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

NICOLAS DIAS 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. Esp. Márcio Torres
Orientador

Rio Grande, fevereiro de 2014

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Bento, Nicolas Dias

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

67 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. Esp. Márcio Josué Ramos Torres
Orientador

Prof. Dr. Eduardo Wenzel Brião
IFRS - Campus Rio Grande

Prof. Msc. Igor Ávila Pereira
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 está 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 complementares 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, que oferece diversos recursos e funcionalidades que auxiliam na projeção e execução de

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

projetos utilizando OOP e AOP de forma conjunta.

Visando complementar e aumentar os estudos sobre AOP, decidiu-se desenvolver um *framework*[2], cujo objetivo é desenvolver uma funcionalidade genérica capaz de encapsular os interesses transverssais de um sistema, com o intuíto de aplicar os conceitos de AOP no desenvolvimento de softwares de forma simples e descomplicada, de forma que todo o código-fonte da ferramenta seja aberto para a comunidade de desenvolvedores e pesquisadores, podendo servir como uma base mais concreta para o estudo, desenvolvimento e aprimoramento de outros trabalhos.

1.1 Objetivo

Este trabalho tem como objetivo geral desenvolver um *framework* AOP de código-fonte aberto visando encapsular os interesses transversais de um sistema Java.

Os objetivos específicos deste trabalho são:

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

1.2 Motivação

Existe uma variedade de ferramentas que implementam os conceitos de AOP, porém estas ferramentas não servem de base para outros estudos, por não oferecerem informações concretas, ou robustas sobre o real funcionamento interno das mesmas, fazendo com que pesquisadores e estudiosos que desejam ingressar e aprofundar-se dentro deste paradigma tenham uma grande dificuldade ao encontrar um ponto de partida. Por este motivo este trabalho propõe não só demonstrar o uso e o desenvolvimento de um *framework* AOP, mas também oferecer uma base concreta e prática, buscando incentivar e facilitar a aprendizagem e o entendimento do paradigma.

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

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

Capítulo 3 BYTECODE ENGINEERING: 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 META PROGRAMAÇÃO: este capítulo descreve os principais conceitos de meta programação, reflexão e metadados. Aborda de forma prática o uso de anotações (metadados) na linguagem Java.

Capítulo 5 TECNOLOGIAS E FERRAMENTAS UTILIZADAS: este capítulo fala de forma superficial sobre as tecnologias e ferramentas utilizadas na realização deste trabalho.

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

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

Capítulo 8 ESTUDO DE CASO - INSTRUMENTAÇÃO: neste capítulo será desenvolvido um estudo de caso para comprovar as funcionalidades do framework.

Capítulo 9 CONCLUSÃO: este capítulo descreve as etapas do projeto, resultados obtidos, trabalhos futuros e outras conclusões sobre o trabalho realizado.

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).

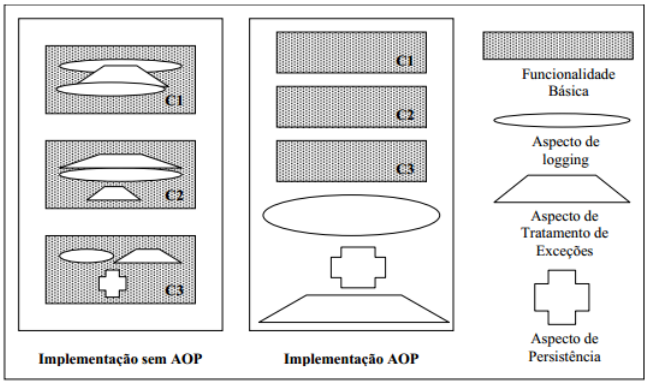

Figura 2.1: Implementação de um sistema com e sem a utilização de aspectos.
Fonte: (STEINMACHER, 2003).

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 ou de carregamento de classes, 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, dificilmente 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 código 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 concentrar nos requisitos básicos atuais do sistema, os novos requisitos que abordam interesses transversais podem ser tratados facilmente com a criação de novos aspectos. AOP trabalha em harmonia com métodos ágeis, por exemplo apoia 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 reutilização de código definitiva é uma implementação flexível, com AOP é possível pois cada aspecto é implementado como um módulo distinto, se tornando adaptável 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 ágeis 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 transversais.

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 analisar, 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 META PROGRAMAÇÃO

Neste capítulo serão introduzidos os principais conceitos de meta programação, 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

Meta programação 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 meta programação 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 meta programação 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 meta programação, é necessário compreender seus princípios (conceitos) básicos:

Metalinguagem linguagem usada para escrever um meta programa.

Meta programa 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 a partir 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 dependências 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 específicos 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 a partir 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 Meta programação, 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 TECNOLOGIAS E FERRAMENTAS UTILIZADAS

Este capítulo apresenta de forma clara e objetiva as tecnologias e ferramentas que serão utilizadas para o desenvolvimento deste trabalho, apresentando os motivos que fizeram com que estas ferramentas e tecnologias fossem as escolhidas para este estudo.

5.1 Eclipse

O projeto Eclipse foi inicialmente criado pela IBM no ano de 2001. No ano de 2004 foi criada a *Eclipse Foundation*, uma corporação sem fins lucrativos, visando desenvolver tecnologias de código-fonte aberto com o intuito de incentivar os desenvolvedores de software a usar a tecnologia Eclipse para a construção de seus produtos e serviços de software comercial (MILINKOVICH, 2008).

O Eclipse foi escolhido como Ambiente de Desenvolvimento Integrado (*Integrated Development Environment - IDE*) para a realização deste trabalho, por ser um dos mais utilizados no desenvolvimento de *software* utilizando a linguagem Java, e também por oferecer diversas ferramentas que facilitam e agilizam o desenvolvimento do(s) projeto(s). Outra vantagem do Eclipse é que todas as ferramentas são oferecidas gratuitamente para uso comercial e não-comercial.

5.2 Java 7

Java é uma linguagem de programação e plataforma computacional lançada pela primeira vez pela *Sun Microsystems* em 1995. O paradigma de programação que mais caracteriza a linguagem Java é OOP, e uma das características que justifica a enorme compatibilidade com diversos aparelhos é a portabilidade da plataforma, que roda independentemente do sistema operacional (STÄRK; SCHMID; BÖRGER, 2001).

A proposta do trabalho justifica a escolha de Java como linguagem de programação, pois os principais objetivos envolvem um *framework* AOP para linguagem Java e a utilização de técnicas de *bytecode enginnering*. Será utilizado a versão 7, por ser uma versão que dispõe de diversos recursos e estabilidade alta.

5.3 JavaAgent

Agentes Java são componentes de *software* que oferecem recursos de instrumentação de aplicações, recurso este que foi introduzido a partir da versão 5 do Java. No contexto dos agentes, instrumentação fornece a capacidade de re-definir o conteúdo da classe que é carregada em tempo de execução (SRINIVASAN, 2011).

Este recurso da linguagem java será utilizado neste trabalho para interceptação das classes, com a finalidade de desenvolver o combinador de aspectos (*weaver aspects*), responsável pela formação do sistema final.

5.4 Javassist

Javassist é uma biblioteca de *Bytecode Enginnering* desenvolvida em cima da linguagem Java e que fornece uma API de mais alto nível em relação a outras bibliotecas do mesmo gênero (CHIBA, 2013).

A maioria das bibliotecas requerem um mínimo de conhecimentos de *bytecode*, Javassist trabalha de forma diferente, o usuário passa para a biblioteca cadeias de caracteres no formato de código-fonte, e a própria biblioteca se encarrega de transformar esta cadeia de caracteres de código de alto nível para a linguagem de *bytecode*.

A escolha da biblioteca Javassist para auxiliar na aplicação das técnicas de *Bytecode enginnering* se deu devido a sua facilidade no aprendizado , além de não requerer como pré-requisito conhecimentos de *bytecode*.

5.5 Considerações finais

Este capítulo apresentou de forma objetiva as principais ferramentas e tecnologias que serão utilizadas no desenvolvimento deste trabalho. Foi justificado de forma clara os motivos que contribuíram para a escolha destas ferramentas e tecnologias com o intuito de resolver o problema apresentado.

6 SOLUÇÕES EXISTENTES

Neste capítulo será abordado as características dos principais projetos relacionados existentes, e um exemplo prático de cada um deles.

6.1 PostSharp

Postsharp é um *framework* para a plataforma .NET da Microsoft, responsável por encapsular os interesses transversais de sistemas.

O Postsharp possui internamente uma biblioteca de aspectos já implementados para que o desenvolvedor possa incorporar aos seus projetos (GROVES, 2013).

O principal foco do Postsharp é a automação padrão dos produtos (por isso a biblioteca interna de aspectos), mas mesmo assim possui uma mega estrutura que dá suporte a criação de diversos outros tipos de aspectos (GROVES, 2013).

