Componentes Reutilizáveis em Java com

Reflexão e Anotações

Casa do Código

EDUARDO GUERRA

Edição

Adriano Almeida
Vivian Matsui

Revisão

Bianca Hubert
Vivian Matsui

[2017]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso e PDF: 978-85-66250-50-3

EPUB: 978-85-5519-144-2

MOBI: 978-85-5519-145-9

Você pode discutir sobre este livro no Fórum da Casa do Código: http://forum.casadocodigo.com.br/.

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

AGRADECIMENTOS PESSOAIS

A Deus primeiro agradeço Por ter iluminado meu caminho Pois sei em meu coração Que eu nunca estou sozinho

Agradeço aos meus pais Por cada ensinamento Deles aprendi valores Presentes a todo momento

A Duda filhinha querida De quem além de pai sou amigo Seu sorriso e seu carinho Carrego sempre comigo

A Bia filhinha querida Mais esperta a cada dia A cada coisinha que fala Vai trazendo sempre alegria

A Roberta esposa amada Companheira sempre presente Sempre firme ao meu lado Me fazendo seguir em frente

Adiciono mais um verso Não que tenha me esquecido Pra dizer que minha vida Sem vocês não faz sentido

AGRADECIMENTOS PROFISSIONAIS

Antes de mencionar alguém de forma específica, gostaria de agradecer a todos meus professores, alunos, colegas de trabalho, amigos e companheiros que de alguma forma contribuíram para meu conhecimento e me incentivaram a seguir em frente. Nossa vida é feita de pessoas e de experiências, e é a partir disso que seguimos nossos caminhos e realizamos nossas escolhas.

Primeiramente deixo um agradecimento institucional ao ITA, um instituto que é referência nacional e internacional em engenharia, em onde me formei em Engenharia da Computação e fiz meu mestrado e doutorado. Além disso, também foi um local onde tive a oportunidade de atuar como professor por mais de 5 anos. Foram experiências que me marcaram e são uma importante parte da minha história. Eu saí do ITA, mas com certeza o ITA não saiu de mim! Agradeço em especial ao professor Clovis Fernandes, meu orientador na graduação, no mestrado e no doutorado. Um grande amigo, que sempre me incentivou e que foi grande mentor, cujos ensinamentos foram muito além de questões técnicas.

Agradeço à revista MundoJ, principalmente ao Marco Guapo, inicialmente por ter aberto espaço para que eu escrevesse sobre minhas ideias e meus conhecimentos. Em seguida, Guapo confiou a mim o conteúdo da revista como editor-chefe. Através da minha atuação na revista eu pude aprender muita coisa e me manter sempre atualizado durante os últimos anos. Por meio dos artigos que escrevi, ganhei experiência e tomei gosto pela escrita de conteúdo técnico. Posso afirmar com certeza que a semente desse livro foi plantada em alguns de meus artigos durante os últimos

anos.

Agradeço também à comunidade de padrões, tanto nacional quanto internacional, por ter me influenciado com toda sua cultura e ideias. Sou grato por terem me recebido de braços abertos e por terem fornecido um fórum onde foi possível debater as minhas ideias e obter feedback dos trabalhos que estou realizando. Agradeço em especial pelo apoio e incentivo de Fabio Kon, Fábio Silveira, Jefferson Souza, Uirá Kulezsa, Ademar Aguiar e Filipe Correia. Também agradeço pelos meus gurus Joseph Yoder e Rebecca Wirfs-Brock, com quem aprendi demais e tive discussões muito interessantes sobre modelagem e arquitetura de software.

Agradeço ao INPE, instituição onde trabalho e desenvolvo minha pesquisa. Apesar de estar aqui a pouco tempo, eu me identifiquei muito com seu ambiente propício para o desenvolvimento e aplicação de novas ideias. Agradeço por ter me recebido de braços abertos e confiado em minhas ideias. Tenho certeza de que nos próximos anos poderei contribuir para seu crescimento a partir de minha ideias e da minha pesquisa.

Finalmente, agradeço à Casa do Código por confiar em mim para escrita desse livro. Deixo um agradecimento em especial ao Paulo Silveira e ao Adriano Almeida, por estarem sempre disponíveis a discussões relacionadas ao conteúdo desse livro.

SOBRE O AUTOR

Eduardo Martins Guerra nasceu em Juiz de Fora em 1980 e cursou o ensino básico e médio em escola pública, o Colégio de Aplicação João XXIII. Desde essa época, ele já despertou seu interesse pela programação, começando com a digitação de código Basic que vinha em revistas no seu defasado computador Exato Pro. A diversão era mais ver o efeito de pequenas modificações no código do que o jogo que saía como resultado. Nessa época, pouco depois, também fez um curso de Clipper, onde desenvolveu seu primeiro software: um gerenciador de canil!

Incentivado por seus professores, pela facilidade com disciplinas na área de exatas, prestou o vestibular para o ITA em 1998, logo após o término de seus estudos, e foi aprovado. Durante o curso, ele optou pela carreira militar e pelo curso de computação, além de participar de atividades extracurriculares como projetos para a empresa júnior e aulas no curso pré-vestibular para pessoas carentes, CASD Vestibulares. Nesses cinco anos, Eduardo se apaixonou pela área de desenvolvimento de software e pela possibilidade de usar constantemente sua criatividade para propor sempre soluções inteligentes e inovadoras. Dois dias depois de se formar, casou-se com sua mais forte paixão, sua atual esposa, Roberta.

Após formado, em 2003, foi alocado no Centro de Computação da Aeronáutica de São José dos Campos (CCA-SJ), onde ficaria pelos próximos 4 anos. Durante esse período, trabalhou com o desenvolvimento de software operacional para atender às necessidades da Força Aérea Brasileira, adquirindo uma valiosa

experiência na área.

No final de 2005, com dedicação em tempo parcial, Eduardo conseguiu seu título de mestre em Engenharia da Computação e Eletrônica apresentando a dissertação "Um Estudo sobre Refatoração de Código de Teste". Nesse trabalho, Eduardo desenvolve sua paixão pelo design de software, realizando um estudo inovador sobre boas práticas ao se codificar testes automatizados. Essa dissertação talvez tenha sido um dos primeiros trabalhos acadêmicos no Brasil no contexto de desenvolvimento ágil. No ano de conclusão de seu mestrado, veio sua primeira filha, Maria Eduarda.

Devido à sua sede de conhecimento, ainda nesse período, estudou por conta própria e tirou sete certificações referentes à tecnologia Java, sendo na época um dos profissionais brasileiros com maior número de certificações na área. Além disso, tem ocupado o cargo de editor-chefe da revista MundoJ desde 2006, na qual já publicou dezenas de artigos técnicos de reconhecida qualidade. Esses fatos lhe deram uma visão da indústria de desenvolvimento de software que foi muito importante em sua trajetória, direcionando sua pesquisa e seus trabalhos para problemas reais e relevantes.

No CCA-SJ, Eduardo atuou de forma decisiva no desenvolvimento de frameworks. Seus frameworks simplificaram a criação das aplicações e deram maior produtividade para a equipe de implementação. Um deles, o SwingBean, foi transformado em um projeto open-source e já possui mais de 5000 downloads. A grande inovação e diferencial de seus frameworks estavam no fato de serem baseados em metadados. A partir desse caso de sucesso,

ele decidiu estudar mais sobre como a utilização de metadados poderia ser feita em outros contextos. Foi uma surpresa descobrir que, apesar de outros frameworks líderes de mercado utilizarem essas técnicas, ainda não havia nenhum estudo sobre o assunto. Foi, então, que Eduardo começou sua jornada para estudar e tornar acessível o uso dessa técnica por outros desenvolvedores, pois ele sabia do potencial que os frameworks baseados em metadados têm para agilizar o desenvolvimento de software e o tornar mais flexível.

Então, em 2007, ele ingressou no curso de doutorado do ITA pelo Programa de Pós-Graduação em Aplicações Operacionais – PPGAO. A pesquisa sobre frameworks baseados em metadados se encaixava perfeitamente no objetivo do programa. A criação de arquiteturas flexíveis, nas quais se pode acrescentar funcionalidade de forma mais fácil, é algo crítico para aplicações de comando e controle, foco de uma das áreas do programa. Orientado pelo professor Clovis Torres Fernandes, começou uma pesquisa que envolveu a análise de frameworks existentes, a abstração de técnicas e práticas, a criação de novos frameworks de código aberto e a execução de experimentos para avaliar os conceitos propostos.

Foi nessa época que Eduardo começou a participar e se envolveu na comunidade de padrões. Em 2008 submeteu para o SugarLoafPLoP em Fortaleza os primeiros padrões que identificou em frameworks que utilizavam metadados. Nesse momento, ele se empolgou com o espírito de colaboração da comunidade de padrões, onde recebeu um imenso feedback do trabalho que estava realizando. Desde então se tornou um membro ativo da comunidade, publicando artigos com novos padrões em eventos

no Brasil e no exterior. Além disso, em 2011 participou da organização do MiniPLoP Brasil, em 2012 foi o primeiro brasileiro a ser chair da principal conferência internacional de padrões, o PLoP, e em 2013 também participou da organização do próximo MiniPLoP.

Enquanto realizava seu doutorado, Eduardo foi incorporado, ainda como militar, no corpo docente do Departamento de Ciência da Computação do ITA, onde pôde desenvolver uma nova paixão: ensinar! Durante esse período, ministrou aulas na graduação e em cursos de especialização, e orientou uma série de trabalhos que, de certa forma, complementavam e exploravam outros aspectos do que estava desenvolvendo em sua tese. Em consequência do bom trabalho realizado, foi convidado por três vezes consecutivas para ser o professor homenageado da turma de graduação Engenharia da Computação e duas vezes para a turma de especialização em Engenharia de Software.

Em 2010, apresentou seu doutorado chamado "A Conceptual Model for Metadata-based Frameworks" e concluiu com sucesso essa jornada que deixou diversas contribuições. Devido à relevância do tema, foi incentivado pelos membros da banca a continuar os estudos que vinha realizando nessa área. Apesar de a tese ter trazido grandes avanços, ele tem consciência de que ainda há muito a ser feito. Uma de suas iniciativas nessa área foi o projeto Esfinge (http://esfinge.sf.net), que é um projeto guarda-chuva para a criação de diversos frameworks com essa abordagem baseada em metadados. Até o momento, já existem cinco frameworks disponíveis e vários artigos científicos em cima de inovações realizadas nesses projetos. No mesmo ano em que terminou seu doutorado, veio sua segunda filhinha, a Ana Beatriz.

Em 2012, Eduardo prestou concurso para o Instituto Nacional de Pesquisas Espaciais, onde assumiu o cargo de pesquisador no início 2013. Hoje ele segue com sua pesquisa na área de arquitetura, testes e design de software, buscando aplicar as técnicas que desenvolve para projetos na área científica e espacial. Adicionalmente, atua como docente na pós-graduação de Computação Aplicada desse instituto, onde busca passar o conhecimento que adquiriu e orientar alunos para contribuírem com essas áreas.

POR QUE UM LIVRO SOBRE REFLEXÃO E ANOTAÇÕES?

Quando descobri a API de reflexão na linguagem Java foi como se um novo mundo se abrisse para mim. Logo busquei aprender mais sobre o tema para conseguir aplicar esse conhecimento da melhor forma possível nos softwares que desenvolvia. Foi então que conheci o livro "Java Reflection in Action" dos autores Ira Forman e Nate Forman, publicado em 2004. A partir desse livro, conheci a base da API de reflexão e comecei a trabalhar nas minhas próprias soluções. Foi nessa época que desenvolvi o framework SwingBean (http://swingbean.sf.net), um framework para a geração de interfaces gráficas a partir das informações de classes.

Não demorou muito e foi lançada a JDK 1.5, que dava suporte nativo a anotações de código. Como utilizar aquele novo conceito? Só que, para responder essa pergunta, não esperei sair um livro sobre o assunto. Quando percebi que esse seria um tema que ainda tinha muito a ser explorado, ingressei no doutorado com o desafio de estudar boas práticas para componentes e frameworks que utilizam metadados. Desde essa época até hoje, foram diversos trabalhos com esse foco, não apenas buscando aplicações para utilização de metadados, mas também estudando práticas mais gerais na utilização dessa abordagem.

Desde então, nenhum livro saiu sobre o assunto. Nem no Brasil, nem no exterior. Isso me fez sentir com a obrigação de consolidar todo esse conhecimento que estava adquirindo e desenvolvendo nos últimos anos em um lugar só. Eu já vinha

publicando artigos mais voltados para a aplicação desses conceitos na indústria na revista MundoJ e para a comunidade acadêmica em conferências e periódicos científicos. Porém, ainda faltava consolidar todo esse conhecimento em um lugar só. Então veio a ideia desse livro!

O que você tem nas mãos agora é a consolidação do conhecimento que adquiri nos últimos anos sobre reflexão e metadados, costurado e desenvolvido com muito cuidado com o objetivo de tornar acessível a todos os desenvolvedores algo considerado por muitos como complicado. Houve também uma preocupação com a completude do livro em relação ao tema: não queria deixar nada importante de fora! Sendo assim, diferente do livro citado anteriormente que abordava somente a API de reflexão, esse livro, além da adição das anotações, também aborda outras questões, como técnicas de teste, uso de ferramentas de código aberto, boas práticas e manipulação de bytecode. Também houve a preocupação em levar ao leitor um conteúdo atualizado, apresentando as adições trazidas pelo Java 8 na área de reflexão e configuração de metadados.

O conhecimento sobre esse tema foi, e ainda é, um grande diferencial nos projetos de software que participo e participei. Com ele, é possível desenvolver soluções mais inteligentes, com potencial de tornar o software mais flexível e diminuir o tempo de desenvolvimento. Mas por que guardar esse conhecimento só para mim? Sendo assim, aí está! Pegue esse conhecimento e faça você agora a diferença!

LISTA DE DISCUSSÃO

A aplicação de reflexão e anotações para o desenvolvimento de frameworks e componentes abre novas possibilidades que permitem que se utilize a criatividade para o desenvolvimento de soluções mais reutilizáveis e flexíveis. Apesar de ter procurado expressar todo conhecimento que consegui reunir sobre esse assunto nos últimos anos nesse livro, certamente ainda há muito a ser discutido e muito a ser desenvolvido. A cada dia, nas pesquisas que realizo sobre o assunto, estou sempre buscando desenvolver novas soluções, abstrair padrões e aplicar esse conhecimento em diferentes domínios.

Sendo assim, criei uma lista de discussão com o nome **"Reflexão e Anotações"** no endereço reflexao-e-anotacoes@googlegroups.com. Mas você também pode discutir sobre este livro no Fórum da Casa do Código: http://forum.casadocodigo.com.br/.

Se você quer discutir, colocar suas dúvidas e saber eventos e novidades reflexão e anotações, deixo aqui o meu convite para a participação no grupo!

