquanto as classes são carregadas em uma JVM (KALINOVSKY, 2004).

- É muito mais fácil e rápido automatizar a geração de *bytecode* por ser um processo mais baixo nível e também por não haver a necessidade de execução do compilador, ao contrário da geração de código fonte que possui alguns passos a mais no seu processo, e também necessita da execução do compilador (KALINOVSKY, 2004).

- Instrumentar métodos, tendo como objetivo a inserção de lógica adicional, que não necessita estar nos arquivos de código fonte. Por exemplo, pode-se deixar no arquivo fonte apenas os interesses centrais do sistema, e incluir os interesses transversais utilizando *Bytecode Engineering*.

3.4 Ferramentas

Pode-se dizer que a manipulação direta de *bytecode* é uma tarefa complexa, pois se trata de uma linguagem de baixo nível, e o aprendizado desta linguagem para programadores de alto nível, tende a ser muito lenta, fazendo com que seja inviável o seu aprendizado dentro do escopo de um projeto.

Pensando neste problema, foram criadas algumas bibliotecas que tornam o processo de manipulação de *bytecode* mais alto nível, facilitando o seu aprendizado. As bibliotecas mais utilizadas que aplicam a técnica de *Bytecode Engineering* são: *Javassist*, *Byte Code Engineering Library (BCEL)* e *ASM*.

3.4.1 Javassist

Javassist (Assistente de Programação Java) é uma biblioteca de classes utilizada na manipulação de *bytecode* em Java, permitindo que classes sejam criadas em tempo de execução e manipuladas no tempo de carregamento da JVM. A principal diferença do

Javassist para outros manipuladores de *bytecode*, é que Javassist possui API no nível de código fonte e no nível de *bytecode* (CHIBA, 2013).

3.4.2 BCEL

BCEL tem como objetivo oferecer aos usuários (desenvolvedores) uma maneira conveniente de analizar, criar, manipular arquivos de classe Java (binário). No BCEL as classes são representadas por objetos que contêm todas as suas informações (métodos, campos e instruções de *bytecode*) (BCEL, 2014).

3.4.3 ASM

ASM é uma estrutura de manipulação de *bytecode* Java. Ela pode ser usada para gerar classes dinamicamente, diretamente na forma binária, ou modificar dinamicamente as classes em tempo de carregamento, ou seja, antes de serem carregados pela JVM. ASM oferece funcionalidades semelhantes ao BCEL, mas é muito menor e mais rápido do que essa ferramenta (JAVA-SOURCE, 2012).

3.5 Considerações finais

Pode-se perceber que a manipulação de bytecodes não requer necessariamente conhecimentos específicos de bytecode, além do mais existem uma variedade de frameworks que fazem o trabalho "pesado", possibilitando a manipulação utilizando uma API de mais alto nível. Os conceitos abordados neste capítulo auxiliam o entendimento do trabalho em capítulos futuros.

4 METAPROGRAMAÇÃO

Neste capítulo serão introduzidos os principais conceitos de metaprogramação, falando também dos principais benefícios em aplicar esta metodologia em seus programas. Também é apresentado neste capítulo informações relevantes sobre o uso de anotações e reflexão na linguagem Java.

4.1 Definição

Metaprogramação por sua vez, pode ser definida como sendo um programa de computador que escreve e/ou manipula programas em tempo de execução ou compilação, fazendo com que o programa se adapte a diferentes circunstâncias, podendo controlar, monitorar ou invocar a si mesmo, visando alcançar as funcionalidades desejadas (HAZZARD; BOCK, 2013).

4.2 Benefícios

O uso de metaprogramação no desenvolvimento de software pode trazer uma série de benefícios tanto para a empresa, quanto para o desenvolvedor, entre alguns dos principais benefícios estão:

- Simplicidade - com metaprogramação tem-se a possibilidade de adaptar um único bloco de código a diversas situações, sem a necessidade de geração de código, fazendo com que as classes se tornem simples e ao mesmo tempo pequenas em comparação a outros programas que não usam esta abordagem.