A figura 6.1 mostra a implementação de um aspecto atráves do *framework* Postsharp.

```
[Serializable]
public sealed class FirstTraceAttribute : OnMethodBoundaryAspect
{
    public override void OnEntry(
        MethodExecutionArgs args )
    {
        System.Diagnostics.Trace.WriteLine( string.Format(
            "Entrando {0}.{1}.",
            args.Method.DeclaringType.FullName,
            args.Method.Name ));
        System.Diagnostics.Trace.Indent();
    }

    public override void OnExit(
        MethodExecutionArgs args )
    {
        System.Diagnostics.Trace.Unindent();
        System.Diagnostics.Trace.WriteLine(
            string.Format(
            "Saindo {0}.{1}.",
            args.Method.DeclaringType.FullName,
            args.Method.Name) );
    }
}
```

Figura 6.1: Exemplo de aspecto no Postsharp.

Para criação de um aspecto através do Postsharp deve-se implementar a interface `OnMethodBoundaryAspect`, que obrigará a implementar os métodos `OnEntry` e `OnSucess`, que representam os *advices before* e *after*, respectivamente.

6.2 AspectJ

O AspectJ é uma linguagem de extensão da linguagem Java, pois possui suas próprias características e palavras chaves reservadas provenientes de uma linguagem de programação. Se o AspectJ é uma extensão de Java, então todo programa que seja válido em Java, também será validado com o AspectJ (LADDAD, 2003).

Diferentemente do Postsharp o AspectJ é considerado uma linguagem de programação, pois possui sua própria sintaxe e também possui um compilador próprio que gera arquivos de *bytecodes* compatíveis com a JVM (LADDAD, 2003).

A figura 6.2 mostra a criação de um aspecto utilizando a sintaxe padrão da linguagem AspectJ.

```
public aspect TratamentoErroAspect {

    pointcut tratamento(): call (Boolean *DAO.*());

     Boolean around():tratamento()
    {
        try
        {
            return  proceed();

        }catch(Error e)
        {
            System.out.println("Erro: "+e.getMessage());
        }
        return false;
    }
    before():tratamento()
    {
        System.out.println("Antes");
    }

    after():tratamento()
    {
        System.out.println("Depois");
    }
}
```

Figura 6.2: Exemplo de aspecto com a sintaxe padrão do AspectJ.

A linguagem AspectJ possui palavras reservadas como `before, after, around, aspect, pointcut,call` que são interpretas pelo compilador do AspectJ durante a combinação entre classes e aspectos.

A figura 6.3 mostra a criação de um aspecto utilizando o modelo de anotações do AspectJ.

```
@Aspect
public class LoggingAspect {

    @Before
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("Antes do método!");
    }

    @After
    public void logAfter(JoinPoint joinPoint) {
        System.out.println("Depois do método!");
    }

   @Around
   public void logAround(ProceedingJoinPoint joinPoint)
        throws Throwable {

        System.out.println("Inicio do around!");
        joinPoint.proceed(); //executa o método
        System.out.println("Fim do around!");
   }

}
```

Figura 6.3: Exemplo de aspecto na forma de anotações no AspectJ.

O AspectJ faz uso de uma sintaxe própria, mas também dá suporte ao uso de anotações para criação de aspectos (como mostrado na figura 6.3). A anotação responsável por informar que se trata de um aspecto é a `@Aspect`, e para informar que os métodos são *advices* utilizasse das anotações `@Before`, `@After` e `@Around`, cuja nomenclatura representa o tipo de *advice*.

6.3 Considerações finais

Este capítulo abordou as características básicas do *framework* Postsharp e da linguagem AspectJ. Foi demonstrado também um exemplo prático de criação de aspectos em cada um dos projetos relacionados.

7 O FRAMEWORK

Este capítulo aborda em detalhes o *framework* desenvolvido, desde a parte de análise e projeto até a parte de implementação, explicando de forma clara os caminhos percorridos durante a evolução do trabalho.

7.1 Introdução

Para a realização da análise e desenvolvimento do *framework* fez-se necessário o estudo de diversos conceitos e tecnologias, cujo embasamento teórico se encontra nos capítulos anteriores. O trabalho apresentado se concentra em torno dos seguintes conceitos: Programação Orientada à Aspectos, *Bytecode Enginnering*, Meta programação (metadados e reflexão).

Para o desenvolvimento do *framework* foi utilizado o Kit de Desenvolvimento Java versão 7 (*Java Development Kit* - JDK), cuja versão é estável, proporcionando uma alta utilização e aceitação no mercado, fez-se uso também do Eclipse como ambiente de desenvolvimento de *software*, pois possui diversas ferramentas integradas que facilitam e agilizam a programação. Foi utilizado também a biblioteca Javassist, responsável por abstrair a forma como é criado, manipulado e modificado *bytecode* de classes Java em tempo de carregamento.

Este projeto tem por intenção estudar os principais conceitos de AOP, criando uma ferramenta para abstrair estes conceitos, de forma que facilite a aprendizagem, a aplicação prática e também sirva de base para outros estudos e pesquisas. A ferramenta projetada e desenvolvida será em forma de *framework*, que terá como metodologia de uso a utilização de anotações, com o intuito de facilitar a utilização por parte do desenvolvedor e também visando um código mais limpo, concentrando as informações principais do módulo em

forma de código, e as informações adicionais (conceitos de AOP) sobre as classes, métodos e atributos serão informadas em forma de metadados (anotações).

7.2 Análise e projeto

Nesta seção será apresentada a parte de análise e projeto do *framework*, visando o entendimento da estrutura, arquitetura e funcionamento da ferramenta.

7.2.1 Especificações do projeto

Para cumprimento dos objetivos propostos pelo projeto, foi definido algumas especificações que devem ser cumpridas para que o projeto seja considerado apto. Entre as principais especificações estão:

- A ferramenta deve ser capaz de encapsular os conceitos básicos de AOP (*joinpoint, advice, aspect* e *weaver*), visando uma funcionalidade genérica, que tem por objetivo separar os interesses centrais dos interesses transversais, concentrando cada interesse em seu devido módulo.

- A ferramenta terá a capacidade de interceptação de métodos, ou seja, executar algum bloco de código antes, depois ou em volta da chamada ao método interceptado, podendo também durante esta interceptação modificar o retorno do método (caso ele retorne alguma informação). Com base nessas informações pode-se definir que os *joinpoints* (pontos de junção) serão as chamadas dos métodos interceptados.

- Os *advices* (antes, depois ou em volta) serão implementados por meio de uma `interface`, que quando implementada deverá ser tratada pelo *weaver* (combinador) como sendo um aspecto. Para informar ao método o tipo de aspecto e *advice* que deverá ser executado na sua interceptação deve ser utilizado *annotations*.

- Para implementação do combinador deve ser utilizado técnicas de *Bytecode Enginnering* para realizar modificações em classes já compiladas durante o carregamento destas classes para a Máquina Virtual do Java (JVM). O combinador será utilizado para integração dos interesses centrais com os interesses transversais durante o carregamento do sistema.

7.2.2 Fluxograma

A figura 7.1 mostra todos os processos realizados pela JVM quando é chamado um *Agent* junto à chamada de um sistema.

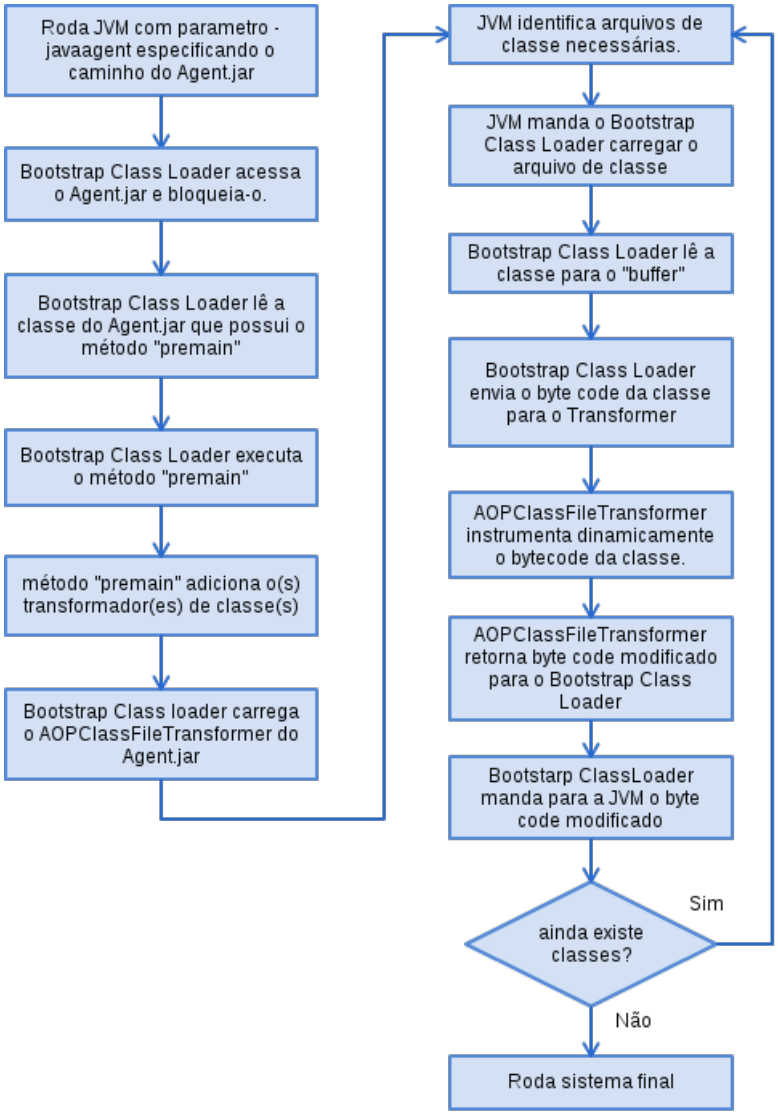

Figura 7.1: Fluxograma de processos da JVM na execução de um *Agent* (STÄRK; SCHMID; BÖRGER, 2001).