PREFÁCIO — OLHANDO PARA TRÁS SOBRE O OLHAR PARA DENTRO

As ideias-chave que fundamentam a reflexão estão por aí desde os primórdios da computação moderna. No entanto, não foi antes que a API para reflexão fosse publicada para Java, nos meados da década de 1990, que "reflexão" virou uma palavra "de casa", pelo menos naquelas casas afortunadas o suficientes para serem habitadas por desenvolvedores de software.

A ideia de que os dados é que direcionam a computação, assim como de que os dados a respeito dessa computação coabitam em um local de armazenamento comum, foi um elemento essencial na máquina universal de Turing. A ideia de que o programa que dirige a execução da unidade de controle do computador deveria ser armazenado na mesma memória compartilhada que os seus dados foi fundamental para o que seria conhecido como a arquitetura de Von Neumann.

Uma consequência de se ter um programa armazenado coabitando a mesma memória que seus dados, ou seja, fazendo que ele **seja** também um dado, permite que um programa possa ver e modificar qualquer coisa nesse armazenamento, até ele mesmo, caso seja necessário. Em outras palavras, um programa pode escrever outros programas, ou reescrever a si próprio.

Nós dizemos que um programa é **introspectivo** quando ele examina os dados que dirigem seu comportamento. E dizemos que ele é totalmente **reflexivo** quando essa autoexaminação influencia na forma como ele se comporta.

Uma explicação possível pela atual onda de interesse em reflexão, particularmente na comunidade orientada a objetos, é a sua herança na longa tradição de tentar fazer linguagens de programação serem mais abertas possível.

A definição de metacircular foi dada em Lisp por McCarthy em 1962. Essa definição permitiu a criação de um "modelo" para um programa que pode pegar outros programas em Lisp na forma de estruturas de dados em Lisp e executá-los. Esse programa, por sua vez, permitia que estrutura de dados em Lisp (*s-expressions*) fossem passadas para uma função (*eval*) e executadas como código.

Estava claro que McCarthy reconhecia que uma linguagem de programação poderia, pelo menos de forma tão efetiva quanto abordagens formais, definir como uma linguagem de programação funciona. McCarthy e seu grupo não viu a princípio o potencial prático que essa sua abordagem tinha para metaprogramação. Porém, a composição e execução de listas que representam programas rapidamente se tornou uma indispensável parte da mala de truques de um programador Lisp avançado.

Dados descrevem ou caracterizam algum aspecto de um domínio, quantitativamente ou qualitativamente. Metadados são dados também. Eles apenas são dados que descrevem outros dados. Objetos são construídos de dados, e código que opera esses dados. Os objetos que descrevem os objetos dos quais os programas são construídos são chamados de Metaobjetos.

A ideia de construir programas a partir de objetos apareceu primeiro em Smalltalk-76, e foi em seguida refinada em Smalltalk-80. O requisito de que o usuário deveria ser capaz de examinar ou alterar o máximo do sistema possível se encaixa perfeitamente com

nossas definições contemporâneas de reflexão. De fato, o extensivo conjunto de metaobjetos de Smalltalk-80 é um dos melhores exemplos existentes do poder desse tipo de objeto. Por exemplo, em Smalltalk-80, Classes, Processes, Methods, MethodDictionaries, CompiledMethods, Contexts, Blocks, e o próprio compilador eram todos criados a partir de objetos. Como resultado, Smalltalk possui um conjunto muito rico do que agora chamamos de **reflective facilities**.

Brian Cantwell Smith foi o primeiro a utilizar os termos **introspecção** e **reflexão** para descrever como programas podem tratar a si mesmos como "sujeitos" em 1983, em seu trabalho sobre torres reflexivas em 3-Lisp. Em 1987, Pattie Maes trouxe essas ideias de volta à comunidade orientada a objetos, ganhando uma audiência que aos poucos foi gerando mais repercussão. Mesmo assim, essas ideias foram tratadas como obscuras e impraticáveis por grande parte de comunidade que realizava pesquisa em programação. Smith recebeu de um revisor a crítica de que "ele tinha resolvido um problema que ninguém nunca tinha tido".

Isso foi até que, vinte anos atrás, quando o Java, no ápice do boom da internet, incorporou sua API de reflexão e um nível de bytecode, a descrição do arquivo de uma classe em seu código. Então, essas ideias começaram a ganhar uma certa tração.

Os metadados vêm em três diferentes sabores. O "Markup" que mistura os dados de fato com sua descrição. As anotações em Java são um exemplo onde as descrições do código são misturadas com o próprio código. HTML e XML são exemplos primordiais dessa abordagem, onde as descrições dos dados, como tags e atributos, são entrelaçadas com os dados que eles descrevem.

O "Manifesto" é um descrição ou um mapa separado dos dados que estão sendo descritos. Da mesma forma que um manifesto de carga que deve estar na parte de fora do container que está descrevendo. O manifesto é normalmente mantido a parte das instâncias de seus dados. Devido ao fato do manifesto ser dado e ser mutável, os próprios dados não precisam carregar a própria descrição.

Você tem os dados, é claro, e uma descrição separada que permite você, ou a um programa, ler, mapear e interpretá-la. Isso pode economizar um bom espeço quando comparado com as abordagens "embutidas", especialmente quando os dados que são mapeados são arrays de estruturas, o que, em essência, acaba sendo o caso na maioria das vezes.

Pense em um manifesto como um roteiro, ou uma Pedra de Roseta. Ele ajuda você, ou um programa, a compreender o que de outra forma seriam incompreensíveis zeros e uns.

Aliás, o próprio manifesto é muitas vezes escrito em uma linguagem ou estilo de notação diferente da informação que ele mapeia. Assim como as convenções utilizadas pelo estilo "markup", esses mapas podem ser escritos utilizando padrões e convenções diferentes do que o conteúdo que ele descreve. A vantagem é que permite que parsers de dados existentes possam trabalhar sem modificação quando os layouts dos dados são alterados. A desvantagem é que os manifestos podem se tornar obsoletos, necessitando que você encontre o manifesto correto de um determinada carga que ele descreve.

Por fim, os computáveis "Metaobjetos" proveem metadados em tempo de execução: Smalltalk, Java, C#, CLOS, e qualquer outra

API de reflexão os proveem. Oh, e o também chamado Document Object Model (DOM), visto nos modernos navegadores. A vantagem é que metaobjetos permitem que programas inspecionem, ou mesmo modifiquem a forma que se comportam on-the-fly. Eles criam uma ponte com uma linha de demarcação entre o código e os dados. As desvantagens: o desempenho e, claro, a verbosidade.

As APIs de reflexão em Java geraram uma onda de invocação dirigida pela reflexão que continua inabalada até hoje. JUnit, talvez o framework orientado a objetos de maior influência construído até hoje, mostrou as facilidades da reflexão em Java utilizando-as para localizar testes e as suítes de teste automaticamente, somente pelo framework, em tempo de execução, on-the-fly.

Plataformas como o Eclipse amarram vários plugins utilizando metadados e reflexão para colar o sistema em tempo de execução, permitindo que novas funcionalidades sejam introduzidas e incorporadas dinamicamente. Frameworks de injeção de dependência, como o Spring, dependem da reflexão para localizar e resolver dependências, ou fazer decisões de "casting" baseadas em descrições dinâmicas. De fato, esse estilo de integração do sistema em tempo de execução, de colar juntos plugins, componentes e classes on-the-fly, é um triunfo da reflexão!

Com o Java 8, a introdução de closures adicionam um vigor renovado a essa renascença dinâmica que a Lei de Moore e o poder da reflexão desencadearam. Essas funcionalidades também mostram o poder de escrever programas em bytecode, outra noção que surgiu a partir de nossa herança reflexiva.

Alguém pode argumentar que os metadados e a reflexão

floresceram como as velhas ideias da orientação a objetos, e estão diminuindo em importância. "Meta" venceu! "Objetos" perderam!

Viva la revolucion!

Por Brian Foote

Sumário

Versão: 20.9.6

Conceitos básicos

Essa primeira parte do livro vai ensinar os conceitos a respeito de reflexão e metadados, assim como do uso das APIs da linguagem Java para a utilização desses recursos. A partir da leitura dos quatro primeiros capítulos, espera-se que o leitor obtenha familiaridade com o uso de reflexão na linguagem Java, sabendo como obter e manipular seus principais elementos, utilizando-os para desenvolver componentes reutilizáveis em diferentes contextos.

O capítulo *Conhecendo a reflexão* fala sobre as deficiências da orientação a objetos e faz uma introdução à reflexão computacional, mostrando seu poder através de um pequeno exemplo. Já o capítulo *Java Reflection API* entra em detalhes a respeito da API Reflection da linguagem Java, mostrando seus principais elementos e como podem ser utilizados. Em seguida, o capítulo *Metadados e anotações* aborda a configuração de metadados adicionais, falando sobre as diferentes alternativas para defini-los e entrando em detalhes sobre como as anotações funcionam na linguagem Java. Finalizando essa primeira parte, o capítulo *Proxy dinâmico* fala sobre o recurso de proxies dinâmicos, quais as formas de implementá-los e como combiná-los com as técnicas mostradas nos capítulos anteriores. No final de cada capítulo é demonstrado um exemplo realista que aplica os conceitos apresentados.

Capítulo 1

CONHECENDO A REFLEXÃO

"O mundo é como um espelho que devolve a cada pessoa o reflexo de seus próprios pensamentos." - Luis Fernando Verissimo

Lembro-me até hoje quando estava começando a amadurecer nos conceitos da orientação a objetos. Um fato que me marcou muito foi quando pela primeira vez eu vi a necessidade de criar uma interface para encapsular um comportamento. Foi como se um novo mundo se abrisse na minha frente. Como diz a expressão em linguagem popular, "a ficha caiu". Eu finalmente compreendia como utilizar aquelas funcionalidades da orientação a objetos para implementar de uma forma inteligente os requisitos de um sistema. Com o estudo dos padrões de projeto, esse conhecimento foi cada vez mais se refinando, até ser concretizado no livro *Design Patterns com Java: Projeto orientado a objetos guiado por padrões* (GUERRA, 2012). Só que a minha jornada para desenvolver software de uma forma cada vez melhor não parou na orientação a objetos...

Com o tempo, eu fui vendo que apenas com a utilização da orientação a objetos muitos problemas ainda não eram resolvidos. Foi aí que conheci a reflexão e um outro novo mundo se abriu à

minha frente! A partir da reflexão é possível, em tempo de execução, conhecer a estrutura de uma classe e utilizar essas informações para a execução de uma lógica. Dessa forma, alguns tipos de código que dependiam de cada classe, mas que eram repetitivos em sua essência, agora podem ser criados de forma mais geral e reutilizados em diversos contextos.

A partir desse conhecimento de reflexão e das ferramentas para aplicar isso na linguagem Java, comecei a desenvolver soluções reutilizáveis no meu dia a dia, enxergando o potencial dessa técnica de aumentar a reutilização de código a partir da eliminação de tarefas repetitivas. De forma complementar, comecei a utilizar metadados, a princípio apenas com anotações de código, e em seguida combinando essas informações com fontes externas e padrões de nomenclatura. Como pouco estudo sobre esse tipo de solução havia sido conduzido até o momento, decidi assumir esse desafio em meus estudos de doutorado. Comecei então a estudar o código de diversos frameworks utilizados no mercado e, abstraindo suas soluções, consegui enxergar diversos padrões, tanto para sua estrutura interna, quanto para identificar cenários onde esse tipo de solução é aplicável.

Sendo assim, o objetivo desse livro é apresentar passo a passo todo o conhecimento que adquiri nessa jornada! Desde os primeiros passos no conhecimento da reflexão até técnicas avançadas para a utilização dos metadados. O que este livro possui diferente de outros que ensinam a utilizar reflexão em Java é que ele não se limita a mostrar como funcionam as APIs, mas também apresenta como elas podem ser utilizadas no projeto de um software para aumentar a reutilização e, consequentemente, a produtividade da equipe. Em outras palavras, será mostrado não só

como usar, mas como utilizá-la da forma correta e mais eficiente!

Este capítulo explora alguns problemas que a orientação a objetos não consegue resolver de forma direta, e introduz o conceito de reflexão, mostrando como ela funciona, especialmente na linguagem Java. Para que os leitores possam começar a saborear o poder da reflexão, alguns exemplos de código irão mostrar um pouco do que é possível fazer com reflexão. Em seguida, será mostrado que tudo tem um preço, e faremos uma análise comparativa de desempenho na invocação de um método via reflexão com sua chamada direta. Para finalizar, é feita uma apresentação de como o livro está organizado, fazendo uma prévia do que será tratado em cada capítulo.

1.1 AINDA FALTA ALGUMA COISA NA ORIENTAÇÃO A OBJETOS?

A orientação a objetos possui diversos recursos poderosos, que nos permitem modelar um sistema de forma a aumentar o reúso, tornando possível a criação de um código com melhor qualidade. A utilização de padrões potencializa ainda mais esses recursos, pois são soluções recorrentes que possuem uma estrutura adequada para um conjunto de cenários. Porém, mesmo assim, ainda existem cenários em que a orientação a objetos não ajuda muito.

Lendo parâmetros de aplicações web

Um exemplo comum desse tipo de código é a recuperação das informações de uma requisição web para serem inseridas em um outro objeto. O trecho de código a seguir ilustra essa situação com o código de dois servlets, sendo que um deles recebe informações

referentes a um produto e o outro referente a um cliente. Se observarmos os dois códigos, eles são bem parecidos, porém apenas utilizando as técnicas de orientação a objetos não é possível a reutilização.

Códigos de servlets para popular propriedades de objetos:

```
// código em um servlet que recupera informação de um produto

@WebServlet("/novoProduto")
public class NovoProdutoServlet extends HttpServlet{
    protected void doPost(HttpServletRequest rq,
                          HttpServletResponse rp)
            throws ServletException, IOException {
        Produto p = new Produto();
        p.setNome(rq.getParameter("nome"));
        p.setCategoria(rq.getParameter("categoria"));
        p.setPreco(Double.parseDouble(rq.getParameter("preco"));
        p.setDescricao(rq.getParameter("descricao"))
        //outras informações
    }
}

//código em um servlet que popula informações de um novo cliente

@WebServlet("/cadastro")
public class CadastroClienteServlet extends HttpServlet{
    protected void doPost(HttpServletRequest rq,
                          HttpServletResponse rp)
            throws ServletException, IOException {
        Cliente c = new Cliente()
        c.setNome(rq.getParameter("nome"));
        DateFormat formatadorData =
            new SimpleDateFormat("MM/dd/yy");
        c.setDataNascimento(
            (Date)formatter.parse(
                rq.getParameter("dataNascimento"));
        c.setLogin(rq.getParameter("login"));
        c.setSenha(rq.getParameter("senha"));
        //outras informações
    }
}
```

Um fato interessante de ser notado é que ambos os códigos seguem uma mesma lógica, ou seja, nos dois códigos parâmetros da requisição web são recuperados e inseridos no objeto em uma propriedade com o mesmo nome. Existem alguns casos especiais, como a conversão para `double` do preço do produto e a conversão para `Date` da data de nascimento do cliente. Mas, de qualquer forma, é fácil de notar a similaridade entre os códigos. Esse tipo de código normalmente é maçante de ser criado, principalmente quando o número de parâmetros a serem recuperados é grande. Como consequência, frequentemente são cometidos erros em que uma `String` acaba sendo escrita errada ou uma das propriedades acaba sendo esquecida pelo desenvolvedor.