- Adaptação em tempo real - pode-se fazer uma análise minuciosa do programa em tempo de execução, fazendo com que alterações no modelo de dados da aplica-

ção sejam percebidas em tempo real, ou seja, pode-se tornar o sistema altamente configurável sem a necessidade de compilação do projeto.

4.3 Principais conceitos

Para entender melhor a metaprogramação, é necessário compreender seus princípios (conceitos) básicos:

Metalinguagem linguagem usada para escrever um metaprograma.

Metaprograma representa um conjunto de componentes semelhantes que contém funcionalidades diferentes, que podem ser instanciados através de parametrização de modo a criar uma instância do componente específico (DAMAŠEVIČIUS; ŠTUIKYS, 2008).

Metadados são dados estruturados ou codificados que descrevem as características das entidades , portando informações que visam auxiliar na identificação, descoberta, avaliação e gestão das entidades descritas (ASSOCIATION et al., 1999).

Reflexão é a capacidade de um programa em observar e modificar, possivelmente, a sua estrutura e comportamento, fazendo com que a própria linguagem de programação faça o papel de metalinguagem (MALENFANT; JACQUES; DEMERS, 1996).

4.4 Anotações em Java

Metadados estão disponíveis na linguagem Java apartir da versão 5, representados através de anotações (*annotations*), sendo apresentada como uma das principais novidades no lançamento desta versão.

4.4.1 Definição

Uma anotação associa uma informação arbitrária ou um metadado com um elemento de um programa Java. Cada anotação tem um nome e zero ou mais membros. Cada membro tem um nome e um valor, e são estes pares *nome* = *valor* que carregam as informações da anotação (FLANAGAN, 2005).

Anotações são um tipo especial de interface, designado pelo caractere @ que é precedido da palavra-chave *interface*. Anotações são aplicadas aos elementos do programa,

ou também em outras anotações, com o intuito de fornecer informações adicionais (ARNOLD; GOSLING; HOLMES, 2000).

4.4.2 Conceitos

As anotações em Java possuem alguns conceitos importantes que devem ser compreendidos:

Tipo de anotação (*annotation type*) - O nome de uma anotação, bem como os nomes, tipos e valores padrão de seus membros são definidos pelo tipo de anotação. Um tipo de anotação é essencialmente uma interface Java com algumas restrições sobre seus membros (FLANAGAN, 2005).

Membro de anotação (*annotation member*) - Os membros de uma anotação são declarados em um tipo de anotação como métodos sem argumentos. O nome do método e o tipo de retorno define o nome e o tipo do membro (FLANAGAN, 2005).

Anotação marcadora (*marker annotation*) -É um tipo de anotação que não define membros, ou seja, uma anotação desse tipo traz informações simplesmente pela sua presença ou ausência (FLANAGAN, 2005).

Meta-anotação (*meta-annotation*) - são anotações utilizadas para descrever o comportamento de um tipo de anotação (HORSTMANN; CORNELL, 2004).

Alvo (*target*) - elemento do programa que será anotado.

Retenção (*retention*) - é a especificação do tempo em que a informação contida na anotação é mantida. A especificação deste tempo é feita através de uma meta-anotação (FLANAGAN, 2005).

4.4.3 Utilidades

Utilizando anotações em Java pode-se adicionar diversas funcionalidades e recursos ao *software*, sem que haja depêndencias entre as classes.

Dentre as principais utilidades em usar anotações estão:

- Fornecer informações ao compilador - por exemplo, pode-se utilizar este recurso para detectar erros especifícos em determinada parte do programa em tempo de

compilação, também pode-se informar ao compilador que ignore determinados tipos de avisos, etc.

- Processamento em tempo de compilação - normalmente utiliza-se este recurso através de ferramentas capazes de processar anotações, com o intuito de gerar código, arquivos, etc.