7.2.3 Diagrama de classes

Na figura 7.2 é apresentado o diagrama de classes de implementação na sua forma "reduzida"utilizado como base para a implementação do *framework*. Este diagrama serve para visualização do modelo de classes do sistema e seus relacionamentos, e foi construído com a ajuda da ferramenta *Astah Community 6.9.0*. O diagrama de classes de implementação completo está no Apêndice A.

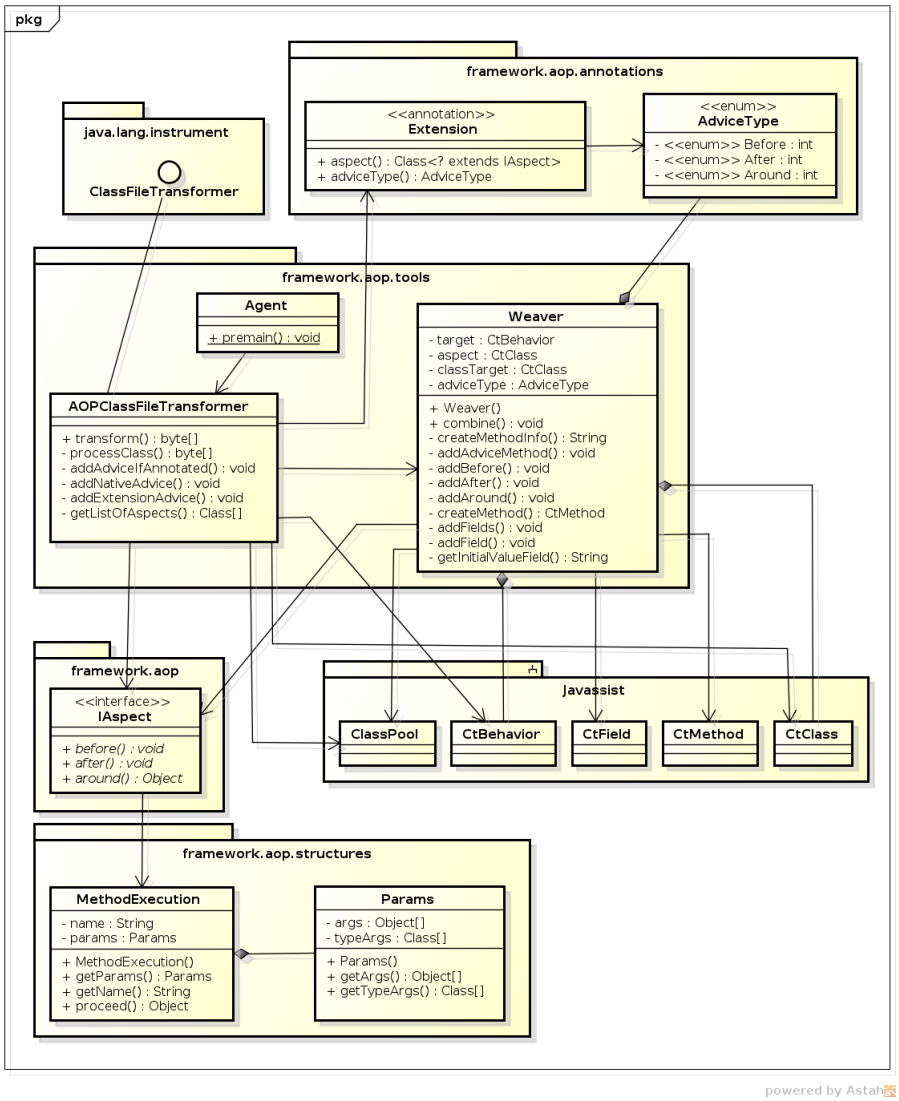

Figura 7.2: Diagrama de classes de implementação.

Abaixo está a descrição de cada uma das classes apresentadas no diagrama da figura 7.2:

Agent Esta é a classe que possui o `premain`, sendo assim é a classe principal do agente que será utilizado para interceptação das classes durante o seu carregamento para a JVM.

ClassFileTransformer Esta é uma `interface` que deve ser implementada por classes que desejam ser responsáveis pela modificação de outras classes antes do carregamento para a máquina virtual.

AOPClassFileTransformer Esta é a classe responsável por realizar e retornar para o *Class Loader* (carregador de classes do Java) as classes já modificadas. Esta classe

implementa a *interface* ClassFileTransformer. É ela que faz a análise de quais classes vão ser combinadas, e quais as regras de combinação que devem ser consideradas para cada método.

Weaver Esta classe representa o combinador de aspectos, ou seja, é nesta classe que são feitas as combinações de módulos OOP com módulos AOP. As informações necessárias para combinação são recebidas de `AOPClassFileTransformer`.

AdviceType É um enumerado que representa o tipo de *advice*, pode ser *Before* (antes), *After* (depois) ou *Around* (em volta). Utilizado para representar o tipo de *advice* que está sendo combinado.

IAspect Esta é a `interface` que deve ser implementada pelo desenvolvedor (usuário do *framework*) para informar que a classe será interpretada pelo modificador como sendo um aspecto.

MethodExecution Representa o método que está sendo interceptado. Possui informações sobre os parâmetros do método, e também um método auxiliar, cuja função é continuar a execução do método interceptado.

Params Esta classe é responsável por encapsular (guardar) os tipos e valores dos parâmetros do método interceptado.

ClassPool (Javassist) Esta classe simula o *Class Loader* do Java, esta classe é bastante utilizada para montar em forma de objeto as classes interceptadas.

CtClass (Javassist) Esta classe modela a estrutura de uma classe da linguagem Java, uma instância desta classe possui informações relevantes sobre ela e também métodos que navegam pela sua estrutura (métodos, atributos, construtores, etc).

CtBehavior (Javassist) Esta classe modela de forma genérica métodos e construtores. Ela oferece métodos que são bastante utilizados para adição de código para dentro do escopo de um construtor ou método.

CtMethod (Javassist) Esta classe é parecida com *CtBehavior*, porém modela apenas métodos, ou seja, possui algumas informações mais específicas sobre métodos.

CtField (Javassist) Esta classe modela um campo ou atributo de uma classe, possuindo uma gama de informações sobre eles e diversos métodos para possíveis modificações.

7.2.4 Diagrama de sequência

O diagrama de sequência foi utilizado para demonstrar a forma como se dá a chamada dos métodos durante o processo inicialização do *Agent* até a parte de modificação (combinação de classes e aspectos) de *bytecode* das classes interceptadas. Este diagrama pode ser visto na figura 7.3.

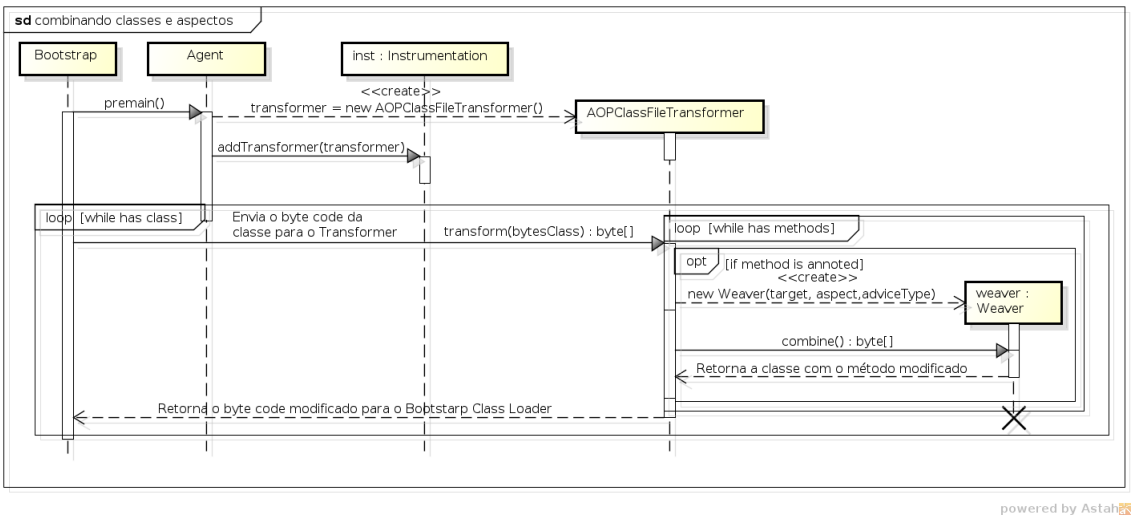

Figura 7.3: Diagrama de sequência.