Uma outra questão é que, se por acaso for adicionado mais um atributo em alguma das classes, isso vai acarretar em uma mudança de código no Servlet e um novo parâmetro vai precisar ser lido e atribuído a ele. Essa necessidade de mudança é um sintoma de que esse código poderia ser melhorado ou simplificado de alguma forma.

Criando proxies para executar métodos de forma assíncrona

Vamos agora considerar um exemplo bem diferente do anterior. Imagine que um sistema possua um serviço de logging, ou seja, para o registro de informações de auditoria, que tenha métodos para registrar informações e erros. Seu contrato é representado pela interface `LoggingService` . Nesse sistema, existem várias implementações desse serviço, que podem, por exemplo, armazenar as informações em banco de dados, em

arquivos etc. Devido a questões de desempenho, decidiu-se que esse serviço deveria executar de forma paralela à funcionalidade principal.

Seguindo um bom design orientado a objetos, e principalmente por existirem diversas implementações dessa interface, decidiu-se fazer um *proxy* que encapsula a classe original e executa seus métodos em uma nova *thread*. Veja adiante como seria o código dessa classe. Ela implementa a interface `LoggingService` e possui um atributo desse mesmo tipo, que irá conter o objeto encapsulado pelo *proxy*. Observe que, a cada chamada de método, o mesmo método do objeto encapsulado é invocado em uma thread diferente.

Proxy para executar as funções de logging de forma assíncrona:

```
public class LoggingAsyncProxy implements LoggingService{

    private final LoggingService service;

    public LoggingAsyncProxy(LoggingService service){
        this.service = service;
    }
    public void registerInformation(String info){
        new Thread(
            public void run(){
                service.registerInformation(info);
            }
        ).start();
    }
    public void registerError(Exception error){
        new Thread(
            public void run(){
                service.registerError(error);
            }
        ).start();
    }
```

```
}
```

Se observar o código do *proxy* criado, é possível perceber que existe uma certa duplicação nos métodos, porém essa é uma duplicação difícil de ser removida pois o método que é invocado é diferente. Imagine agora que seja necessário criar essa mesma funcionalidade para uma classe que possui uma interface diferente. Imagine, por exemplo, em uma classe do tipo DAO, *Data Access Object*, responsável por persistir e recuperar objetos, que se deseje executar uma *thread* diferente os métodos que não tenham retorno. O código resultante do código criado está mostrado a seguir.

Proxy para executar as funções de logging de forma assíncrona:

```
public class ProdutoDAOAsync implements ProdutoDAO{

    private final ProdutoDAO dao;

    public ProdutoDAOAsync(ProdutoDAO dao){
        this.dao = dao;
    }
    public void inserir(Produto p){
        new Thread(
            public void run(){
                dao.inserir(p);
            }
        ).start();
    }
    public Produto recuperar(int id){
        return dao.recuperar(id);
    }
    public void excluir(Produto p){
        new Thread(
            public void run(){
                dao.excluir(p);
            }
        ).start();
```

```
    }
    public void atualizar(Produto p){
        new Thread(
            public void run(){
                dao.atualizar(p);
            }
        ).start();
    }
}
```

Se compararmos, é possível observar que os dois códigos são muito parecidos, porém podemos também ver que é difícil de reaproveitar o código de um no outro caso. O motivo dessa dificuldade é que, por mais que a funcionalidade seja parecida, ela precisa invocar um método da classe que está sendo encapsulada no meio de sua execução. Dessa forma, é até possível reutilizar essa mesma classe de proxy para classes que implementem uma mesma interface. Mas para interfaces diferentes é mais complicado.

1.2 REFLEXÃO, MUITO PRAZER!

"Vá para o seu negócio, o prazer, enquanto eu vou para meu prazer, negócios." - William Wycherley

A noção de reflexão computacional foi introduzida pela primeira vez, ainda no contexto de linguagens procedurais, em 1982 (SMITH, 1982). Porém, foi em 1987 que o artigo *"Concepts and Experiments in Computational Reflection"* (MAES, 1987) consolidou o conceito. Reflexão pode ser definida como o processo pelo qual um programa de computador pode observar e modificar sua própria estrutura e comportamento. A utilização de reflexão também é conhecida como metaprogramação, pois com ela um programa pode realizar computações a respeito dele próprio.

A introspecção é um subconjunto da reflexão que permite um programa obter informações a respeito de si mesmo. A partir das informações obtidas com a introspecção, é possível manipular instâncias acessando seus atributos e invocando seus métodos. Dessa forma, pode-se criar um código que lida com uma classe cuja estrutura ele não conhece.

A reflexão também já foi documentada como um padrão no primeiro livro da série *"Pattern-Oriented Software Architecture"*, conhecido informalmente como POSA (ROHNERT; SOMMERLAD; STAL; BUSCHMANN; MEUNIER, 1996). Segundo essa fonte, um dos benefícios de sua utilização está na flexibilidade e adaptabilidade do código. Se um código que faz uso reflexão se baseia na estrutura de uma classe, por exemplo, ele se adaptará mais facilmente a mudanças nessa estrutura. Ainda segundo esse padrão, a piora no desempenho é uma das desvantagens, assim como um aumento na complexidade do código.

Todos os programadores sabem que, para executar um programa, um computador deve carregar seus comandos na memória. Adicionalmente, ele também utiliza a memória para armazenar variáveis que são manipuladas como parte da execução do software. Dentro desse contexto, a reflexão nada mais é do que o programa acessar e modificar as suas instruções. Esse é um recurso poderoso, mas que precisa ser utilizado com bastante responsabilidade.

Reflexão em Java

Em Java, as classes que implementam funcionalidades

relacionadas à reflexão ficam no pacote `java.lang.reflect` e são conhecidas como **API Reflection**. Apesar do nome, muitas dessas classes na verdade implementam apenas funções de introspecção. Em outras palavras, é possível obter informações sobre as classes de um software, mas não é possível modificá-las. Grande parte desse livro foca principalmente nessas funcionalidades.

A alteração de classes em Java não é suportada pela API padrão de reflexão, mas pode ser feita através da manipulação de bytecode. De uma forma mais simples, a aplicação modifica o código compilado que será carregado pela máquina virtual. Isso pode ser feito estaticamente depois da compilação do código, no momento do carregamento da classe ou em tempo de execução. Como é complicado substituir na máquina virtual uma classe que já foi carregada, quando a modificação é feita em tempo de execução, normalmente é carregada uma nova classe com as modificações feitas na existente. O penúltimo capítulo deste livro é dedicado à técnica de manipulação de bytecode, porém em capítulos anteriores são apresentadas algumas ferramentas que utilizam esse recurso de uma forma encapsulada.

Diferença em relação a linguagens dinâmicas

O termo **linguagem dinâmica** é utilizado para descrever linguagens de alto nível que executam em tempo de execução comportamentos que as linguagens normalmente têm durante a compilação. Dentre esses comportamentos, normalmente está a adição de código em objetos e a modificação de tipos. Nessa linguagem, esse tipo de operação é comum e muitas vezes é suportado a partir de recursos e ferramentas da própria linguagem. Exemplos desse tipo de linguagem são Ruby, Python e Groovy.

A utilização de reflexão em linguagens dinâmicas acaba sendo mais simples do que em linguagens estaticamente tipadas, como Java. Em alguns casos, a invocação dinâmica de métodos e a modificação de tipos são utilizadas como mecanismo padrão da linguagem, e não apenas em situações em que se deseja uma maior flexibilidade. As técnicas mostradas neste livro focam em como utilizar com segurança os recursos da reflexão em Java, que é uma linguagem estaticamente tipada. Dessa forma, a aplicação pode usufruir de recursos que permitem uma maior segurança de código e aplicar a reflexão somente em pontos onde uma maior flexibilidade é necessária. Por mais que o uso desses recursos seja diferente em linguagens dinâmicas, acredito que alguns conceitos aqui apresentados também podem ser utilizados nesse contexto.

Como a reflexão pode me ajudar?

Reflexão é muito útil em situações em que o código precisa lidar com classes com interfaces diferentes, porém sua criação é repetitiva e segue uma lógica. Para esse tipo de código acaba sendo difícil a utilização de técnicas tradicionais da orientação a objetos, pois os objetos possuem contratos diferentes e os métodos que precisam ser invocados dependem dos objetos. Os exemplos apresentados no começo deste capítulo ilustram dois cenários onde isso ocorre.

A ideia do uso de reflexão é tornar o código adaptável à estrutura dos objetos e com isso permitir que ele seja mais reutilizável. Dessa forma, é possível substituir a criação de um código que precisaria ser repetido para cada classe, pela chamada a um método ou uma classe que utilize reflexão. Dessa forma, pode-se diminuir a quantidade de código, aumentando a velocidade de desenvolvimento da equipe.

Por mais que a utilização de reflexão seja uma técnica mais complexa, seu uso pode ser feito de forma bem localizada. Em outras palavras, é possível encapsular sua utilização dentro de classes ou componentes, de forma que isso fique transparente para o resto do software. Um ótimo exemplo disso são os frameworks, pois grande parte deles utiliza esses recursos de reflexão de forma encapsulada e isso não precisa ser conhecido pelo desenvolvedor.

1.3 O PRIMEIRO CONTATO COM A API REFLECTION

Como também sou programador, sei que vários leitores devem

estar ansiosos para ver um pouco de código com a reflexão sendo utilizada na prática. Esta seção apresenta uma pequena prévia sobre o funcionamento da API de reflexão em Java, mostrando um pequeno exemplo que ilustra a sua utilização. Aqui também falaremos das anotações e como podem ser usadas em conjunto com a reflexão. Peço para que nesse momento se preocupem apenas em entender os conceitos e o exemplo, pois maiores detalhes sobre essas APIs serão dados nos próximos capítulos.

Transformando as propriedades de uma classe em mapas

O exemplo que vamos explorar neste primeiro capítulo envolve a geração de um mapa de propriedades a partir de uma classe no formato Java Bean, ou seja, com métodos de acesso nos formatos GET e SET. Ao criar um algoritmo utilizando reflexão, a ideia é ver qual seria o procedimento que um desenvolvedor utilizaria para escrever o código manualmente e tentar reproduzi-lo no código. No caso da geração do mapa, o procedimento é basicamente invocar cada método getter da classe e adicionar no mapa o valor retornado com a chave do nome da propriedade.

Esse exemplo, de certa forma, é parecido com o que vimos anteriormente para a recuperação dos parâmetros do Servlet. No exemplo anterior, precisávamos recuperar os parâmetros passados para página de forma a popular as propriedades do objeto; aqui iremos recuperar as propriedades de um objeto para inserirmos em um mapa. Em ambos os casos, se fosse desenvolvido um código para cada classe, esse código seria repetitivo e chato de ser criado. O objetivo é automatizar essa tarefa, de forma que esse tipo de código não precise ser criado, diminuindo a chance de erros e

aumentando a produtividade dos desenvolvedores.

A classe apresentada a seguir mostra uma possível implementação desse código para gerar o mapa de propriedades. O método `gerarMapa()` percorre todos os métodos do objeto passado como parâmetro, recuperando como retorno somente os métodos identificados como getters e inserindo no mapa. Para poder acessar os dados de uma classe, o primeiro passo é recuperar a instância da classe `Class` relativa à classe do objeto, o que é feito através do método `getClass()` . A partir dessa instância, os métodos são recuperados através de `getMethods()` e essa lista de instâncias de `Method` é percorrida procurando os que podem ser classificados como getter.

O método `isGetter()` acessa as informações do método verificando se se trata de um método getter. As informações consideradas para isso são: se nome do método se inicia com "get", se o retorno é diferente de `void` e se ele não possui parâmetros. Caso o método seja identificado como getter, ele é invocado a partir do método `invoke()` e seu valor recuperado para ser inserido no mapa. Observe que o nome do método é transformado no nome da propriedade pelo método `deGetterParaPropriedade()` para adição no mapa. Outra questão interessante de ser notada é que a invocação do método via reflexão está envolta por um bloco `try/catch` , o que indica que existem erros que podem ocorrer nessa invocação (o que será abordado mais à frente neste livro).

Classe que utiliza reflexão para a geração de um mapa:

```
public class GeradorMapa {
    public static Map<String, Object> gerarMapa(Object o){
        Class<?> classe = o.getClass();
```

```
        Map<String, Object> mapa = new HashMap<>();
        for(Method m: classe.getMethods()){
            try {
                if(isGetter(m)){
                   String propriedade = deGetterParaPropriedade(
                              m.getName());
                   Object valor = m.invoke(o);
                   mapa.put(propriedade, valor);
                }
            } catch (Exception e) {
                throw new RuntimeException(
                    "Problema ao gerar o mapa",e);
            }
        }
        return mapa;
    }
    private static boolean isGetter(Method m) {
        return m.getName().startsWith("get") &&
            m.getReturnType() != void.class &&
            m.getParameterTypes().length == 0;
    }
    private static String
    deGetterParaPropriedade(String nomeGetter){
        StringBuffer retorno = new StringBuffer();
        retorno.append(nomeGetter.substring(3, 4)
                                  .toLowerCase());
        retorno.append(nomeGetter.substring(4));
        return retorno.toString();
    }
}
```

A próxima listagem mostra a utilização desse método criado para a geração de mapas. A classe `Produto`, a mesma do exemplo do Servlet, possui quatro propriedades com métodos get e set para sua respectiva recuperação e modificação. Em seguida, é apresentado um código que cria uma nova instância dessa classe, recupera o mapa com o método criado e imprime as propriedades recuperadas no console. Sendo assim, vê-se que com reflexão foi possível criar uma rotina que acessa métodos de uma classe sem depender diretamente dela.