- Processamento em tempo de execução - pode-se utilizar para marcar elementos, com o objetivo de ler sua estrutura (tipo, modificadores de acesso,etc) através de reflexão em tempo de execução, fazendo com que o programa se adapte a determinadas situações.

4.4.4 Exemplo

Na figura 4.1 é mostrado como criar uma anotação básica, que terá como alvo uma classe, e estará visível em tempo de execução, e também irá possuir um membro cujo tipo é uma *String*.

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

@Target(ElementType.TYPE) //alvo da anotação vai ser uma classe
@Retention(RetentionPolicy.RUNTIME) // a anotação estará visível em tempo de execução
public @interface Configuracao { //tipo da anotação

    String ipServidor(); //membro da anotação
}
```

Figura 4.1: Criando uma anotação.
Fonte: Autoria própria.

Na figura 4.2 é mostrado como utilizar a anotação criada na figura 4.1.

```
@Configuracao(ipServidor="10.1.1.50")
public class Terminal {

    //código da classe
}
```

Figura 4.2: Utilizando uma anotação.
Fonte: Autoria própria.

4.5 Reflexão em Java

Na linguagem Java pode-se utilizar reflexão por meio do pacote *java.lang.reflect*, juntamente com o pacote *java.lang* que contém as classes *Class* e *Package*, e *java.lang.annotation*

que faz referência à classe *Annotation*. Este conjunto de pacotes oferecem uma variedade de classes responsáveis pela leitura e modificação da estrutura de um programa em tempo de execução.

4.5.1 Aplicações

Normalmente utiliza-se reflexão para tomar alguma decisão com base na estrutura do programa, ou apenas mostrar informações sobre ela, ou seja, utiliza-se o(s) pacote(s) de reflexão do Java para descobrir diversos tipos de informações relacionadas com a estrutura de classes, métodos, atributos e etc. Também pode-se utilizar reflexão para instanciar objetos de um tipo determinado, invocar método(s) de uma classe específica, ler anotações, entre outros.

4.5.2 Funcionamento

A reflexão em um programa Java tem inicio apartir de um objeto do tipo *Class*. Com um objeto do tipo *Class* pode-se obter sua lista completa de membros (métodos, atributos) e informações sobre eles, podendo descobrir também todos os tipos desta classe (interfaces que implementa, classes que estende), e descobrir informações sobre a própria classe, como os modificadores aplicados a ela (*public*, *protected*, *private*, etc) (ARNOLD; GOSLING; HOLMES, 2000).

4.5.3 Lendo anotações

```
public class Configurador {

    public static void main(String[] args) {

        Terminal terminal = new Terminal();

        //pega a classe
        Class classe = terminal.getClass();