O *Bootstrap Class Loader* é o carregador de classes inicial, ou seja, é ele que carrega as classes e bibliotecas padrões do Java para JVM (KALINOVSKY, 2004). O *Bootstrap Class Loader* é responsável por invocar (chamar) o método `premain` que por sua vez irá criar e adicionar o transformador de classes, neste caso `AOPClassFileTransformer`, que irá realizar a instrumentação do *bytecode* das classes antes de seu carregamento para JVM.

Após criado e adicionado o transformador pelo `premain` começa o processo de instrumentação, o *Bootstrap* pega o *bytecode* da classe interceptada e envia através do método `transform` para o transformador realizar possíveis modificações no *bytecode*.

O método `transform` por sua vez verifica todos os métodos da classe, procurando por métodos que utilizam algum tipo de aspecto de forma anotada, ou seja, métodos marcados com anotações específicas do *framework*. Se alguma anotação correspondente

for encontrada, então é recolhida as informações contidas nela (classe do aspecto, tipo de *advice*) e enviadas para o combinador junto ao método anotado, que a partir deste momento torna-se um método alvo de interceptação.

O combinador (`Weaver`) recebe as informações necessárias para realizar a combinação da classe interceptada e do aspecto informado na anotação. Após realizada a combinação, em todos os métodos anotados da classe, é retornado então para o *Bootstrap* o *bytecode* da classe já modificado. O processo citado acima se dá em todas as classes que estão sendo carregadas para JVM, basicamente pode-se dizer que estas classes interceptadas são as classes do sistema que está utilizando o *framework*. Maiores detalhes sobre a implementação do *framework* e o seu funcionamento são encontradas na seção 7.3.

7.3 Implementação

Nesta seção será detalhada a abordagem utilizada para a implementação deste *framework*. Também será discutida as características e o funcionamento geral do *framework*.

7.3.1 Primeiros passos

Muitas pesquisas foram realizadas para encontrar as tecnologias capazes de suprir as necessidades do projeto, inicialmente sabia-se que deveria-se trabalhar em cima de algum *framework* para aplicar as técnicas de *Bytecode Enginnering* e também que deveria ser aplicada alguma técnica de interceptação ou em tempo de carregamento das classes, ou em tempo de compilação, para possíveis combinações.

Inicialmente foi realizado diversos testes utilizando a técnica de Processadores de Anotações (*Annotations Processors*), que é um recurso utilizado para ler as anotações que estão visíveis em tempo de compilação. A ideia central para a utilização de Processadores de Anotações era de ler as anotações das classes em tempo de compilação, verificar se havia regras de combinação nestas anotações e posteriormente realizar a combinação.

Esta abordagem não foi bem-sucedida pois em tempo de compilação não haveria a possibilidade de modificação das classes, pois o *framework* de *Bytecode Enginnering* utilizado (Javassist) só dá suporte á modificação em tempo de carregamento. Cogitou-se então 2 possíveis saídas: mudar o *framework* de *Bytecode Enginnering* ou mudar a forma de interceptação das classes.

Decidiu-se então mudar a técnica de interceptação de classes, pois a mudança de *fra-*

mework iria atrasar o andamento do projeto, pois outros *frameworks* disponíveis exigem conhecimentos específicos de *bytecode*, fazendo com que se tornasse inapropriado para o escopo do projeto. Foi neste momento que decidiu-se utilizar Java Agentes como abordagem de interceptação de classes.

7.3.2 Premain

O `premain` é o método responsável pela inicialização do *agent*. É dentro deste método que são adicionados o(s) transformador(es) de classes que irão realizar a interceptação. Como mostrado na figura 7.4 o método `premain` do nosso *agent* AOP adiciona o transformador `AOPClassFileTransformer` ao instrumentador.

```
18 class Agent {
19
20     public static void premain(String agentArgs, Instrumentation inst) {
21
22         inst.addTransformer(new AOPClassFileTransformer());
23     }
24 }
```

Figura 7.4: Implementação do método `premain`.

7.3.2.1 Manifesto

Existe uma configuração que deve ser feita para que o *agent* saiba qual classe possui o método `premain`. Esta configuração deve ser feita em um arquivo de manifesto cujo nome deve ser *Manifest.mf*, localizado dentro da pasta do projeto. A figura 7.5 mostra o conteúdo do arquivo.

```
1 Manifest-Version: 1.0
2 Premain-Class: framework.aop.tools.Agent
3 Boot-Class-Path: ../lib/javassist.jar
```

Figura 7.5: Conteúdo do arquivo de manifesto.

A `linha 2` informa qual a classe possui o *premain*, e a `linha 3` adiciona o Javassist ao caminho de classes do *agent*. Este manifesto vai ser utilizado no momento da construção do arquivo *.jar*.

7.3.3 Transform

Quando implementa-se a `interface ClassFileTransformer`, o desenvolvedor é obrigado a implementar o método `transform`, pois a ideia central do transforma-

dor de classes é receber e modificar o *bytecode* de uma classe, sendo que o recebimento destes *bytes* é feito pelo método `transform`, então obrigatoriamente ele deve estar implementado.

Na figura 7.6 é mostrado a forma de implementação do método `transform` da classe `AOPClassFileTransformer`.

```
27    public byte[] transform(ClassLoader loader, String className,
28            Class<?> classBeingRedefined, ProtectionDomain protectionDomain,
29            byte[] classfileBuffer ) {
30
31        String[] ignoredPackages = new String[] { "sun/", "java/", "javax/" };
32
33        for (int i = 0; i < ignoredPackages.length; i++) {
34            if (className.startsWith(ignoredPackages[i])) {
35                return classfileBuffer;
36            }
37        }
38        return processClass(className, classfileBuffer);
39    }
```

Figura 7.6: Implementação do método `transform`.

O `transform` armazena na variável `ignoredPackages` os pacotes que devem ser ignorados, ou seja, as classes que pertencem aos pacotes listados (classes padrão do Java) vão ser ignoradas pelo transformador, retornando para o *Bootstrap Class Loader* o seu *bytecode* original. Se a classe interceptada não for uma classe padrão do Java, então ela será processada pelo método `processClass`, cuja implementação está na figura 7.7.

```
41    private byte[] processClass(String clazzName, byte[] bytes) {
42        ClassPool pool = ClassPool.getDefault();
43        CtClass cl = null;
44        byte[] bts={};
45        try {
46            cl = pool.makeClass(new java.io.ByteArrayInputStream(bytes));
47
48            for(CtBehavior behavior : cl.getDeclaredBehaviors())
49                for(Object a : behavior.getAnnotations())
50                    this.addAdviceIfAnnotated((Annotation)a, behavior);
51
52            bts = cl.toBytecode();
53
54        } catch (Exception e) {
55            System.err.println("Não foi possível modificar a classe " + clazzName
56                    + ",  erro : " + e.toString());
57        } finally {
58            if (cl != null) cl.detach();
59        }
60        return bts;
61    }
```

Figura 7.7: Implementação do método `processClass`.

Através do *bytecode* da classe que foi recebido por parâmetro, o método `processClass`, monta a classe recebida em um objeto do tipo `CtClass`. Objetos do tipo `CtClass` pos-

suem informações sobre toda a estrutura de uma classe, sendo assim percorre-se todos os métodos existentes na classe em busca de métodos anotados. Para cada anotação existente em um método é invocada uma chamada ao método `addAdviceIfAnnotated`. A implementação deste método pode ser vista na figura 7.8.

```
65    private void addAdviceIfAnnotated(Annotation annotation, CtBehavior behavior)
66            throws NotFoundException
67    {
68        String typeAnnotation = annotation.annotationType().getName();
69
70        if(typeAnnotation.equals("framework.aop.annotations.Before") ||
71           typeAnnotation.equals("framework.aop.annotations.After")  ||
72           typeAnnotation.equals("framework.aop.annotations.Around"))
73        {
74            this.addNativeAdvice(annotation, behavior);
75        }else
76        {
77            Annotation a = annotation.annotationType().getAnnotation(Extension.class);
78
79            if(a!=null)
80                this.addExtensionAdvice((Extension)a, behavior);
81        }
82    }
```

Figura 7.8: Implementação do método `addAdviceIfAnnotated`.

O método `addAdviceIfAnnotated` é responsável por verificar se a anotação presente no método é uma anotação suportada pelo *framework*, se for repassa a responsabilidade para o método mais apropriado para tratar o modelo de anotação utilizado.