Utilização do método de geração do mapa de propriedades:

```
// Classe para ser utilizada como base
public class Produto {

    private String nome;
    private String categoria;
    private Double preco;
    private String descricao;

    public Produto(String nome, String categoria, Double preco,
                   String descricao) {
        this.nome = nome;
        this.categoria = categoria;
        this.preco = preco;
        this.descricao = descricao;
    }
    //metodos get e set omitidos
}

//Código que executa o método de geração do mapa e imprime
public static void main(String[] args){

    Produto p = new Produto("Design Patterns","LIVRO",59.90,
            "Publicado pela Casa do Codigo");
    Map<String,Object> props = GeradorMapa.gerarMapa(p);
    for(String prop : props.keySet()){
        System.out.println(prop+" = "+props.get(prop));
    }
}
```

Conhecendo anotações

Quando você implementa um método para cada classe, por mais que ele seja repetitivo, tem-se a liberdade de realizar alterações para atender necessidades de negócio. Tomando como exemplo a geração do mapa, poderia ser necessário que alguma propriedade fosse ignorada ou que fosse adicionada com um nome diferente. Ao criar cada método individual isso é fácil de fazer, porém, utilizando reflexão, como fazer para que o algoritmo trate

algumas propriedades diferente? Apesar de a reflexão permitir a definição de um método genérico para a geração de mapas, ele não atende aos requisitos quando algum elemento precisa ser tratado de forma diferente.

É nesse ponto que entram as anotações! Elas permitem marcar os elementos de forma que um algoritmo que utilize reflexão possa identificar os elementos que ele deve tratar de forma diferente. As anotações permitem adicionar novas informações em elementos de programação. Essas informações são chamadas de específicas de domínio, pois elas dizem respeito ao interesse da classe que irá consumi-las. É importante já deixar claro que uma anotação não possui comportamento e somente sua presença não faz nada além de adicionar uma informação, porém elas podem ser acessadas por outras classes e componentes para permitir que eles adaptem seu comportamento de acordo com sua presença.

As duas listagens a seguir mostram como as anotações poderiam ser criadas, sendo que a primeira é para ignorar uma propriedade e a segunda para definir um nome diferente para uma propriedade. Novamente peço para os leitores não se preocuparem muito com os detalhes de implementação, que serão explicados nos próximos capítulos. O que é importante perceber nessas anotações é que a primeira não possui propriedades e a segunda possui uma propriedade do tipo `String` chamada `value` .

Anotação que define quando uma propriedade precisa ser ignorada:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignorar {
}
```

Anotação que define um nome diferente para a propriedade:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface NomePropriedade {
    String value();
}
```

Como já foi dito anteriormente, nada adianta criar anotações sem um código que as consome, pois sozinhas elas não fazem nada. Dessa forma, a próxima listagem mostra como o código do gerador de mapas poderia ser modificado para considerar a presença das anotações. No método `isGetter()` foi acrescentada uma condição que verifica se a anotação `@Ignorar` está presente, fazendo com que os métodos getters com essa anotação não sejam considerados. Adicionalmente, no método principal `gerarMapa()`, antes de chamar o método `deGetterParaPropriedade()`, é verificado se a anotação `@NomePropriedade` está presente e, em caso positivo, o valor configurado é recuperado e utilizado.

Modificação da classe que gera o mapa para considerar as anotações:

```
public static Map<String, Object> gerarMapa(Object o){
    Class<?> classe = o.getClass();
    Map<String, Object> mapa = new HashMap<>();
    for(Method m: classe.getMethods()){
        try {
            if(isGetter(m)){
                String propriedade = null;
                if(m.isAnnotationPresent(NomePropriedade.class))
                {
                    propriedade =
                        m.getAnnotation(
                            NomePropriedade.class).value();
                }else{
                    propriedade =
                        deGetterParaPropriedade(m.getName());
```

```
                }
                Object valor = m.invoke(o);
                mapa.put(propriedade, valor);
            }
        } catch (Exception e) {
            throw new RuntimeException(
                "Problema ao gerar o mapa",e);
        }
    }
    return mapa;
}
private static boolean isGetter(Method m) {
    return m.getName().startsWith("get") &&
        m.getReturnType() != void.class &&
        m.getParameterTypes().length == 0 &&
        !m.isAnnotationPresent(Ignorar.class);
}
```

A partir dessa nova versão, é possível, por meio da adição de anotações, alterar parte do comportamento do algoritmo reflexivo, fazendo-o ignorar uma propriedade ou mudando o nome que é adicionado no mapa. A seguir está um exemplo de como alguns dos métodos da classe `Telefone` seriam anotados para que o código do gerador de mapas conseguisse capturar as anotações. Mas esse exemplo foi só para dar um gostinho do que virá nos próximos capítulos do livro, onde será mostrado não só como utilizar reflexão e anotações, mas também em que situações e quais técnicas podem ser utilizadas.

Exemplo do uso das anotações pela classe Telefone:

```
public class Telefone {

    //atributos e outros métodos omitidos

    @NomePropriedade("codigoInternacional")
    public String getCodigoPais() {
        return codigoPais;
    }
```

```
    @Ignorar
    public String getOperadora() {
        return operadora;
    }
}
```

1.4 USAR REFLEXÃO TEM UM PREÇO?

"There's no such thing as a free lunch. (Não existe esse negócio de almoço grátis.)" - Expressão popular que expressa a ideia de que é impossível conseguir algo sem dar nada em troca

Um dos principais preços que se paga ao se utilizar reflexão é o desempenho. Invocar um método a partir de reflexão é mais demorado do que invocar um método diretamente no objeto. Isso é um fato! Porém, somente essa informação não é suficiente para avaliar se isso pode gerar um problema de desempenho. Para termos noção do prejuízo ao desempenho trazido pela reflexão, esta seção apresenta um teste que compara o tempo de uma invocação de método feita diretamente com a API de reflexão.

Teste para medida de desempenho

Para realizar esse teste, foi criada uma classe chamada `ClasseTeste` que possui um método chamado `metodoVazio()`. Como o próprio nome sugere, seu conteúdo é vazio para que o tempo medido seja apenas relativo à sua invocação. Em seguida, foi criada uma interface chamada `InvocadorMetodo`, que será o contrato de classes que irão invocá-lo repetidas vezes utilizando diferentes abordagens. O método `invocarMetodo()` recebe como parâmetro o número de vezes que o método precisa ser invocado e o chama repetidas vezes.

A primeira implementação dessa interface está representada na listagem a seguir. A classe `InvocadorNormal` faz a invocação do método diretamente através de uma instância da classe. Essa implementação servirá como base para compararmos com a invocação do método utilizando reflexão.

Implementação que invoca o método diretamente:

```
public class InvocadorNormal implements InvocadorMetodo {
    public void invocarMetodo(int vezes) {
        ClasseTeste ct = new ClasseTeste();
        for(int i=0; i<vezes; i++){
            ct.metodoVazio();
        }
    }
}
```

As duas listagens a seguir, `InvocadorReflexao` e `InvocadorReflexaoCache`, utilizam reflexão para invocar o método. Em ambas, a classe do objeto é recuperada através de `getClass()` e o método pelo seu nome através de uma chamada a `getMethod()`. Em seguida, ele é invocado na instância através de uma chamada de `invoke()`. Observe que o objeto no qual ocorre a invocação é passado como argumento. A diferença entre essas duas implementações é que, na primeira, a representação do método é recuperada a cada iteração e, na segunda, ela é recuperada uma vez e reaproveitada.

Implementação que a cada iteração busca o método e o invoca utilizando reflexão:

```
public class InvocadorReflexao implements InvocadorMetodo {
    public void invocarMetodo(int vezes) {
        try {
            ClasseTeste ct = new ClasseTeste();
            for(int i=0; i<vezes; i++){
                Method m =
```

```
                    ct.getClass().getMethod("metodoVazio");
                m.invoke(ct);
            }
        } catch (Exception e) {
            throw
                new RuntimeException(
                    "Não consegui invocar o método",e);
        }
    }
}
```

Implementação que busca o método uma vez e o invoca utilizando reflexão a cada iteração:

```
public class InvocadorReflexaoCache implements InvocadorMetodo {
    public void invocarMetodo(int vezes) {
        try {
            ClasseTeste ct = new ClasseTeste();
            Method m = ct.getClass().getMethod("metodoVazio");
            for(int i=0; i<vezes; i++){
                m.invoke(ct);
            }
        } catch (Exception e) {
            throw
                new RuntimeException(
                    "Não consegui invocar o método",e);
        }
    }
}
```

Para comparar as formas de invocação das implementações de `InvocadorMetodo`, foi criada a classe `TesteDesempenho` apresentada a seguir. O método estático `executaTeste()` recebe como parâmetro uma instância de `InvocadorMetodo` para realizar o teste. Ele chama o método `invocarMetodo()` para que o método seja invocado 100.000 vezes, e mede o tempo que ele leva para executar em nanosegundos. Além de imprimir esse resultado no console, ele retorna o tempo medido.

Classe que compara as formas de invocação:

```
public class TesteDesempenho {
    public static void main(String[] args){
        double normal = executaTeste(new InvocadorNormal());
        double reflection =
            executaTeste(new InvocadorReflexao());
        System.out.println(
            (reflection/normal) + " vezes mais que o normal"
        );
        double reflectionCache = executaTeste(
            new InvocadorReflexaoCache());
        System.out.println(
            (reflectionCache/normal) +
                " vezes mais que o normal"
        );

    }
    public static double executaTeste(InvocadorMetodo invoc){
        long millis = System.nanoTime();
        invoc.invocarMetodo(100000);
        String nomeClasse = invoc.getClass().getName();
        long diferenca = System.nanoTime() - millis;
        System.out.println("A classe "+nomeClasse+
            " demorou " + diferenca + " nano segundos");
        return diferenca;
    }
}
```

O método principal dessa classe invoca o método para execução do teste para cada uma das implementações de `InvocadorMetodo` . Adicionalmente, ele faz a divisão do tempo que cada implementação que utilizou reflexão pelo tempo de referência da implementação que invocou o método diretamente. Dessa forma, conseguimos ver quantas vezes mais demora a invocação com reflexão em cada caso. O resultado que foi impresso no console quando executei o teste em minha máquina está apresentado na listagem a seguir.

Resultado da execução da comparação de desempenho:

```
A classe InvocadorNormal demorou 2334000 nano segundos
```

```
A classe InvocadorReflexao demorou 90823000 nano segundos
38.913024850042845 vezes mais que o normal
A classe InvocadorReflexaoCache demorou 6934000 nano segundos
2.9708654670094257 vezes mais que o normal
```

Os resultados em nanosegundos acabam sendo um pouco confusos pois são números muito grandes. Andando seis casas decimais à esquerda temos o resultado em milissegundos, que foram aproximadamente 2 ms, 90 ms e 7 ms para as respectivas classes. Porém, esse resultado pode variar de acordo com a máquina e não é tão interessante quanto à proporção entre esses valores. Pelo que foi impresso no console, podemos observar que, quando a recuperação do método é incluída em cada iteração, o tempo é cerca de 39 vezes maior. Porém, quando o método é recuperado apenas uma vez, essa diferença cai significativamente, sendo aproximadamente apenas 3 vezes maior.

Reflexões sobre a perda de desempenho com reflexão

"Escreva seu código da forma mais simples, mais fácil de manter e mais eficiente possível. Só depois faça a medição de seu desempenho, encontre onde estão os problemas e comece a introduzir otimizações." - Danny Simmons

Incluí a discussão sobre desempenho logo no primeiro capítulo do livro pois, se você vai utilizar reflexão em alguma solução, você precisa saber o preço que vai estar pagando. O fato de uma invocação de método com reflexão demorar cerca de três vezes mais do que uma invocação direta do método, isso não significa que o tempo de execução de um algoritmo irá triplicar se utilizar reflexão. É preciso avaliar cada caso e levar em consideração o impacto que isso terá no resultado final.

Se observarmos o valor bruto obtido com a invocação de um método por reflexão, podemos ver que o valor de uma única invocação é na ordem de nanosegundos. Isso é um valor bem pequeno se compararmos com o tempo para execução de tarefas de entrada e saída, como leitura de um arquivo, acesso a banco de dados e envio de dados pela rede. Em outras palavras, em relação ao tempo total de execução de uma funcionalidade que envolver funções de entrada e saída, a perda com o uso de reflexão pode ser desprezível.

Essa diferença no desempenho pode se tornar significativa quando uma operação realizada com reflexão é executada dentro de um loop repetidas vezes. Nesse caso, a pequena diferença multiplicada por milhares de vezes pode se transformar em um gargalo significativo. Porém, mesmo nesses casos, muitas vezes é possível fazer otimizações. Pelo exemplo apresentado, ficou claro que a recuperação da representação do método leva mais tempo do que sua execução. No caso de uma invocação repetida de métodos por reflexão, uma otimização seria armazenar previamente em uma variável os métodos que seriam invocados.

Tomando como exemplo a geração do mapa de propriedades, imagine que em uma aplicação essa transformação precise ser feita em uma lista com milhares de objetos. Uma possível otimização seria, em vez de buscar os métodos getters da classe durante a criação do mapa, fazer isso previamente e reutilizar para todos os objetos. Mesmo que na lista houvesse objetos de diferentes classes, essas informações poderiam ser armazenadas por classes e reaproveitadas para todas as suas instâncias.

Um exemplo desse tipo de otimização de desempenho foi feito

com o gerador de mapas na próxima listagem. Em vez de receber o objeto, recuperar sua classe e executar o processamento, essa nova implementação recebe a classe no construtor e já cria um mapa com as propriedades e os métodos para serem utilizados posteriormente. Dessa forma, ao executar o gerador de mapas ele irá verificar inicialmente se o objeto passado realmente é daquela classe e depois irá utilizar os métodos armazenados no mapa. Isso evitará o overhead da obtenção dos métodos, tornando a geração do mapa mais eficiente se executado para diversos objetos da mesma classe.

Gerador de mapas que faz cache dos métodos:

```
public class GeradorMapaPerformance {

    private Map<String, Method> propriedades = new HashMap<>();
    private Class<?> classe;

    public GeradorMapaPerformance(Class<?> classe) {
        this.classe = classe;
        for (Method m : classe.getMethods()) {
            if (isGetter(m)) {
                String propriedade = null;
                if (m.isAnnotationPresent(
                    NomePropriedade.class)) {
                    propriedade =
                        m.getAnnotation(NomePropriedade.class)
                         .value();
                } else {
                    propriedade =
                        deGetterParaPropriedade(m.getName());
                }
                propriedades.put(propriedade, m);
            }
        }
    }
    public Map<String, Object> gerarMapa(Object o) {
        if (!classe.isInstance(o)) {
            throw new RuntimeException(
                "O objeto não é da classe" + classe.getName());
```

```
        }
        Map<String, Object> mapa = new HashMap<>();
        for (String propriedade : propriedades.keySet()) {
            try {
                Method m = propriedades.get(propriedade);
                Object valor = m.invoke(o);
                mapa.put(propriedade, valor);
            } catch (Exception e) {
                throw
                    new RuntimeException(
                        "Problema ao gerar o mapa", e);
            }
        }
        return mapa;
    }
}
```

Como a frase no início desta seção sugere, inicialmente deve-se desenvolver o código da forma mais limpa e simples possível. Em seguida, caso os requisitos de desempenho não estejam sendo atendidos, deve-se procurar onde está o gargalo e realizar a otimização daquela parte do código. A otimização de desempenho precoce é uma má prática, pois pode focar esforços na otimização de uma parte do código que não é significativa no panorama geral da aplicação. Adicionalmente, essa otimização pode ter efeitos colaterais negativos no código, como prejudicar a sua legibilidade ou sua flexibilidade.