        // verifica se a anotação Configuracao esta presente no terminal
        if(classe.isAnnotationPresent(Configuracao.class))
        {
            //pega a anotação Configuracao em terminal
            Configuracao conf = (Configuracao)classe.getAnnotation(Configuracao.class);
            //mostra o valor do membro de anotação ipServidor
            System.out.println(conf.ipServidor());
        }
    }
}
```

Figura 4.3: Lendo uma anotação.
Fonte: Autoria própria.

Os pacotes utilizados para reflexão possuem uma API[1] de fácil entendimento, e que oferecem suporte à leitura de anotações.

Na figura 4.3 é mostrado como ler anotações presentes em classes, e também como ler informações presentes nos membros de uma anotação.

4.6 Considerações finais

Este capítulo apresentou os principais conceitos de Metaprogramação, Reflexão e Metadados. Também foi abordado neste capítulo a utilização superficial da API de reflexão e metadados da linguagem Java.

[1] Visite: "http://docs.oracle.com/javase/7/docs/api/"para saber mais sobre a api.

5 FERRAMENTAS UTILIZADAS

6 SOLUÇÕES EXISTENTES

6.1 PostSharp

6.2 AspectJ

7 O FRAMEWORK

7.1 Análise e projeto

7.2 Implementação

8 ESTUDO DE CASO - INSTRUMENTAÇÃO

9 CONCLUSÃO

Conclusões ...

REFERÊNCIAS

ARNOLD, K.; GOSLING, J.; HOLMES, D. **The Java programming language**. [S.l.]: Addison-wesley Reading, 2000. v.2.

ASSOCIATION, A. L. et al. **Task Force on Metadata Summary Report**. [S.l.]: June, 1999.

BCEL, A. C. **Apache Commons BCEL™**. Disponível em: <http://commons.apache.org/proper/commons-bcel/>. Acesso em: 20 junho. 2014.

BECK, K.; ANDRES, C. **Extreme programming explained**: embrace change. [S.l.]: Addison-Wesley Professional, 2004.

CHIBA, S. **Javassist**. Disponível em: <http://www.csg.ci.i.u-tokyo.ac.jp/ chiba/javassist/>. Acesso em: 16 junho. 2014.

DAMAŠEVIČIUS, R.; ŠTUIKYS, V. Taxonomy of the fundamental concepts of metaprogramming. **Information Technology and Control**, [S.l.], v.37, n.2, p.124–132, 2008.

FLANAGAN, D. **Java in a Nutshell**. [S.l.]: O'Reilly Media, 2005.

GROVES, M. D. **AOP in. NET**. [S.l.]: Manning Publ., 2013.

HAZZARD, K.; BOCK, J. **Metaprogramming in. NET**. [S.l.]: Manning Pub., 2013.

HORSTMANN, C. S.; CORNELL, G. **Core Java 2, Advanced Features, Vol. 2**. [S.l.]: Prentice Hall, 2004.

JACOBSON, I.; NG, P.-W. **Aspect-oriented software development with use cases**. [S.l.]: Addison-Wesley Professional, 2004.

JAVA-SOURCE. **Open Source ByteCode Libraries in Java**. Disponível em: <http://java-source.net/open-source/bytecode-libraries>. Acesso em: 20 junho. 2014.

KALINOVSKY, A. **Covert Java**: techniques for decompiling, patching, and reverse engineering. [S.l.]: Pearson Higher Education, 2004.

LADDAD, R. **AspectJ in action: practical aspect-oriented programming**. [S.l.]: Manning, 2003.

MALENFANT, J.; JACQUES, M.; DEMERS, F.-N. A tutorial on behavioral reflection and its implementation. In: REFLECTION, 1996. **Proceedings...** [S.l.: s.n.], 1996. v.96, p.1–20.

PRESSMAN, R. **Software engineering: a practitioner's approach**. [S.l.]: McGraw Hill, 2010.

STÄRK, R. F.; SCHMID, J.; BÖRGER, E. **Java and the Java virtual machine**. [S.l.]: Springer Heidelberg, 2001.

STEINMACHER, I. F. Estudo de Princípios para Modelagem Orientada a Aspectos. **Trabalho de Graduação. Universidade Estadual de Maringá, Maringá, PR**, [S.l.], 2003.

GLOSSÁRIO

SoC è um princípio de projeto, criado com a finalidade de subdividir o problema em conjuntos de interesses tornando a resolução do problema mais fácil.Cada interesse fornece uma funcionalidade distinta, podendo ser validado independentemente das regras negócio (PRESSMAN, 2010).

YAGNI é um princípio de projeto, bastante usado em equipes XP , cuja principal finalidade é implementar apenas o necessário.

Extreme Programming XP é um estilo de desenvolvimento de software com foco em excelentes técnicas de programação, comunicação clara e trabalho em equipe, que permite grande produtividade no desenvolvimento (BECK; ANDRES, 2004).

Decorator é um padrão de projeto estrutural, cujo principal objetivo é adicionar funcionalidades a um objeto dinamicamente.

Proxy é um padrão de projeto estrutural, cujo principal objetivo é controlar as chamadas a um objeto através de outro objeto de mesma interface.