Se a anotação for nativa é chamado o método `addNativeAdvice`, caso contrário ela será uma anotação de extensão, cujo método chamado será `addExtensionAdvice`. A implementação de `addNativeAdvice` e `addExtensionAdvice`, são mostradas nas figuras 7.9 e 7.10, respectivamente.

```
84    private void addNativeAdvice(Annotation annotation, CtBehavior behavior)
85    {
86        String typeAnnotation = annotation.annotationType().getSimpleName();
87        AdviceType adviceType = AdviceType.valueOf(typeAnnotation);
88
89        for(Class<? extends IAspect> aspect : this.getListOfAspects(annotation))
90        {
91            try
92            {
93                Weaver weaver = new Weaver(behavior,aspect,adviceType);
94                weaver.combine();
95
96            }catch(Exception e)
97            {
98                System.err.println(e.getMessage());
99            }
00        }
01    }
```

Figura 7.9: Implementação do método `addNativeAdvice`.

```
103    private void addExtensionAdvice(Extension extension, CtBehavior behavior)
104    {
105        try
106        {
107            Weaver weaver = new Weaver(behavior,extension.aspect(),extension.adviceType());
108            weaver.combine();
109
110        }catch(Exception e)
111        {
112            System.err.println(e.getMessage());
113        }
114    }
```

Figura 7.10: Implementação do método `addExtensionAdvice`.

O foco dos métodos representados nas figuras 7.9 e 7.10 se resume em recolher as informações contidas nas anotações (anotações nativas e de extensão), e posteriormente mandar as informações recolhidas para o combinador (*Weaver*). As informações que o combinador precisa receber são: método interceptado, aspecto e tipo de *advice*.

A classe `Weaver`, responsável pela combinação será detalhada na seção 7.3.4, neste mesmo capítulo.

7.3.3.1 Anotações nativas

As anotações nativas são anotações (*annotations*) utilizadas por padrão pelo *framework* para marcar o tipo de *advice* utilizado e a qual aspecto este *advice* pertence.

O *framework* possui 3 anotações nativas: `@Before, @After, @Around`. Sendo que todas elas recebem um parâmetro que representa o aspecto que possui o *advice*, da seguinte forma: `NomeDoAspecto.class`.

7.3.3.2 Anotações de extensão

As anotações de extensão foram criadas para oferecer ao desenvolvedor uma forma de estender as anotações nativas, dando-lhes a oportunidade de criar uma API mais simples e uma melhor descrição do interesse transversal implementado como aspecto.

Por exemplo se é criado um aspecto para modularizar a lógica de *cache*, sendo que esta lógica envolve apenas o *advice around*. De forma nativa a anotação utilizada para marcar a utilização de *cache* em um método seria a seguinte: `@Around(Cache.class)`. Por outro lado se o desenvolvedor criar uma anotação que estende a anotação `@Around`, ele pode conseguir o seguinte resultado na utilização: `@Cache`, obtendo uma leitura mais limpa e uma API personalizada. A criação e a utilização de anotações de extensão será demonstrada no capítulo 8 referente ao estudo de caso.

7.3.4 Combinador

A utilização de AOP junto à OOP, resulta em um sistema altamente modularizado, organizado e semanticamente correto à nível de código-fonte. Mas a integração entre os dois paradigmas deve ser feita em algum momento para que haja uma comunicação entre os módulos. Esta comunicação é feita à nível de *bytecode*, sendo assim o desenvolvedor não visualiza esta comunicação. A única forma de relacionamento entre os dois paradigmas à nível de código-fonte é através de anotações, tornando muito mais simples o entendimento e a aplicação de relacionamentos entre módulos. Desta forma os módulos se tornam completamente desacoplados, ou seja, um módulo não depende do outro diretamente.

A responsável por realizar essas combinações à nível de *bytecode* é a classe `Weaver`. Esta classe faz uso das técnicas de *Bytecode Enginnering* através da utilização do *framework* Javassist.

```
20 class Weaver {
21
22     private CtBehavior target;
23     private CtClass    aspect;
24     private CtClass    classTarget;
25     private AdviceType adviceType;
26
27     public Weaver(CtBehavior target, Class<? extends IAspect>  aspect, AdviceType adviceType)
28             throws NotFoundException
29     {
30         this.target      = target;
31         this.aspect      = ClassPool.getDefault().get(aspect.getName());
32         this.adviceType  = adviceType;
33         this.classTarget = this.target.getDeclaringClass();
34     }
```

Figura 7.11: Atributos e construtor da classe `Weaver`.

A classe `Weaver` possui quatro atributos, como pode ser visto na figura 7.11:

target Este atributo irá armazenar o método alvo de interceptação e a sua estrutura, para possíveis modificações.

aspect Este atributo armazena a classe que representa o aspecto utilizado pelo método alvo. Este atributo é uma das chaves para a combinação.

classTarget Este atributo armazena a classe que possui o método alvo, ou seja, alterações realizadas fora do método alvo serão feitas através deste objeto.

adviceType Este atributo é utilizado para saber qual o tipo de *advice* do aspecto está sendo utilizado. Este atributo será fundamental para o recolhimento de informações

do aspecto.

Para construir um novo combinador é preciso informar algumas regras básicas de combinação. Estas regras são informadas através do construtor que recebe uma instância do método alvo, o *.class* do aspecto (`nomeDoAspecto.class`), e o tipo de *advice* da combinação.

A classe estática `ClassPool` é um canal intermediário de comunicação entre o *Class Loader* e o *framework*. No construtor é necessário acessar o *Class Loader* através do `ClassPool` para conseguir uma instância de `CtClass` para acessar as estrutura do aspecto.

Após a execução do construtor da classe `Weaver`, ou seja, após a criação do combinador, é executado o método `combine`, que inicia o processo de combinação. A implementação do método `combine` é mostrada na figura 7.12.

```
36    public void combine()
37            throws ClassNotFoundException,
38                    CannotCompileException,
39                    NotFoundException
40    {
41        this.addFields();
42        this.addAdviceMethod();
43    }
```

Figura 7.12: Implementação do método `combine`.

A implementação do método `combine` é bem simples, ele apenas chama outros dois métodos. O `addFields` e seus "sub-métodos"encapsulam toda a lógica de adição de novos campos na classe alvo, ou seja, todos campos existentes no aspecto e que não estão na classe alvo são copiados. Já o método `addAdviceMethod` e seus "sub-métodos"encapsulam a lógica de combinação do método interceptado com o seu *advice* correspondente.

Nas próximas subseções serão abordados os detalhes de implementação dos dois métodos citados acima.

7.3.4.1 AddFields

Para iniciar realmente o processo de combinação, deve-se começar copiando os campos existentes no aspecto para a classe alvo, caso esses campos ainda não existam. Este processo inicia pelo método `addFields`, cuja implementação está na figura 7.13.

```
45    private void addFields()
46            throws ClassNotFoundException, CannotCompileException
47    {
48        for(CtField field : this.aspect.getDeclaredFields())
49        {
50            try
51            {
52                this.classTarget.getDeclaredField(field.getName());
53
54            }catch(Exception e)
55            {
56                this.addField(field);
57            }
58        }
59    }
```

Figura 7.13: Implementação do método `addFields`.

O método `addFields` verifica se cada campo existente no aspecto, existe na classe alvo, caso não exista é chamado o método `addField` para realizar a cópia deste campo para a classe alvo. Pode-se notar que a API do Javassist não oferece um método para verificar se determinado campo existe, mas ela lança uma exceção quando tenta-se acessar algum membro inexistente. Pensando desta forma fez-se o uso de um bloco `try-catch` para verificar se o campo já existe, se a execução do método cair no bloco `catch` significa que uma exceção foi capturada e o campo não existe, providenciando então a sua cópia pelo método `addField`. O método `addField` e sua implementação pode ser vista na figura 7.14.

```
61    private void addField(CtField field)
62            throws ClassNotFoundException, CannotCompileException
63    {
64        String initialValue = this.getInitialValueField(field);
65
66        if(initialValue.equals(""))
67            this.classTarget.addField(new CtField(field,this.classTarget));
68        else
69            this.classTarget.addField(new CtField(field,this.classTarget),initialValue);
70    }
```

Figura 7.14: Implementação do método `addField`.

O método `addField` é responsável por adicionar o campo à classe alvo. Primeiramente é obtido o valor inicial do campo através do método `getInitialValueField`, caso não tenha sido declarado nenhum valor inicial é retornado uma `String` vazia. Após obter, ou não o valor inicial do campo, é utilizado o método `addField` da classe alvo para realizar a adição do campo, este método recebe uma nova instância de *CtField* que recebe o campo a ser copiado, e a classe que receberá este campo.

Na figura 7.15 é mostrado a implementação de `getInitialValueField`.

```
72    private String getInitialValueField(CtField field)
73            throws ClassNotFoundException
74    {
75        String value = "";
76        Object[] annotationsField = field.getAnnotations();
77
78        for (Object annotationField : annotationsField)
79        {
80            Annotation f = (Annotation) annotationField;
81            if (f.annotationType().getSimpleName().equals("Initialize")) {
82                value = ((Initialize)f).value();
83            }
84        }
85        return value;
86    }
```

Figura 7.15: Implementação do método `getInitialValueField`.

No aspecto o valor inicial de um campo pode ser informado através da utilização da anotação `@Initialize("valor")` antes do campo a ser inicializado, esta anotação foi criada pois através do Javassist não é possível pegar o valor de um campo. Desta forma se um campo for inicializado no aspecto através do operador =, esta inicialização não estará visível na combinação, por exemplo: `private int idade = 18`. Sendo assim a inicialização correta do campo seria: `@Initialize("18") private int idade`.