Esse livro irá focar no uso de técnicas de reflexão para a elaboração de soluções flexíveis e inteligentes, com potencial de simplificar a criação de aplicações e aumentar a produtividade da equipe. O foco não será na perda ou ganho de desempenho trazido por cada uma das técnicas, a não ser que isso seja um fator relevante no tópico que estiver sendo apresentado. Porém, sugere-se que, quando a reflexão for utilizada em uma solução, os requisitos de desempenho sejam verificados e que técnicas de

otimização sejam utilizadas quando necessário.

1.5 CONSIDERAÇÕES FINAIS

Esse primeiro capítulo teve o objetivo de apresentar o conceito de reflexão computacional e utilização de metadados, mostrando cenários em que as técnicas de orientação a objetos não são suficientes para criar uma solução reutilizável e inteligente. Para os que não se contentam se não verem um pouco de código, foi apresentado um exemplo simples de como a reflexão e o uso de metadados podem auxiliar na criação de uma solução. Para os que olham para reflexão como se fosse um bicho de sete cabeças, peço que não se assustem com o código, pois tudo que foi utilizado será explicado com bastante detalhe nos próximos capítulos. Se desejarem, poderão revisitar esse exemplo mais tarde depois de lerem os dois próximos capítulos.

Por fim, este capítulo falou sobre uma questão delicada na utilização de reflexão, que é o seu *trade-off* em relação ao desempenho. Como foi mostrado, a flexibilidade do uso dessa técnica tem seu custo em termos de desempenho, e é por isso que é importante ter ciência disso para utilizá-la de forma adequada. Um outro efeito do uso da reflexão é o aumento da complexidade do código, porém isso é algo que pode ser facilmente controlado encapsulando seu uso em classes e componentes que são utilizados pelo resto do código. Afinal, quantos frameworks que utilizavam reflexão você já usou sem nem saber como ele estava fazendo aquelas coisas?

Capítulo 2

JAVA REFLECTION API

"Diga espelho meu se há na avenida alguém mais feliz que eu" - "É Hoje", Didi e Maestrinho

Reflexão é considerado um tema avançado, pois trabalha com metaprogramação, ou seja, é um código que trabalha com o próprio código. Por esse motivo, é evitado e temido por muitos programadores. Se por um lado identificar as situações adequadas para o uso dessa técnica realmente não é fácil e exige uma certa experiência, a API Reflection do Java para manipular essas estruturas não é difícil.

Por exemplo, imagine que você esteja começando a trabalhar com uma API para manipular objetos em 3 dimensões. Certamente essa API exigirá uma certa curva de aprendizado até que se obtenha proficiência nela. Porém, a dificuldade não é saber quais são as classes e quais métodos chamar, mas sim compreender os conceitos que ela representa para que se saiba como utilizá-la de forma adequada. Se você já trabalhou com objetos em 3 dimensões com outra ferramenta que utilizava os mesmo conceitos, certamente essa curva de aprendizado será muito mais suave. A maior dificuldade em se aprender uma API está na compreensão dos conceitos que ela representa e não em saber quais são seus métodos e classes.

A API Reflection da linguagem Java trabalha com conceitos familiares a qualquer programador, que são os próprios elementos da linguagem. Uma classe, por exemplo, possui informações como sua superclasse, suas interfaces, seu nome, seu pacote, seus métodos e seus atributos. Você pode fazer com ela o que se faz com uma classe, instanciar. Da mesma forma, um método possui retorno, nome, parâmetro, modificadores e exceções, e o que se pode fazer com ele é invocar. Por isso, ao começar a trabalhar com a API Reflection, qualquer programador se sente em casa, trabalhando com conceitos que já são familiares.

Este capítulo tem o objetivo de apresentar as principais funcionalidades da API Reflection. Não se tem a intenção de ser completo e cobrir cada método e cada classe, porém a ideia é mostrar as principais classes com suas respectivas funcionalidades, através de exemplos como podem ser utilizadas. Ao fim do capítulo, exponho um exemplo mais completo que utiliza diversas funcionalidades.

2.1 OBTENDO INFORMAÇÕES SOBRE AS CLASSES

"Informação não é conhecimento." - Albert Einstein

Quando se começa a desenvolver um software em Java, o ponto inicial é sempre uma classe. Para trabalhar com reflexão não é diferente, pois a classe principal, que é o ponto de partida para obtermos informações sobre os elementos de um programa, é a `java.lang.Class` — que representa justamente uma classe. Sendo assim, o primeiro passo para começar a trabalhar com reflexão é obtermos a instância de `Class` da classe com a qual

queremos trabalhar.

Para entendermos como isso funciona, considere um objeto qualquer de uma aplicação. Esse objeto é descrito pela sua classe, que determina quais são os seus métodos e seus atributos, e por isso podemos dizer que ela possui as metainformações a respeito de seus objetos. Porém, que tipo de coisa essa classe pode ter? Se ela descreve o objeto, quem a descreve?

Aí é que entra a instância da classe `Class` que possui todas as informações de uma classe. Essa instância possui as metainformações, não do objeto, mas da classe, como quais são seus atributos, quais são os seus métodos e qual é sua superclasse e suas interfaces. E daí surge mais uma dúvida: quem descreve a instância de `Class` ? Como vimos, os objetos são descritos por suas classes, então essa instância é descrita por `Class` . E quem descreve `Class` ? Uma instância da própria classe `Class` !

Eu sei que isso tudo pode causar um grande nó na nossa cabeça, e a figura seguinte mostra de forma esquemática o que foi descrito nos parágrafos anteriores. O que é importante entender aqui é que, quando trabalhamos com reflexão, estamos trabalhando um nível acima do que costumamos trabalhar. Enquanto normalmente trabalhamos com objetos que são descritos por classes, com reflexão trabalhamos com classes cujas instâncias descrevem as nossas classes.

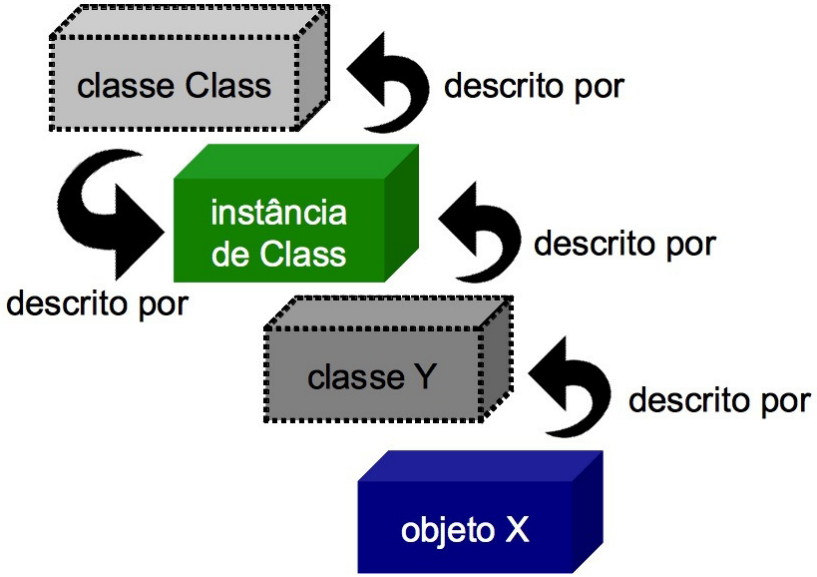

Figura 2.1: Quem possui as metainformações de quem

Utilizando referências estáticas

A forma mais direta de obtermos uma instância de `Class` é através de uma referência estática da classe. Isso é utilizado quando sabemos em tempo de compilação qual classe precisa ser referenciada. Para que isso seja feito, usamos o nome da classe seguido por `.class`, o que retorna a instância de `Class` respectiva.

O código apresentado a seguir mostra como as referências estáticas a classes podem ser utilizadas. Inicialmente é criada uma instância de `Class` que recebe a instância referente à classe `String`, cujo nome é impresso no console. Pode não parecer muito útil utilizar referências estáticas da classe, pois, se já sei com qual vou trabalhar, para que preciso utilizar reflexão? Porém, essas

referências estáticas são muito utilizadas quando precisamos passar uma instância de `Class` como parâmetro para um método, como também é apresentado na listagem.

Utilizando referências estáticas de Class:

```
public class ReferenciaEstaticaClasse {

    public static void main(String[] args) {
        Class<String> classe = String.class;
        System.out.println(classe.getName());
        imprimeNomeClasse(Integer.class);
    }
    public static void imprimeNomeClasse(Class<?> classe){
        System.out.println("Chamado o método com " +
                           classe.getName());
    }
}
```

O que talvez tenha chamado a atenção no exemplo é o fato de que a classe `Class` possui um tipo genérico. Esse tipo genérico é relativo à classe que está sendo representada. Dessa forma, como será visto posteriormente neste capítulo, isso permite a inferência de retorno de alguns métodos, o que evita que sejam feitos casts desnecessários.

No caso de não sabermos qual é o tipo, podemos sempre utilizar o wildcard `?` para dizer que não sabemos o tipo de classe que será passado. Um exemplo é o método `imprimeNomeClass()` que recebe como parâmetro o tipo `Class<?>` , significando que qualquer tipo pode ser passado como parâmetro. O uso de tipos genéricos também possibilita que seja feita uma certa restrição no tipo de classe que pode ser passada como parâmetro. Por exemplo, se definirmos o tipo do parâmetro como `Class<? extends Serializable>` , aceitamos como parâmetro somente instâncias de `Class` de tipos que implementam a interface `Serializable` .

Apesar de tipos primitivos em Java não serem considerados classes, é possível obter uma representação de `Class` para eles. Para isso, a referência estática desses tipos pode ser utilizada, como por exemplo `int.class` e `char.class`. Até mesmo o `void` possui uma representação de `Class`, sendo obtido através da expressão `void.class`. Na seção *O primeiro contato com a API Reflection*, onde fizemos a geração do mapa, essa referência foi utilizada no método `isGetter()` para verificar se ele possuía retorno.

Recuperando a classe de um objeto

Uma outra forma muito comum de se obter uma instância de `Class` é através de um objeto dessa classe. A classe `Object` define o método `getClass()` que retorna a representação da classe daquele objeto. A recuperação da `Class` dessa forma é muito comum quando se recebe o objeto como parâmetro e desejam-se informações sobre sua classe. Um exemplo disso foi mostrado no código da geração do mapa, que recebia como parâmetro um `Object` e utilizava o método `getClass()` para obter a sua classe.

Normalmente, essa forma de recuperar a classe é utilizada em métodos mais gerais que recebem um `Object` como parâmetro e fazem uso de reflexão para conhecer mais sobre a classe e saber quais métodos invocar ou quais atributos acessar. Isso possibilita que objetos de qualquer classe possam ser passados como parâmetro e utilizados pela lógica do método.

A listagem a seguir mostra um exemplo da recuperação da classe a partir do objeto. O tipo genérico retornado pelo método

`getClass()` será sempre `Class<? extends Tipo>`, onde "Tipo" representa o tipo da variável na qual o método está sendo invocado. Isso faz sentido, pois uma variável de um determinado tipo pode armazenar objetos de um subtipo. No exemplo, o objeto é do tipo `Integer`, mas está sendo atribuído a uma variável do tipo `Number`. Sendo assim, apesar do tipo retornado pelo `getClass()` ser `Class<? extends Number>`, ao imprimir o nome do objeto `Class` obtido, será impresso no console "`java.lang.Integer`".

Recuperando as classes pelos objetos:

```
public class RecuperandoPeloObjeto {

    public static void main(String[] args) {
        Number object = new Integer(100);
        Class<? extends Number> c = object.getClass();
        System.out.println(c.getName());
        //...
    }
}
```

Uma String com o nome da classe

A última forma de se obter uma instância de `Class` que será apresentada é através de um `String` com o nome completo da classe. Essa forma é muito útil para obter nomes de classes de arquivos de configuração e instanciá-las a partir da reflexão. Para fazer isso, é preciso chamar o método estático `forName()` de `Class` passando o nome da classe como parâmetro. Ao fazer isso, ela será procurada pela máquina virtual, carregada caso isso ainda não tenha sido feito e retornada. A chamada desse método pode lançar um `ClassNotFoundException` caso a classe não seja encontrada.

USO DO CLASS.FORNAME() NO JDBC

Talvez alguns sintam alguma familiaridade com a chamada `Class.forName()`, pois já viram isso ser utilizado em algum lugar. Um uso comum dessa chamada é para carregar o driver JDBC para o acesso ao banco de dados. Muitos desenvolvedores acabam utilizando-a como parte de uma receita de bolo para acessar um banco de dados, sem saber exatamente o que ela está fazendo. Nesse caso, a chamada faz com que o driver de acesso ao banco seja carregado pela máquina virtual. O carregamento dessa classe resulta na execução de um bloco estático que registra o driver na classe `DriverManager`, o que faz com que as URLs de conexão daquele banco sejam repassadas para aquela classe.

Para exemplificarmos o uso dessa abordagem, a listagem a seguir mostra uma classe que lê um arquivo de propriedades (o qual, em cada linha, possui propriedade = valor), com interfaces e suas respectivas implementações. Nesse arquivo, as propriedades seriam os nomes das interfaces e o valor o nome de sua respectiva implementação que deve ser utilizada na aplicação. A classe `FornecedorImplementacoes` recebe o nome do arquivo no construtor e utiliza a classe `Properties` para fazer a leitura do arquivo. Para cada propriedade, o `Class.forName()` é utilizado para obter instância de `Class` da interface e sua respectiva implementação, adicionando ambas em um mapa para posterior recuperação.

Classe que lê arquivo de propriedades com interfaces e implementações:

```
public class FornecedorImplementacoes {

    private Map<Class<?>, Class<?>> implementacoes =
                                                    new HashMap<>();

    public FornecedorImplementacoes(String nomeArquivo)
                    throws IOException, ClassNotFoundException{
        Properties p = new Properties();
        p.load(new FileInputStream(nomeArquivo));
        for(Object interf : p.keySet()){
            Class<?> interfType =
                Class.forName(interf.toString());
            Class<?> implType =
                Class.forName(p.get(interf).toString());
            implementacoes.put(interfType, implType);
        }
    }
    public Class<?> getImplementacao(Class<?> interf){
        return implementacoes.get(interf);
    }
}
```

A classe `FornecedorImplementacoes` pode ser utilizada para configurar em uma aplicação qual a implementação de cada interface. O método `getImplementacao()` recebe como argumento a interface e retorna a classe da implementação. O código a seguir mostra como ela seria utilizada. Ela é criada passando como parâmetro o arquivo `implementacoes.prop` e tenta recuperar a implementação de uma interface chamada `DAO`. Caso a implementação seja retornada, o nome da classe será impresso no console, e caso contrário, a mensagem de erro será exibida.

Exemplo de uso da classe FornecedorImplementacoes:

```
public static void main(String[] args){
```

```
    try {
        FornecedorImplementacoes f =
            new FornecedorImplementacoes("implementacoes.prop");
        Class<?> impl = f.getImplementacao(DAO.class);
        System.out.println("Implementação recuperada: "
                           + impl.getName());
    } catch (ClassNotFoundException | IOException e) {

        System.out.println(
            "Problemas ao obter implementações:" +
                 e.getMessage());

    }
}
```

2.2 TRABALHANDO COM CLASSES

Depois de obter a representação de uma classe através de uma instância de `Class`, é possível conseguir diversas informações além de criar instâncias. Como foi dito na introdução, a API Reflection da linguagem Java não permite que as classes existentes sejam modificadas ou manipuladas, sendo todas as informações somente para leitura. O que pode ser manipulado a partir da API Reflection são os objetos dessas classes, nos quais os atributos podem ser modificados e os métodos podem ser invocados. Nesta seção veremos quais informações podem ser extraídas de uma classe e quais são as formas para instanciá-la através de reflexão.

Acessando informações de classes

Através de uma instância de um objeto do tipo `Class` é possível recuperar diversas informações de uma classe. Como `Class` é utilizada para a representação de qualquer tipo, mesmo que esse não seja exatamente uma classe, existem métodos que podem ser utilizados para verificar o que aquela instância está

representando, como `isInterface()`, `isPrimitive()`, `isEnum()` e `isArray()`. A recuperação de métodos e atributos será abordada nas seções seguintes deste capítulo.

A recuperação dos modificadores de uma classe já é um pouquinho mais complicada, pois o método `getModifiers()` retorna um inteiro que possui uma representação de quais são os modificadores da classe. Eles incluem, para uma classe, `public`, `protected`, `private`, `final`, `static`, `abstract` e `interface`. Para métodos e atributos, a recuperação é similar, porém os atributos também possuem modificadores como `volatile` e `transient` e os métodos podem possuir os modificadores `synchronized`, `strictfp` e `native`. Esses outros também são incluídos quando aplicável.

Para decodificar os modificadores, verificando se algum está presente, a classe `Modifier` precisa ser utilizada. Ela possui métodos estáticos que recebem o inteiro retornado e verificam se um modificador específico está presente. A listagem a seguir mostra um código que exemplifica esse processo, verificando se a classe é abstrata.

Acessando os modificadores da classe:

```
int modificadores = clazz.getModifiers()
if(Modifier.isAbstract(modificadores)){
    //codigo se a classe for abstrata
}
```

Um tipo de verificação que frequentemente precisamos realizar em uma classe é a respeito da sua relação com outros tipos e objetos. A operação `instanceof` é possível ser realizada com um objeto e uma referência estática de uma classe. O método `isInstance()` possui uma lógica similar, porém utilizando uma

instância de `Class`. Ele verifica se o objeto passado como parâmetro é uma instância daquela classe.

Outro método relacionado à tipagem verifica se um objeto da classe passada como parâmetro pode ser convertido na classe onde o método está sendo chamado. O método `isAssignableFrom()` irá retornar `true` caso a classe onde ele está sendo chamado seja uma superclasse, uma interface ou a própria classe que está sendo passada como parâmetro.

Para exemplificar a recuperação de informações da classe, segue um exemplo de aplicação em que o usuário escreve o nome da classe no console e ele imprime a hierarquia de classes e interfaces. No método `main()`, o programa lê o nome da classe utilizando `Scanner` e obtém a instância de `Class` com o método `forName()`. Em seguida, o método `imprimirHierarquia()` é chamado passando a classe como parâmetro.

Classe que imprime a hierarquia de classes e interfaces:

```
public class InformacaoClasse {

    public static void main(String[] args) {
        System.out.println(
            "Entre com o nome da classe que deseja informação:"
        );
        Scanner in = new Scanner(System.in);
        String nomeClasse = in.nextLine();
        try {
            Class<?> c = Class.forName(nomeClasse);
            imprimirHierarquia(c, 1);
        } catch (ClassNotFoundException e) {
            System.out.println(
                "A classe "+nomeClasse+" não foi encontrada"
            );
        }
        in.close();
    }
```

```
    private static void
    imprimirHierarquia(Class<?> c, int nivel){
        List<Class<?>> lista = getSuperclasseEInterfaces(c);
        String recuo = "";
        for(int i=0; i<nivel; i++){
            recuo+="   ";
        }
        for(Class<?> clazz : lista){
            System.out.println(recuo+"|-> "+clazz.getName());
            if(clazz != Object.class){
                imprimirHierarquia(clazz, nivel+1);
            }
        }
    }
    private static List<Class<?>>
    getSuperclasseEInterfaces(Class<?> c){
        List<Class<?>> lista = new ArrayList<>();
        if(c.getSuperclass() != null)
            lista.add(c.getSuperclass());
        lista.addAll(Arrays.asList(c.getInterfaces()));
        return lista;
    }
}
```

O método `imprimirHierarquia()` possui uma lógica recursiva e irá ser chamado novamente para cada classe ou interface encontrados, parando ao encontrar a classe `Object`. O parâmetro `nivel` representa a profundidade da classe na hierarquia e é utilizado para determinar a quantidade de espaços antes de a classe ser impressa no console. Dessa forma, a impressão das classes ficará com um formato similar ao de uma árvore. A cada chamada recursiva, adiciona-se um no parâmetro `nivel`.

O método `getSuperclasseEInterfaces()` retorna uma lista de `Class` com sua superclasse e as interfaces diretamente implementadas por ela. Observe que os métodos `getSuperclass()` e `getInterfaces()` são utilizados para isso. O método `getInterfaces()` irá retornar as interfaces

implementadas caso o `Class` represente uma classe e as interfaces estendidas caso seja uma interface. O código adiante apresenta a saída do software caso seja fornecida a entrada `java.util.ArrayList`.

Saída do programa para a classe ArrayList:

```
Entre com o nome da classe que deseja informação:
java.util.ArrayList
   |-> java.util.AbstractList
      |-> java.util.AbstractCollection
         |-> java.lang.Object
         |-> java.util.Collection
            |-> java.lang.Iterable
      |-> java.util.List
         |-> java.util.Collection
            |-> java.lang.Iterable
   |-> java.util.List
      |-> java.util.Collection
         |-> java.lang.Iterable
   |-> java.util.RandomAccess
   |-> java.lang.Cloneable
   |-> java.io.Serializable
```

Criando instâncias

Uma classe não é nada sem as suas instâncias! Se queremos carregar classes de arquivos de configuração para utilizarmos na aplicação, é preciso criar instâncias delas. Isso pode ser feito facilmente através do método `newInstance()` de `Class`! Esse método cria uma nova instância da classe a partir de um construtor vazio. Ao invocar esse método, existem duas exceções que precisam ser tratadas: `InstantiationException` e `IllegalAccessException`. A primeira será lançada caso não exista um construtor vazio ou a instância de `Class` represente uma classe abstrata, interface, tipo primitivo ou array. A segunda exceção já será lançada caso o construtor não esteja acessível, por

exemplo, por ser privado.

Frameworks que pedem construtores sem parâmetros

Existem vários frameworks que em sua documentação dizem que, para suas classes poderem ser manipuladas por ele, precisam possuir um construtor sem parâmetro. Agora você já sabe que isso é porque ele utiliza esse método `newInstance()` para instanciá-la. Repare que isso é comum quando o framework precisa instanciar classes da sua aplicação, por exemplo, em frameworks web para classes que representam controllers, que são criadas e gerenciadas pelo framework para tratar requisições. Outro exemplo ocorre em frameworks de mapeamento objeto-relacional, os quais precisam instanciar as classes mapeadas para o banco quando informações são recuperadas.

O método `newInstance()` tem um sério problema em relação a exceções! Caso o construtor lance uma exceção, mesmo que checada, essa exceção será propagada diretamente para o método que tentou criar a instância. Isso sem exigir que essa exceção seja tratada! Veja um exemplo disso no código apresentado na listagem a seguir. Quando ele for executado, a `IOException` lançada no construtor será propagada para o método `main()` sem cair em nenhum dos blocos `catch`. Para conseguir tratar essa questão, deve-se utilizar diretamente a classe `Constructor`.

Tratamento de exceções com o método newInstance():

```
public class CriacaoClasse {

    public CriacaoClasse() throws IOException{
        throw new IOException();
    }
    public static void main(String[] args){
        try {
            CriacaoClasse obj =
                CriacaoClasse.class.newInstance();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
}
```

Os construtores podem ser obtidos de uma classe através dos métodos `getConstructor()` ou `getConstructors()` . O primeiro recebe como parâmetro os tipos dos parâmetros do construtor, permitindo recuperar um construtor específico. Esse método utiliza o recurso da linguagem de **varargs** para permitir que se passem diversos parâmetros com os tipos. O segundo recupera uma lista com todos os construtores da classe. Para utilizar o construtor, basta chamar o método `newInstance()` passando os parâmetros adequados. Da mesma forma que a recuperação do construtor, ele também utiliza **varargs** para que se possa passar quando parâmetros forem necessários.

Em relação às exceções, a recuperação de um construtor específico pode lançar o erro `NoSuchMethodException` caso aquele construtor não exista. Ao se fazer a invocação de um construtor, além das duas exceções que podem ser lançadas pelo `newInstance()` de `Class` , duas outras exceções podem ocorrer. A exceção `IllegalArgumentException` é lançada quando a quantidade e os tipos dos parâmetros não corresponderem aos do

construtor e `InvocationTargetException` é a exceção que encapsula uma exceção que é lançada pelo construtor. Nesse caso, a original pode ser recuperada através do método `getTargetException()`.

O código a seguir mostra o exemplo do uso de um construtor. A classe `UsoConstrutor` possui um construtor que recebe uma `String` como argumento. O método `getConstructor()` é utilizado para recuperar o construtor e o método `newInstance()` para a criação do objeto. Observe que, no tratamento de `InvocationTargetException`, a exceção original é recuperada e impressa no console.

Exemplo de criação de instância com construtores:

```
public class UsoConstrutor {

    public UsoConstrutor(String s){
        System.out.println("Construtor invocado com: "+s);
    }
    public static void main(String[] args)
        throws NoSuchMethodException, SecurityException {
        Class<UsoConstrutor> c = UsoConstrutor.class;
        Constructor<UsoConstrutor> constr =
            c.getConstructor(String.class);
        try {
            UsoConstrutor obj = constr.newInstance("teste");
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (IllegalArgumentException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            System.out.println(
            "Exceção lançada pelo construtor: "
             + e.getTargetException());
        }
    }
```

```
}
```

QUANDO CRIAR OBJETOS UTILIZANDO REFLEXÃO?

A criação de objetos utilizando reflexão é indicada quando não se conhece em tempo de compilação a classe que vai ser instanciada. No caso de haver diversas implementações, mas todas serem conhecidas, um padrão de criação, como Factory Method ou Builder, pode ser utilizado para determinar a classe que será criada. A criação de objetos através de reflexão é indicada quando se deseja que novas classes possam ser criadas e configuradas como plugins de um software ou framework existente.

Melhorando o fornecedor de implementações

A partir do que foi visto nessa seção, podemos melhorar a implementação da classe `FornecedorImplementacoes`. A primeira verificação será na leitura do arquivo, onde podemos verificar se a classe utilizada como chave é realmente uma interface ou uma classe abstrata e a classe passada como valor é realmente uma implementação concreta dela. Outra modificação é que em vez de retornar somente a classe da implementação, vamos ter também métodos para já retornar instâncias das implementações.

Começando pelas verificações, inicialmente vamos criar métodos auxiliares que verificam propriedades das classes, conforme mostrado na listagem a seguir. O método `isInterfaceOuAbstract()` verifica se o `Class` passado como parâmetro é uma interface ou uma classe abstrata. Observe que a

classe auxiliar `Modifiers` foi utilizada para verificar se o modificador `abstract` está presente. Em seguida, temos o método `isAbstracaoEImplementacao()` que verifica se a primeira classe é uma abstração, se a segunda é uma implementação e se a implementação é do tipo da abstração.

Métodos auxiliares para verificação:

```
private boolean isInterfaceOuAbstract(Class<?> c){
    return c.isInterface() ||
           Modifier.isAbstract(c.getModifiers());
}
private boolean isAbstracaoEImplementacao
        (Class<?> interf, Class<?> impl){

    return isInterfaceOuAbstract(interf) &&
           !isInterfaceOuAbstract(impl)
            && interf.isAssignableFrom(impl);

}
```

A listagem a seguir mostra como o construtor da classe `FornecedorImplementacoes` pode ser modificado para incluir a verificação. Observe que, depois que os objetos `Class` são criados, caso o método `isAsbtracaoEImplementacao()` retorne `false`, uma exceção será lançada para indicar um erro de configuração no arquivo.

Construtor modificado para fazer as verificações:

```
public FornecedorImplementacoes(String nomeArquivo) throws
Exception{
    Properties p = new Properties();
    p.load(new FileInputStream(nomeArquivo));
    for(Object interf : p.keySet()){
        Class<?> interfType = Class.forName(interf.toString());
        Class<?> implType =
            Class.forName(p.get(interf).toString());
        if(!isAbstracaoEImplementacao(interfType, implType)){
```

```
            throw new Exception(
                "Erro na configuração do arquivo " +
                nomeArquivo + " : " + interfType.getName() +
                " não é abstração de " + implType.getName());
        }
        implementacoes.put(interfType, implType);
    }
}
```

A segunda parte que será adicionada no fornecedor de implementações é a que cria os objetos a partir da classe da respectiva implementação. Porém, não queremos que seja utilizado apenas o construtor sem parâmetros. O objetivo é que, a partir dos parâmetros passados para a criação do objeto, seja encontrado um construtor que possa receber aqueles parâmetros. A listagem a seguir apresenta um método auxiliar que será utilizado para achar um construtor adequado aos parâmetros passados.

Métodos auxiliar que busca um construtor adequado:

```
private Constructor<?> acharConstrutor(Class<?> c,
Object... objs) throws Exception{
    for(Constructor<?> constr : c.getConstructors()){
        Class<?>[] params = constr.getParameterTypes();
        if(params.length == objs.length){
            boolean erro = false;
            for(int i=0; i<objs.length && !erro; i++){
                if(!params[i].isInstance(objs[i]))
                    erro = true;
            }
            if(!erro)
                return constr;
        }
    }
    throw new Exception("Construtor não encontrado");
}
```

Inicialmente, o método recupera a lista de construtores a partir do método `getConstructors()`. Em seguida, para cada

construtor são feitas verificações para ver se ele é adequado aos parâmetros. A primeira questão verificada é se o número de parâmetros do construtor é igual ao número de objetos passados como parâmetro. Depois, para cada parâmetro é verificado se o objeto possui o tipo exigido para ele, utilizando o método `isInstance()`. Caso a verificação seja positiva para todos, o construtor é retornado para o método. Em caso negativo, uma exceção é lançada indicando que um construtor para aqueles parâmetros não foi encontrado. Pela lógica, é possível perceber que o primeiro construtor que se encaixe será retornado.