Basicamente então para o método `getInitialValueField` pegar o valor inicial do campo, basta ler as anotações existentes no campo e verificar se ele possui a anotação `@Initialize`, se possuir retorna o valor inicial, caso contrário retorna uma `String` vazia.

7.3.4.2 AddAdviceMethod

Após a cópia dos campos para a classe alvo, é necessário realizar a integração entre o *advice* e o método interceptado. É dada a partida para essa integração através da chamada ao método `addAdviceMethod`, cuja responsabilidade é fazer uma cópia do *advice* e repassar esta cópia para o método correspondente ao tipo de *advice*. Os métodos que combinam cada tipo de *advice* são: `addBefore`, `addAfter` e `AddAround`. Na figura 7.16 é mostrado a implementação do método `addAdviceMethod`.

O método `createMethod` é responsável por fazer a cópia do *advice* para a classe alvo, e também por adicionar configurações ao método caso seja necessário. Este método faz a verificação se a classe alvo já não possui um método correspondente ao *advice*, caso não possua é feita a cópia. Se o *advice* utilizado for *before* ou *after* é adicionado um

```
84    private void addAdviceMethod()
85            throws CannotCompileException, NotFoundException
86    {
87        String adviceMethodName = this.createMethod();
88
89        switch(this.adviceType)
90        {
91            case Before: this.addBefore(adviceMethodName);
92                break;
93            case After: this.addAfter(adviceMethodName);
94                break;
95            case Around: this.addAround(adviceMethodName);
96                break;
97        }
98    }
```

Figura 7.16: Implementação do método `addAdviceMethod`.

bloco `try/catch` ao *advice* copiado, para tratar possíveis exceções lançadas por estes *advices*. Após estes procedimentos a cópia é realmente adicionada à classe alvo. A figura 7.17 mostra a implementação de `createMethod`.

```
150   private String createMethod()
151           throws CannotCompileException,
152                   NotFoundException
153   {
154       String advice = this.adviceType.toString().toLowerCase();
155       String methodName = this.aspect.getSimpleName().toLowerCase()+"_"+advice;
156
157       try
158       {
159           this.classTarget.getDeclaredMethod(methodName);
160
161       }catch(NotFoundException e)
162       {
163           CtMethod adviceMethod = CtNewMethod.copy(aspect.getDeclaredMethod(advice),
164                   methodName, this.classTarget, null);
165
166           this.addTryCatch(adviceMethod);
167           this.classTarget.addMethod(adviceMethod);
168       }
169       return methodName;
170   }
```

Figura 7.17: Implementação do método `createMethod`.

Para realizar a combinação do método com um *advice* do tipo *before* é utilizado o método `addBefore`. O nome do método copiado recebido por parâmetro é formado da seguinte maneira: `aspecto$advice`. Para adicionar uma chamada à cópia do *advice* antes da execução do método interceptado é necessário utilizar o método `insertBefore` presente na classe `CtMethod` do Javassist. A chamada para o método `insertBefore` deve ser realizada pela instância do método interceptado. Na figura 7.18 está a implementação do método `addBefore` da classe `Weaver`.

Adicionou-se uma variável local que chama-se `methodExecution` e é do tipo `MethodExecution` ao método alvo, esta variável é utilizada para armazenar as informações do método (valor dos parâmetros, tipo dos parâmetros, etc). Para formar a cadeia de caracteres que representa a instanciação da variável é utilizado o método

```
100    private void addBefore(String adviceMethodName)
101            throws CannotCompileException,
102                    NotFoundException
103    {
104        this.target.addLocalVariable("methodExecution",
105        ClassPool.getDefault().
106        get("framework.aop.structures.MethodExecution"));
107
108        this.target.insertBefore(
109                this.createMethodInfo("methodExecution", this.target)
110                +adviceMethodName+"($0,methodExecution);");
111    }
```

Figura 7.18: Implementação do método `addBefore`.

`createMethodInfo` (na figura 7.19 encontra-se a implementação deste método). O método `insertBefore` recebe como parâmetro uma `String` que representa o código que deve ser adicionado antes da execução do método alvo, esta `String` é formada pela instanciação da variável local e pela chamada á cópia do *advice*. Para realizar a chamada ao *advice* que foi copiado para a classe alvo, deve-se ter dois parâmetros em mãos, o primeiro representado por (*$0*) representa a instância real da classe que possui o método, e o segundo a instância de `MethodExecution`, explicado anteriormente.

```
146    private String createMethodInfo(String variableName,CtBehavior method)
147    {
148        return
149        variableName+"= new framework.aop.structures.MethodExecution(\""
150        +method.getName()+"\",new framework.aop.structures.Params($args,$sig));";
151    }
```

Figura 7.19: Implementação do método `createMethodInfo`.

Pode-se notar que para instanciar a classe `Params` na `linha 150` da figura 7.19 é passado dois parâmetros, o primeiro (*$args*) representa um objeto do tipo `Objeto[]` que possui os valores dos parâmetros e o segundo (*$sig*) é do tipo `Class[]` que possui os tipos de cada parâmetro. Esta sintaxe utilizada em forma de `String` é traduzida para *bytecode* pelo Javassist.

O método `addAfter` foi implementado praticamente da mesma forma que o método `addBefore`, sendo que a única diferença é que `addAfter` faz a utilização do método `insertAfter` para adicionar código depois, e o `addBefore` utiliza o método `insertBefore` para adicionar antes. A implementação de `addAfter` pode ser vista na figura 7.20.

O *advice* do tipo *around* se resume em envolver a execução de um método com trechos de código, ou seja, nada mais é do que adicionar código antes e depois da chamada a um método. O tipo *around* é um pouco mais complexo que *before* e *after*, e também o mais utilizado no encapsulamento de interesses transversais mais complexos.

```
113    private void addAfter(String adviceMethodName)
114            throws CannotCompileException,
115                    NotFoundException
116    {
117        this.target.addLocalVariable("methodExecution",
118        ClassPool.getDefault().
119        get("framework.aop.structures.MethodExecution"));
120
121        this.target.insertAfter(
122                this.createMethodInfo("methodExecution", this.target)
123                +adviceMethodName+"($0,methodExecution);");
124    }
```

Figura 7.20: Implementação do método `addAfter`.

O método da classe `Weaver` responsável por realizar a combinação do método interceptado com o *advice around* é o método `addAround`, e a sua implementação se encontra na figura 7.21.

```
126    private void addAround(String adviceMethodName)
127            throws CannotCompileException,
128                    NotFoundException
129    {
130        String copyName = this.target.getName()+"$Copy";
131
132        CtMethod copyMethod = CtNewMethod.copy((CtMethod)this.target,
133                                               copyName, this.classTarget,
134                                               null);
135        this.classTarget.addMethod(copyMethod);
136
137        this.target.setBody("{framework.aop.structures.MethodExecution "+
138                            this.createMethodInfo("methodExecution", this.target)
139                            +"return ($r)"+adviceMethodName+
140                            "($0,methodExecution);\"\";}");
141    }
```

Figura 7.21: Implementação do método `addAround`.

A implementação de `addAround` segue o mesmo modelo de implementação de `addBefore` e `addAfter`, porém desenvolvida de forma diferente. Primeiramente é preciso realizar uma cópia do método alvo, para manter a implementação intacta, o nome do método copiado terá a seguinte forma: `nomeOriginalMétodo$Copy`. Tendo a cópia do método, deve-se então reescrever o corpo do método original, para isso utiliza-se o método `setBody` disponível em instâncias de `CtMethod` do Javassist. Para adicionar variáveis locais ao corpo do método não é utilizado o método `addLocalVariable`, sendo assim foi utilizado o formato de *String* para adicionar a variável `methodExecution`. É adicionado também ao corpo do método a chamada ao *advice* copiado.

Para informar ao *advice around* o momento que deve ser executado o método original dentro do seu escopo, deve-se fazer a utilização do método `proceed` presente na instância da classe `MethodExecution` recebida por parâmetro. Na figura 7.22 um exemplo da utilização do `proceed` em um *advice around*.

```
9    @Override
10   public Object around(Object source, MethodExecution method){
11
12       //Lógica do interesse transversal executa antes
13
14       Object result = method.proceed(source);
15
16       //Lógica do interesse transversal executa depois
17
18       return result;
19   }
```

Figura 7.22: Exemplo de implementação do *advice around*.

O método `proceed` faz o uso de reflexão para invocar uma chamada à copia do método original feita no `addAround`, esta chamada é feita pelo objeto `source`, que seria uma instância da classe alvo, ou seja, a classe que recebeu a cópia do método. A implementação de `proceed` realizada na classe `MethodExecution`, pode ser vista na figura 7.23.

```
24   public Object proceed(Object source)
25   {
26       try
27       {
28           Method metodo = source.getClass().getMethod(this.name+
29                           "$Copy", params.getTypeArgs());
30           return metodo.invoke(source, params.getArgs());
31
32       }catch(Exception e)
33       {
34           e.getStackTrace();
35       }
36       return null;
37   }
```

Figura 7.23: Implementação do método `proceed`.