Para completar essa funcionalidade, a próxima listagem apresenta o método `criarInstancia()` que efetivamente cria o objeto. Esse método inicialmente utiliza o método `getImplementacao()` para recuperar a implementação da interface ou classe abstrata passada como parâmetro. Em seguida, o método criado na listagem anterior é utilizado para recuperar o construtor, o qual é utilizado para a criação da instância através do método `newInstance()`.

Criação da instância através do construtor:

```
public Object criarInstancia(Class<?> interf, Object... objs)
        throws Exception{
    Class<?> impl = getImplementacao(interf);
    Constructor<?> constr = acharConstrutor(impl, objs);
    return constr.newInstance(objs);
}
```

Uma ressalva em relação a essa implementação é que ela não irá funcionar caso o construtor receba tipos primitivos como parâmetros. O problema é que o tipo primitivo será convertido para sua respectiva classe **wrapper** ao ser passado para o **varargs** no array de objetos. Dessa forma, a invocação do método

`isInstance()` irá retornar `false` por causa dessa conversão. Deixo como exercício ao leitor fazer essa implementação funcionar para construtores que recebem tipos primitivos.

CUIDADO COM OS TIPOS PRIMITIVOS!

Os tipos primitivos são uma grande pedra no sapado de quem trabalha com reflexão. Apesar de o seu tipo ser representado pela classe `Class` , muitas vezes é preciso fazer a distinção dos tipos primitivos e tratá-los como casos especiais. O exemplo da busca de construtores é um caso em que os tipos primitivos precisariam ser tratados separadamente, trocando seu tipo pelo de uma classe **wrapper** ou utilizando um condicional para cada tipo primitivo. Sendo assim, não se incomode ao ter que criar uma série de condicionais para tratar de forma especial cada um dos tipos primitivos, pois nos frameworks que desenvolvi precisei fazer isso algumas vezes.

2.3 MANIPULANDO OBJETOS

Nas seções anteriores vimos como recuperar a referência de uma classe, como acessar suas informações e como criar novas instâncias. Esta seção mostra como é possível manipular os objetos com reflexão através do acesso aos seus atributos e da invocação de seus métodos. Existem algumas questões mostradas para uma classe, cuja recuperação é similar quando lidamos com atributos e métodos. Por exemplo, a recuperação de modificadores em

métodos e atributos também é feita com o método `getModifiers()` e também é decodificada com a classe `Modifier`. Essas questões que são muito similares não serão apresentadas novamente.

Acessando e alterando valores de atributos

Através da API Reflection é possível recuperar os atributos de uma classe, que são representados pela classe `Field`. A partir dessas referências dos atributos, seus valores podem ser acessados e modificados. Existem dois métodos para retornar a lista de atributos de uma classe: `getFields()`, que retorna todos os atributos públicos; e `getDeclaredFields()`, que retorna todos os atributos que foram declarados naquela classe. Para exemplificarmos a diferença entre esses dois métodos, considere o código apresentado na listagem seguinte.

Programa que mostra a diferença entre getFields() e getDeclaredFields():

```
public class SuperclasseAtributo {
    private int atributoSuperclasseUm;
    public String atributoSuperclasseDois;
}

public class AcessoAtributo extends SuperclasseAtributo{

    private int atributoUm;
    public String atributoDois;

    public static void main(String[] args) {
        System.out.println("Retornado pelo getFields()");
        for(Field f : AcessoAtributo.class.getFields()){
            System.out.println(
                "- "+f.getType().getSimpleName() + " " +
                f.getName());
        }
```

```
        System.out.println(
            "Retornado pelo getDeclaredFields()");
        for(Field f : AcessoAtributo.class.getDeclaredFields()){
            System.out.println(
                "- "+f.getType().getSimpleName() + " "
                + f.getName());
        }
    }
}
```

O resultado da execução desse programa está apresentado na listagem a seguir. Observe que o método `getFields()` retorna todos os atributos públicos, incluindo tanto os da própria classe quanto os da superclasse. Já o método `getDeclaredFields()` retorna todos os atributos declarados na própria classe, incluindo os que são privados, porém não incluindo nenhum declarado na superclasse. Se quisermos todos os atributos, incluindo os privados da classe e das superclasses, precisamos ir percorrendo a hierarquia de classes e recuperando os atributos de cada uma com `getDeclaredFields()`. De forma similar, se quisermos acessar um atributo específico da classe, também temos dois métodos com comportamento análogo, sendo eles `getField()` e `getDeclaredField()`.

Saída do programa com a diferença de getFields() para getDeclaredFields():

```
Retornado pelo getFields()
- String atributoDois
- String atributoSuperclasseDois
Retornado pelo getDeclaredFields()
- int atributoUm
- String atributoDois
```

As duas ações que podemos fazer com o atributo de um objeto são a recuperação e a modificação de seu valor. A classe `Field` possui dois métodos que executam respectivamente essas ações,

sendo eles o método `get()` e o método `set()`. Ambos recebem como parâmetro o objeto no qual esse atributo deve ser acessado. Para exemplificar o acesso dos valores de um atributo, considere a classe chamada `ExemploClasse` representada na próxima listagem. Observe que ela possui um atributo público, um atributo privado e um atributo estático e público.

Exemplo de classe com atributos a serem acessados por reflexão:

```
public class ExemploClasse {
    public String publico;
    private String privado;
    public static String estatico;
}
```

A listagem a seguir mostra um código que acessa cada um desses atributos, inserindo como valor seu próprio nome e, em seguida, recuperando esse valor e imprimindo-o no console. O método `escreverLerAtributo()` recebe como parâmetro o `Field` e o objeto no qual ele precisa ser recuperado. Esse método usa os métodos `set` e `get` para primeiro atribuir o nome do atributo como valor e depois recuperá-lo. A exceção `IllegalArgumentException` irá acontecer quando um objeto de um tipo errado for passado para os métodos de `Field`, com o que não precisamos nos preocupar muito nesse caso. Por outro lado, a exceção `IllegalAccessException` irá ocorrer caso haja a tentativa de acesso a um atributo cujo acesso não é permitido.

Acesso a diferentes tipos de atributos por reflexão:

```
public class EscrevendoLendoAtributos {

    public static void main(String[] args)
            throws NoSuchFieldException, SecurityException {
```

```
        ExemploClasse instancia = new ExemploClasse();
        Class<?> clazz = instancia.getClass();
        escreverLerAtributo(
                clazz.getField("publico"), instancia);
        escreverLerAtributo(
                clazz.getDeclaredField("privado"), instancia);
        escreverLerAtributo(clazz.getField("estatico"), null);
    }
    public static void
     escreverLerAtributo(Field f, Object instancia){
        try {
            f.set(instancia, f.getName());
            Object valor = f.get(instancia);
            System.out.println(
                "Escrito e lido o atributo = "+valor);
        } catch (IllegalArgumentException e) {
            System.out.println("Problemas ao acessar atributo "
                +f.getName()+": "+e.getMessage());
        } catch (IllegalAccessException e) {
            System.out.println(
                "Problemas de acesso no atributo "
                +f.getName()+": "+e.getMessage());
        }
    }
}
```

O método `main()` cria uma instância da classe `ExemploClasse` e recupera sua respectiva instância de `Class`, a partir da qual os atributos são recuperados. Observe que o atributo privado precisa ser recuperado com `getDeclaredField()`, pois o método `getField()` só irá encontrar os públicos. Outra observação importante é que não é necessária uma instância para que um atributo estático seja acessado, visto que ele está vinculado à classe.

Ao executar esse programa, o atributo privado não conseguirá ser acessado e a exceção `IllegalAccessException` será lançada. Caso esse código estivesse na própria classe `ExemploClasse`, esse atributo estaria acessível e o erro não aconteceria. É importante

ressaltar que, por mais que os atributos privados estejam acessíveis para ser recuperados via reflexão, nem sempre eles poderão ser acessados. Isso vai depender de onde o código está sendo executado e as mesmas restrições que se aplicariam ao acesso direto também se aplicam a reflexão. O último capítulo deste livro mostra como essa regra pode ser burlada para permitir o acesso de membro privados através da API Reflection.

Invocando métodos

Não sobrou muita coisa a ser dita sobre métodos, pois a recuperação dos métodos é similar à de atributos e a invocação também não é muito diferente da chamada de construtores. A recuperação de um conjunto de métodos é similar à recuperação de atributos, havendo dois métodos chamados `getMethods()` e `getDeclaredMethods()` . Em Java, um método é identificado unicamente pelo seu nome e pelos seus parâmetros, pois podem existir métodos com o mesmo nome e que recebem parâmetros diferentes. Dessa forma, os métodos `getMethod()` e `getDeclaredMethod()` recebem como primeiro parâmetro o nome e em seguida um número variável de objetos do tipo `Class` para representar os tipos dos parâmetros.

Os métodos são representados pela classe `Method` e possuem diversas informações interessantes de serem recuperadas, que podem ser utilizadas quando procuramos um método adequado para ser invocada. O método `getReturnType()` retorna o tipo do retorno, que pode ser utilizado para saber o que deve se esperar de sua invocação. Se o método em questão não possuir retorno, ele irá retornar o tipo `void.class` . Outro método interessante é o `getParameterTypes()` , que retorna um array com os tipos dos

parâmetros. No exemplo apresentado na introdução do livro, esses dois métodos foram utilizados para verificar se o método começado com "get" retornava algo diferente de `void` e não possuía parâmetros. Uma outra informação importante são os tipos de exceções lançadas, que podem ser recuperados com `getExceptionTypes()`.

A invocação de métodos é feita a partir do método `invoke()`, que recebe como primeiro parâmetro o objeto em que ele deve ser invocado e, em seguida, os valores que devem ser passados como argumento. Ele irá retornar o retorno do método que está sendo invocado. Assim como na invocação de construtores, caso uma exceção seja lançada pelo método, será do tipo `InvocationTargetException`, através da qual a exceção original pode ser recuperada com o `getTargetException()`. E assim como o acesso a atributos estáticos, um método estático pode ser invocado passando como `null` o primeiro parâmetro, o qual representa o objeto onde ele deve ser invocado.

Para exemplificar a invocação de métodos por reflexão, vamos mostrar um exemplo onde o usuário escolhe a classe e o método que ele deseja executar. A partir dessa escolha, o programa irá verificar quais são os parâmetros necessários para a execução daquele método e pedir que o usuário entre com cada um deles. Por fim, o método será invocado com os parâmetros passados e o retorno será exibido no console. É importante ressaltar aqui que o objetivo desse exemplo é simplesmente ilustrar com código a invocação de métodos via reflexão, e não realmente suportar a invocação de qualquer tipo de método via linha de comando.

Para começar o exemplo, a partir da entrada do usuário do

nome do método precisaremos procurá-lo entre os métodos da classe. Como não é possível recuperá-lo diretamente de `Class` apenas pelo seu nome, pois os tipos dos parâmetros seriam também necessários, deve-se percorrer toda lista de métodos procurando um método entrado pelo usuário. Sendo assim, o método auxiliar apresentado na listagem a seguir realiza essa procura em uma classe, retornando o primeiro que for encontrado com o nome passado como parâmetro.

Método auxiliar que procura o método pelo seu nome:

```
private static Method
 acharMetodoPeloNome(Class<?> c, String nome) throws Exception{
    for(Method m : c.getMethods()){
        if(m.getName().equals(nome)){
            return m;
        }
    }
    throw new Exception("Método "+nome+" não encontrado");
}
```

Na listagem a seguir, está apresentado o corpo principal do programa que executa o método de acordo com as entradas do usuário. Inicialmente, ele solicita ao usuário o nome da classe e, em seguida, o nome do método. Ele será procurado utilizando o `acharMetodoPeloNome()` apresentado na listagem anterior.

A partir do método recuperado, os tipos e a quantidade dos parâmetros são recuperados para serem solicitados ao usuário. Para cada parâmetro, é solicitado para o usuário que entre com um valor. Esse valor é lido como uma `String` e é utilizado para a criação do objeto que será o valor do parâmetro, através da invocação de um construtor que recebe uma `String` como argumento. Sendo assim, se o tipo do parâmetro for `Integer`, será invocado seu construtor passando a `String` lida como

parâmetro.

Depois que todos os parâmetros forem lidos, o método é invocado utilizando `invoke()` e seu resultado é impresso no console.

Código que executada um método de acordo com as entradas do usuário:

```
public class ExecutaMetodo {

    public static void main(String[] args) throws Exception{
        System.out.println(
            "Entre com o nome da classe "
            + "com método que deseja executar:");
        Scanner in = new Scanner(System.in);
        String nomeClasse = in.nextLine();
        Class<?> c = Class.forName(nomeClasse);
        System.out.println("Entre com o nome do método:");
        String nomeMetodo = in.nextLine();
        Method m = acharMetodoPeloNome(c, nomeMetodo);
        Object[] params =
            new Object[m.getParameterTypes().length];
        for(int i=0; i<params.length; i++){
            Class<?> paramType = m.getParameterTypes()[i];
            System.out.println("Parametro "+(i+1)+
                            " ("+paramType.getName()+")");
            String valor = in.nextLine();
            params[i] = paramType
                        .getConstructor(String.class)
                        .newInstance(valor);
        }
        Object retorno = m.invoke(c.newInstance(), params);
        System.out.println("O método retornou: " + retorno);
        in.close();
    }
}
```

Vale ressaltar que diversas coisas poderiam dar errado na execução desse método, que está simplesmente declarando que se pode jogar uma exceção e não estão sendo tratadas da forma

apropriada. Ele assume, por exemplo, que a classe entrada pelo usuário possuirá um construtor vazio e que os tipos dos parâmetros terão construtores que recebem uma `String` como parâmetro. Em uma aplicação real, seria importante tratar os casos em que essas premissas não são verdade. Por exemplo, se quiséssemos dar suporte a parâmetros que são tipos primitivos, precisaríamos de um tratamento especial para cada um deles.

A classe apresentada a seguir, chamada `Utilitaria`, possui um método que será utilizado para testarmos a execução de métodos através do programa desenvolvido. O método `repetir()` dessa classe recebe três parâmetros e possui uma lógica bem simples. Ele irá retornar uma `String` em que o parâmetro `base` é repetido segundo o parâmetro `vezes`, separado pelo `divisor`.