7.4 Considerações finais

Este capítulo abordou as características do projeto e todos os seus detalhes de implementação, com o intuito de oferecer uma base teórica e prática para estudos sobre AOP. No próximo capítulo será utilizado os conhecimentos aqui adquiridos para o desenvolvimento de um estudo de caso.

8 ESTUDO DE CASO - INSTRUMENTAÇÃO

Neste capítulo será desenvolvido um estudo de caso para demonstrar as funcionalidades do *framework*. Neste capítulo será abordado a parte de análise e desenvolvimento do estudo de caso, e também detalhes sobre a combinação dos aspectos com as classes.

8.1 Introdução

Nos dias atuais a quantidade de produtos, serviços e ferramentas desenvolvidas para resolução de um problema é imensa, devido o aumento das necessidades em geral. No mundo do desenvolvimento de *software* não é diferente, sistemas, aplicativos e ferramentas também são desenvolvidas visando na maioria das vezes a resolução de um problema, ou facilitar (automatizar) a execução de uma tarefa.

Pensando nisso são desenvolvidos casos de testes e/ou estudos de casos para demonstrar a eficiência e a capacidade de resolução do problema proposto, levando em conta a visão do cliente (usuário). Para demonstrar a capacidade do *framework* de encapsular interesses transversais e a facilidade adquirida na sua utilização, será desenvolvido um estudo de caso para comprovar de forma prática os conceitos apresentados.

8.2 O problema

Um sistema hipotético, é formado por diversos módulos, sendo que cada módulo tem uma responsabilidade específica. Levando em conta que o carregamento destes módulos se dá durante a inicialização do sistema, deve-se calcular o tempo de carregamento de cada módulo, e também o tempo total de carregamento do sistema, armazenando estas informações em um arquivo de *log*[1]. A solução deste problema deve se dar de forma que

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

código referente ao cálculo do tempo de execução não esteja entrelaçado e/ou espalhado pelos módulos do sistema, fazendo com que o sistema e seus módulos sejam totalmente independentes em relação ao gerenciamento de tempo de cada módulo.

8.2.1 Instrumentação

Instrumentação é o nome dado à inserção de código adicional ao sistema, visando analisar e avaliar diversas medidas de desempenho durante a execução de um programa (CABRAL, 2005) .

A medida de desempenho aqui utilizada será o cálculo do tempo de execução de um método, com o intuito de avaliar o tempo de carregamento do sistema, realizando uma análise nos métodos envolvidos nesta tarefa.

8.2.2 Solução

Para resolução do problema vai ser utilizado o *framework* desenvolvido para criação dos aspectos, que também será responsável pela integração entre as classes referentes ao sistema e o aspecto referente ao cálculo de execução dos métodos.

A única forma de comunicação entre a classe e o aspecto será uma anotação (nativa ou de extensão), cuja principal finalidade é marcar quais métodos devem ser interceptados, de que forma, e o que deve ser executado durante esta interceptação.

8.3 Análise

Para descrever a parte de análise do estudo de caso foi desenvolvido um diagrama de classes de implementação, que mostra toda a estrutura e o relacionamento entre as partes envolvidas no projeto.

A figura 8.1 mostra o diagrama de classes do estudo de caso, que serviu de base para a implementação.

As classes referentes ao *framework* presentes no diagrama da figura 8.1 não serão descritas neste capítulo porque já foram descritas no capítulo 7. Abaixo está a descrição das outras classes presentes no diagrama.

Main Esta classe é a responsável por iniciar o sistema e suas configurações.

Sistema Esta classe representa o sistema hipoteticamente. Esta classe possui os módulos do estudo.

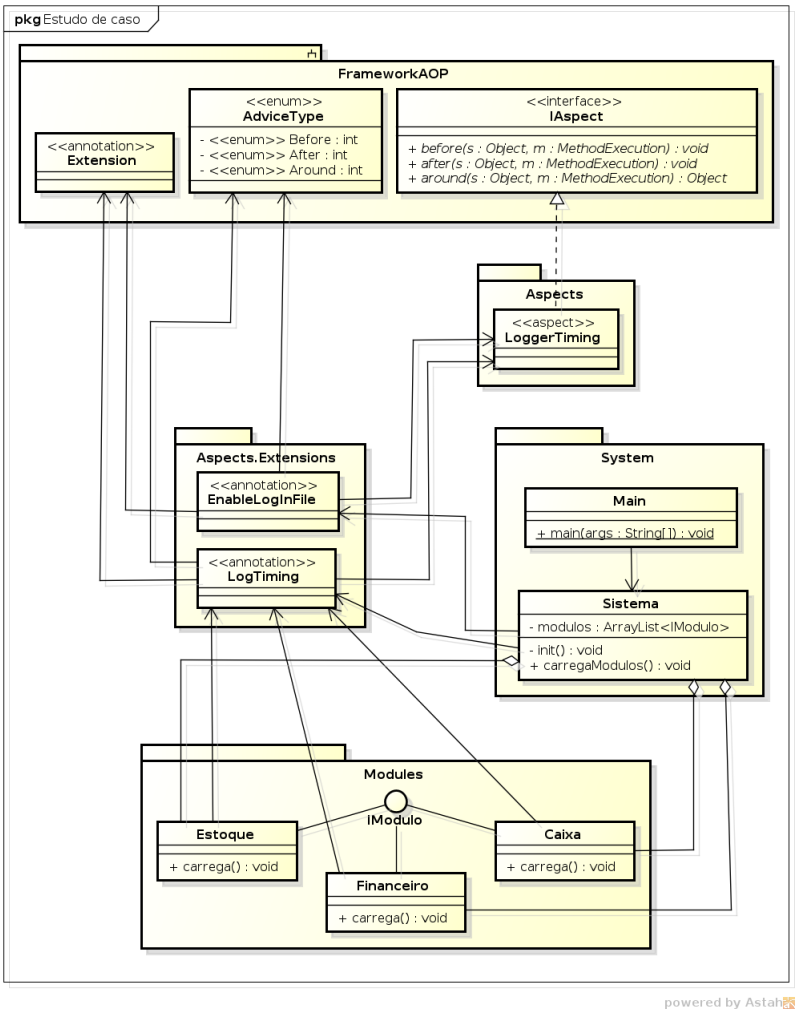

Figura 8.1: Diagrama de classes do estudo de caso.

IModulo Interface que deve ser implementada por todos os módulos. Esta interface obriga a implementação.

Estoque, Caixa, Financeiro Classes hipotéticas que representam os módulos do sistema.

LoggerTiming Este é o aspecto que irá encapsular a lógica responsável por calcular o tempo de execução dos métodos, e também a lógica que ativa o armazenamento das informações obtidas através de instrumentação em um arquivo de *log*.

EnableLogInFile Esta anotação é responsável por marcar o método que irá ativar o armazenamento em *log* antes de sua execução.

LogTiming Esta anotação é utilizada para informar que o método anotado terá o seu tempo de execução calculado.

8.4 Implementação

Para demonstrar as funcionalidades do *framework* procurou-se desenvolver um estudo de caso simples, mas que cumpra o esperado. Pensando desta forma, decidiu-se focar a parte implementação não no sistema que representa os interesses centrais, mas sim nos aspectos que representam os interesses transversais de um sistema. Sendo assim, não foi implementando um sistema funcional, mas sim uma estrutura que representasse este sistema.

8.4.1 Criação de um aspecto

Para criação de um aspecto utilizando o *framework* deve-se implementar a interface `IAspect` presente no pacote `framework.aop` detalhado no diagrama de classes da figura 7.2. A implementação do aspecto `LoggerTiming` está representada nas figuras 8.2 e 8.3.

```
11 public class LoggerTiming implements IAspect {
12
13     private long start;
14
15     @Override
16     public Object around(Object source, MethodExecution method){
17
18         Logger log = Logger.getLogger("logTiming");
19         this.start = System.currentTimeMillis();
20
21         Object result = method.proceed(source);
22
23         long tempoTotal = (System.currentTimeMillis()-this.start);
24
25         if(tempoTotal < 5000)
26             log.info("O método "+method.getName()+" executou em: "
27             +((double)tempoTotal)/1000+"s.");
28         else
29             log.warning("O método "+method.getName()+" está muito lento. "+
30                     "Executou em " +((double)tempoTotal)/1000+"s.");
31         return result;
32     }
```

Figura 8.2: Implementação do *advice* `around` do aspecto `LoggerTiming`.

O *advice* `around` do aspecto `LoggerTiming` possui a função de calcular o tempo de execução de um método, de forma que todo o código necessário para execução desta função está encapsulado neste *advice*. Pode-se notar que na linha `21` é onde ocorre a execução do método interceptado.

Caso o método executado tenha sua execução realizada em um tempo maior que 5 segundos, é mostrado no *log* um aviso, caso contrário é mostrado um *log* de informação

com o tempo de execução.

```
34    @Override
35    public void before(Object source, MethodExecution method)
36            throws Exception
37    {
38        FileHandler fileHandler = new FileHandler("logTiming.log");
39        fileHandler.setFormatter(new SimpleFormatter());
40        fileHandler.setLevel(Level.ALL);
41        Logger.getLogger("").addHandler(fileHandler);
42    }
```

Figura 8.3: Implementação do *advice* `before` do aspecto `LoggerTiming`.

Já o *advice* `before` possui a função de ativar o armazenamento das informações de *log* em um arquivo, para realização de futuras análises. O arquivo de *log* será criado na pasta raiz do projeto com o nome de `logTiming.log`.

8.4.2 Criação das anotações de extensão

Para realizar a comunicação entre o aspecto e as classes é necessário utilizar as anotações nativas, ou criar anotações de extensão. Neste estudo de caso optou-se por criar anotações de extensão por ser uma forma mais descritiva e de melhor entendimento, que facilita a leitura do código fonte.

Na figura 8.4 e 8.5 é demonstrada a criação das anotações `EnableLogInFile` e `LogTiming`, respectivamente.

```
11 @Target(ElementType.METHOD)
12 @Extension(aspect=LoggerTiming.class,
13             adviceType=AdviceType.Before)
14 public @interface EnableLogInFile {
15 }
```

Figura 8.4: Criação da anotação de extensão `EnableLogInFile`.

A anotação de extensão `EnableLogInFile` será utilizada para realizar a comunicação entre o *advice* `before` do aspecto `LoggerTiming` e o método `init` da classe `Sistema`, com o intuito de ativar o armazenamento das informações de *log* em um arquivo.

```
10 @Target(ElementType.METHOD)
11 @Extension(aspect=LoggerTiming.class,
12             adviceType=AdviceType.Around)
13 public @interface LogTiming {
14 }
```

Figura 8.5: Criação da anotação de extensão `LogTiming`.

Por outro lado a anotação de extensão `LogTiming` irá ser utilizada nos métodos que terão o tempo calculado, ou seja, os métodos responsáveis pelo carregamento dos módulos.

8.4.3 Utilizando anotações de extensão

As anotações de extensão serão utilizadas para realizar a comunicação do aspecto com o método interceptado. Na figura 8.6 é mostrado a utilização da anotação `EnableLogInFile`.

```
19    @EnableLogInFile
20    private void init()
21    {
22        //configurações do sistema
23    }
```

Figura 8.6: Utilizando a anotação de extensão `EnableLogInFile`.

Neste estudo de caso foi utilizado a anotação `EnableLogInFile` no método `init` da classe `Sistema`, para indicar que antes da inicialização do sistema deve-se ativar o armazenamento em arquivo de *log*.

```
5  public class Estoque implements IModulo {
6
7      @LogTiming
8      @Override
9      public void carrega() {
10         System.out.println("Carregando o módulo Estoque.");
11         for(int i=0;i<4490000;i++)
12             System.out.println("");
13     }
14 }
```

Figura 8.7: Utilização da anotação `LogTiming` no método `carrega`.

Foi utilizada a anotação `LogTiming` no método `carrega` de todos os módulos (`Estoque, Caixa e Financeiro`) do sistema. O método `carrega` foi implementado de forma hipotética em todos os módulos, para isso utilizou-se um *loop* para simular o tempo em que o método ficar executando.

8.5 Execução do sistema

Após realizar a comunicação entre o(s) aspecto(s) e as classes, já é possível executar o sistema. Para isso deve-se compilar o projeto, acessar a pasta `bin` e executar o seguinte comando para iniciar o sistema utilizando o *framework* para realização da combinação:

```
java -javaagent:../dist/agent.jar System.Main -classpath
../lib/javassist.jar
```

O arquivo *.jar* que representa o *framework* deve estar localizado na pasta `dist` e a biblioteca Javassist utilizada pelo *framework* deve estar na pasta `lib`, ambas dentro do projeto, pois são dependências necessárias para a realização da combinação durante o carregamento das classes.

Após a execução do estudo de caso é criado o arquivo de *log* dentro da pasta do projeto, caso ele ainda não exista. A figura 8.8 mostra uma das saídas do arquivo de *log* obtidas durante o carregamento dos módulos do sistema.

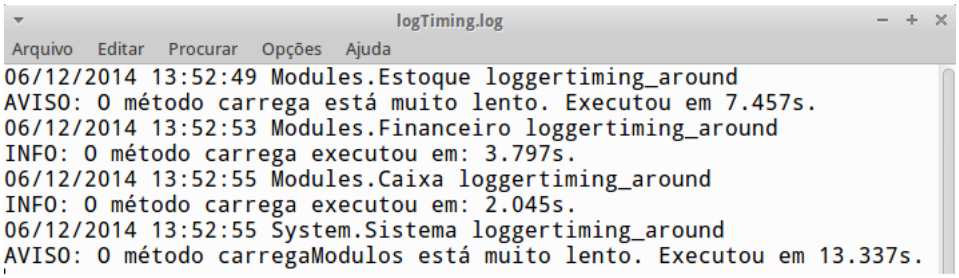

Figura 8.8: Saída do arquivo de *log* após o carregamento dos módulos.

Pode-se perceber que o módulo que demorou mais a carregar foi módulo de `Estoque` que levou 7.457 segundos em seu carregamento, sendo que o tempo total de carregamento dos módulos do sistema foi de 13.337 segundos.

8.6 Considerações finais

Este capítulo focou no projeto e implementação de um estudo de caso, que utilizou um sistema hipotético para demonstrar a criação e utilização de aspectos no encapsulamento de interesses transversais de um sistema, visando comprovar a eficiência do *framework* desenvolvido na resolução deste problema. Também foi abordado neste capítulo a forma como é feita a comunicação entre aspectos e classes de forma simples com a utilização do *framework*.

9 CONCLUSÃO

Durante todo o processo de pesquisa e desenvolvimento deste trabalho, foram abordados diversos conceitos, que giraram em torno da resolução de um único problema, encapsular os interesses transversais de um sistema Java. A solução proposta envolveu conceitos de Programação Orientada a Aspectos, aplicação de técnicas de *Bytecode Engineering* e utilização de Meta Programação (anotações e reflexão).

A abordagem escolhida para resolução do problema foi o desenvolvimento de um *framework* que fosse extensível de forma a servir de base para outras ferramentas específicas. O trabalho foi desenvolvido em 4 etapas que foram executadas paralelamente com duração total de 9 meses, sendo elas em ordem de finalização: pesquisa, análise, implementação e monografia.

Durante a etapa de pesquisa foi feito o estudo dos conceitos envolvidos no trabalho, e o levantamento das referências utilizadas. A etapa de análise envolveu a parte de modelagem do *framework* e do estudo de caso, representada em forma de diagramas, visando facilitar a etapa de implementação. Já a etapa de implementação envolveu a codificação do *framework* utilizando as tecnologias e ferramentas citadas no capítulo 5, e posteriormente foi realizada a implementação do estudo de caso, que demonstrou de forma clara a eficiência do *framework* no encapsulamento dos interesses transversais, e na integração das classes com os aspectos. E por último a etapa que envolveu a escrita desta monografia, que foi executada paralelamente com as etapas anteriores, sendo a última a ser finalizada.

Os resultados obtidos foram satisfatórios, devido a cumprimento dos objetivos propostos, e também pela facilidade de uso do *framework* desenvolvido, comprovados após a etapa de implementação. O *framework* também poderá servir de base para estudos e pesquisas relacionadas à técnicas de *Bytecode Engineering* e Programação Orientada a Aspectos mais especificamente.

9.1 Trabalhos futuros

Para não prolongar o escopo do projeto, algumas funcionalidades não foram implementadas, deixando-as então para implementações futuras. Dentre os principais trabalhos futuros a serem realizados estão:

- Implementar o conceito de *pointcut* no *framework*. A implementação de *pointcut* é considerada uma melhoria e não um requisito para o *framework*.

- Adicionar outros tipos de *advices* ao *framework*. Por exemplo existem outros 2 tipos de *advice after*: *after returning* e *after throwing*, que são executados após o retorno de um método e após o lançamento de uma exceção, respectivamente.

- Criar um compilador para realizar as combinações em tempo de compilação. Uma vantagem em ter esta funcionalidade, é que conseguiria-se desenvolver por exemplo aspectos de tratamento de exceção totalmente modularizados para exceções checadas, o que não é possível com o *framework* atual, devido as restrições do compilador padrão.

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.

CABRAL, B. M. B. **Instrumentação de Código na Plataforma .NET**. 2005. Tese (Doutorado em Ciência da Computação) — Universidade de Coimbra.

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.

MATOS, T. **Você sabe o que é log?** Disponível em: <http://www.tiagomatos.com/blog/voce-sabe-o-que-e-log>. Acesso em: 03 Dezembro. 2014.

MILINKOVICH, M. **About the Eclipse Foundation**. Disponível em: <http://www.eclipse.org/org/>. Acesso em: 19 Outubro. 2014.

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

SRINIVASAN, K. **Introduction to Java Agents**. Disponível em: <http://www.javabeat.net/introduction-to-java-agents/>. Acesso em: 19 Outubro. 2014.

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 é 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.

Log é o arquivo onde é registrado informações relevantes sobre o comportamento do sistema (MATOS, 2014).

APÊNDICE A VERSÃO COMPLETA DOS DIAGRAMAS