Classe utilizada para testar a execução de métodos:

```
public class Utilitaria {

    public String
     repetir(String base, String divisor, Integer vezes){
        String retorno = base;
        for(int i=1; i<vezes; i++){
            retorno += divisor+base;
        }
        return retorno;
    }
}
```

A listagem a seguir apresenta a saída do console a partir da execução desse programa. Observe que o nome da classe `Utilitaria` e o nome do método `repetir()` são entrados como as duas entradas iniciais do usuário. Em seguida, o programa pede o valor para cada parâmetro, mostrando o tipo esperado para

ele. Por fim, depois de entrar com valores para os três parâmetros, é exibido o retorno do método como esperado.

Saída do programa que executa o método repetir():

```
Entre com o nome da classe com método que deseja executar:
org.casadocodigo.Utilitaria
Entre com o nome do método:
repetir
Parametro 1 (java.lang.String)
teste
Parametro 2 (java.lang.String)
##
Parametro 3 (java.lang.Integer)
4
O método retornou: teste##teste##teste##teste
```

QUANDO EXECUTAR MÉTODOS POR REFLEXÃO?

A orientação a objetos provê meios de você invocar métodos em um objeto que você não conhece previamente. Isso pode ser feito definindo uma abstração, como uma interface ou uma classe, que possua esse método e utilizando o polimorfismo para invocá-lo em qualquer objeto que obedeça a essa abstração. Nesses casos, quando é possível ter uma abstração que representa o método que deve ser invocado, a reflexão não precisa ser utilizada. Por outro lado, se você precisa lidar com classes que possuem métodos diferentes e não faz sentido compartilharem uma mesma abstração, essa é a deixa para a invocação desses métodos por reflexão. Um bom exemplo desse caso são os Java Beans, que possuem diversos métodos getter e setter diferentes e não é possível criar uma interface comum que capture essa característica.

2.4 PROCURANDO MÉTODOS E ATRIBUTOS PARA VALIDAÇÃO

Durante as seções anteriores, foram apresentados vários exemplos que demonstraram o funcionamento da API Reflection. O objetivo desta seção é consolidar esse conhecimento através de um exemplo mais realista. O requisito principal é criar uma classe que invoca para um determinado objeto rotinas de validação. Essas rotinas de validação podem ser métodos definidos na própria classe ou através de atributos que implementam a interface `Validador`, que recebe o próprio objeto como parâmetro e está apresentada na listagem a seguir.

A ideia é que os validadores armazenados em atributos possam ser reutilizados em diferentes classes e que os definidos em métodos possuam validações mais específicas da própria classe. O objetivo é que na validação todos os validadores presentes na classe sejam invocados e, caso existam erros, uma lista com eles seja retornada.

Interface para atributos com validadores:

```
public interface Validador {
    public void validar(Object o) throws Exception;
}
```

Vamos começar o exemplo pelo método que executa os validadores que estão presentes nos atributos. Ele deve percorrer os atributos da classe e buscar pelos que implementam a interface `Validador`. À medida que esses validadores forem sendo encontrados, eles devem ser executados e, caso uma exceção seja lançada, esse erro deve ser armazenado e retornado. A implementação desse método está apresentada na listagem a

seguir. Um fator a ser observado é que o método `validar()` não é invocado utilizando reflexão e sim polimorfismo através da própria interface. Como o tipo do atributo era conhecido, não foi necessário utilizar reflexão para essa invocação.

Método que executa os validadores em atributos da classe:

```
private List<Exception>
 chamarValidadores(Object obj, Class<?> clazz){
    List<Exception> erros = new ArrayList<>();
    for(Field f : clazz.getFields()){
        if(Validador.class.isAssignableFrom(f.getType())){
            try {
                Validador v = (Validador) f.get(obj);
                v.validar(obj);
            } catch
                (IllegalArgumentException |
                 IllegalAccessException e) {
                throw new RuntimeException(e);
            } catch (Exception e) {
                erros.add(e);
            }
        }
    }
    return erros;
}
```

Em seguida, foi criado o método que invoca os métodos de validação presentes na própria classe. Por convenção, eles devem começar com a palavra "validar" e não devem possuir parâmetros, visto que, como estão na própria classe, podem acessar seus atributos. A listagem a seguir mostra o método `chamarMetodosValidacao()`, que percorre os métodos da classe buscando os métodos de validação, invocando-os. Observe que, caso o método `invoke()` lance uma `InvocationTargetException`, esta é capturada pela cláusula `catch` e a exceção original é recuperada pelo método

`getTargetException()` e adicionada na lista de erros.

Método que chama os métodos validadores da classe:

```
private List<Exception> chamarMetodosValidacao
                            (Object obj, Class<?> clazz){
    List<Exception> erros = new ArrayList<>();
    for(Method m : clazz.getMethods()){
        if(m.getName().startsWith("validar")
                && m.getParameterTypes().length == 0){
            try {
                m.invoke(obj);
            } catch
                (IllegalAccessException |
                 IllegalArgumentException e) {

                throw new RuntimeException(e);

            } catch (InvocationTargetException e) {
                erros.add((Exception)e.getTargetException());
            }
        }
    }
    return erros;
}
```

Para finalizar, precisamos ter um método principal que invoca os dois métodos e reúne os erros. Para que seja possível fazer isso, precisamos definir uma exceção de validação que reúna esses erros e possa ser lançada a quem invocar o método. Dessa forma, a listagem a seguir mostra a classe `ValidacaoException`, que possui um atributo que recebe uma lista de exceções em seu construtor e permite sua posterior recuperação.

Exceção que reúne diversos erros de validação:

```
public class ValidacaoException extends Exception {

    private List<Exception> erros;

    public ValidacaoException(List<Exception> erros) {
```

```
        this.erros = erros;
    }
    public List<Exception> getErros() {
        return erros;
    }
}
```

Finalmente, a listagem a seguir apresenta o método `validarObjeto()` que orquestra a invocação dos outros dois métodos desenvolvidos. Inicialmente, ele extrai a instância de `Class` do objeto recebido como argumento e cria uma lista para armazenar os erros retornados. Em seguida os métodos são invocados, tendo os erros retornados adicionados na lista. No fim, caso a lista possua algum erro, é criada uma nova `ValidacaoException` com essa lista, que é lançada para o método que o invocou. Sendo assim, caso o método capture essa exceção, ele terá acesso a todos os erros lançados pelos validadores.

Método principal que faz a validação do objeto:

```
public class ValidadorObjetos {

    public void validarObjeto(Object obj) throws
     ValidacaoException {
        Class<?> clazz = obj.getClass();
        List<Exception> erros = new ArrayList<>();
        erros.addAll(chamarMetodosValidacao(obj, clazz));
        erros.addAll(chamarValidadores(obj, clazz));
        if(erros.size() > 0){
            throw new ValidacaoException(erros);
        }
    }
}
```

Diferentemente dos exemplos anteriores, onde mostro o método desenvolvido sendo utilizado, desta vez deixarei isso como exercício para os leitores. Procure criar uma classe que possua tanto métodos de validação quanto atributos que tenha o tipo

`Validator` . Tente também criar contraexemplos, ou seja, métodos e atributos que não deveriam ser incluídos e veja se realmente são deixados de fora.

2.5 COISAS QUE PODEM DAR ERRADO

"Com grandes poderes vêm grandes responsabilidades" - Ben Parker, tio do Homem-Aranha

Quando trabalhamos com reflexão, existem diversos erros que podem acontecer que normalmente são validados em tempo de compilação. Por exemplo, um método pode ser invocado com quantidade ou tipos de parâmetros inválidos, pode-se tentar acessar um atributo em um objeto de uma classe que não o possui ou mesmo pode-se tentar acessar membros de classe que não estão acessíveis. Durante os exemplos, podemos ver diversas situações em que erros desse tipo eram possíveis e foram tratados.

Ao utilizar reflexão conseguimos uma maior flexibilidade, porém esse poder deve ser utilizado com responsabilidade. Todos esses erros devem ser verificados, tratados e evitados quando possível. Por exemplo, se o código assume que o método ou construtor chamado possui parâmetros de um determinado tipo, procure verificar se essa premissa é verdadeira antes de invocá-lo. Em outras palavras, procure verificar as condições com antecedência e tratar as condições excepcionais, em vez de simplesmente chamar os métodos e deixar os erros acontecerem.

Quando lidamos com reflexão, a própria estrutura da classe é um parâmetro, então devemos estar preparados para lidar com qualquer tipo de classe. Tipos como arrays e primitivos costumam

exigir tratamento especial em alguns casos. Dessa forma, o código deve considerar e tratar esses casos, e mesmo que não se deseje dar suporte a esses casos, deve-se verificar se a estrutura da classe ou método é adequada e lançar um erro caso não seja. Quando desenvolvi minhas primeiras classes que utilizavam reflexão, lembro-me de receber alguns bugs relacionados a estruturas de classe que eu não havia pensado. Sendo assim, criar testes que contemplam diversos tipos de classe é essencial para que se tenha uma boa segurança no código desenvolvido.

Por fim, procure evitar esconder erros do usuário, de forma que sua classe "falhe gentilmente" sem que ele perceba. Imagine que um arquivo de configuração possua uma lista de atributos de uma classe que precisam ser utilizados em um algoritmo que utilize reflexão, um exemplo de "falhar gentilmente" seria simplesmente pular um atributo que não existir na classe, realizando o processamento para os outros. Por mais que isso possa parecer ajudar evitando que o desenvolvedor enfrente uma série de exceções antes de tudo funcionar pela primeira vez, na verdade isso mascara um erro que pode ser difícil de ser detectado depois. Sendo assim, se encontrar alguma coisa que não está correta, lance o erro com a maior quantidade de detalhes possível, de forma a permitir que o desenvolvedor que utilizar a sua classe identifique o que está errado e realize a correção o mais cedo possível.

O capítulo *Testando classes que usam Reflexão* apresenta formas de realizar os testes de classes que utilizam reflexão. Para aqueles que pretendem utilizar reflexão em projetos reais, as técnicas apresentadas neste capítulo são essenciais para ajudar na validação e verificação desse tipo de código.

2.6 CONSIDERAÇÕES FINAIS

Este capítulo apresentou o principal da API Reflection da linguagem Java. Apesar de uma noção inicial já ter sido dada na introdução, fomos mais fundo nos detalhes e mostramos mais características dessa API. O capítulo começou apresentando as formas em que uma referência de uma classe pode ser obtida, sendo essas através de uma referência estática, um objeto e uma `String` com seu nome. Através da instância de `Class` foi mostrado como obter informações a respeito da classe e como criar instâncias dela. A partir das instâncias vimos como a reflexão pode ser utilizada para o acesso aos seus atributos e para a invocação de seus métodos.

Em todo o capítulo, a cada novo conceito apresentado, víamos um exemplo que demonstrasse de forma simples como ele funcionava. À medida que o capítulo foi seguindo, os novos exemplos procuravam incluir questões relacionadas a funcionalidades da API apresentadas previamente, mostrando como elas se encaixavam e podiam ser utilizadas em conjunto. No final, um exemplo mais próximo do real foi desenvolvido utilizando o que havia sido apresentado previamente.

Para fechar, foi ressaltada a importância de se tomar cuidado com as situações excepcionais que podem acontecer ao se utilizar reflexão. Como diversos erros que normalmente são pegos em tempo de compilação podem ocorrer, é importante tomar cuidado para que todas as situações sejam tratadas adequadamente.

Capítulo 3

METADADOS E ANOTAÇÕES

"Ao procurar uma citação sobre metadados, só encontrei metadados sobre citações." - metalinguagem sobre a escrita do livro

O prefixo *meta* é frequentemente utilizado junto com uma palavra para significar que ela está se referindo a si própria. Lembro-me das aulas de literatura, em que a professora frequentemente falava em metalinguagem. Se eu me dirijo a você, leitor, e me refiro ao momento em que escrevo esse parágrafo, estou escrevendo um texto que está se referindo ao próprio texto, sendo assim eu posso dizer que isso é metalinguagem.

Dessa forma, podemos dizer que *metadados* são dados que se referem aos próprios dados. A palavra metadados é muito utilizada em computação dentro de diferentes contextos e por isso precisamos ser um pouco mais específicos em relação ao tipo de metadados ao qual estamos nos referindo. Se estamos falando de banco de dados, os dados são aqueles que estão armazenados no banco e os metadados são a descrição desses dados, em outras palavras, a estrutura das tabelas. Para um documento XML, os metadados são representados pelo documento que descreve a estrutura do documento XML, como o DTD ou XML Schema.

Quando o contexto é uma aplicação orientada a objetos, podemos dizer que os dados são representados pelas instâncias e os metadados são os dados que as descrevem, ou seja, as informações a respeito das classes. Sendo assim, os atributos, métodos, interfaces e superclasse são metadados relacionados à classe. Portanto, trabalhar com reflexão é, na verdade, mexer com os metadados do programa, utilizando-os para criar algoritmos que trabalham com uma classe que não conhecem previamente.

A questão é que somente os metadados da própria classe muitas vezes não são suficientes para um componente que utiliza reflexão. Podem ser necessárias mais informações para saber a que um atributo está se referindo, quando um método precisa ser invocado e como uma determinada classe deve ser utilizada pelo componente.

Este capítulo não irá falar sobre os metadados da própria classe, que foram vistos no capítulo anterior, mas de como definir metadados adicionais para serem utilizados para diferenciar uns elementos de um programa dos outros. Esses metadados são chamados de metadados específicos de domínio, pois eles possuem uma semântica específica para serem lidos e tratados por um componente com um determinado objetivo.

Aqui veremos esse tipo de metadados, as formas que eles podem ser definidos e as vantagens e desvantagens de cada abordagem. Adicionalmente, será dada uma ênfase nas anotações, que é um mecanismo da linguagem Java para a definição de metadados diretamente no código. Será mostrado também como definir novas anotações, inseri-las nos elementos de código e consumi-las em tempo de execução através da API Reflection.

3.1 DEFINIÇÃO DE METADADOS

"Se alguma coisa irá variar de forma previsível, guarde a descrição dessa variação em um banco de dados de forma que seja fácil de mudar." - Ralph Johnson

No contexto da orientação a objetos, os metadados são informações sobre os elementos do código. Eles podem ser definidos em qualquer meio, bastando que o software ou componente as recupere e os utilize para agregar novas informações nos elementos do código. Esta seção irá abordar diferentes formas de definição de metadados, falando sobre as vantagens e desvantagens de cada uma delas.

Convenções de código

A primeira forma de definição de metadados que acabamos utilizando sem perceber muito quando começamos a usar reflexão são as convenções de código. Essa estratégia faz uso das próprias construções da linguagem para criar uma semântica particular a ser utilizada pelo componente. Em outras palavras, a presença de determinados elementos, como um padrão de nome, um tipo de retorno ou a implementação de uma interface, passa a ter um significado especial para o componente. A partir dessas informações, é possível fazer a distinção desses elementos, tratando-os de forma diferente dos outros.

Os próprios exemplos que vimos nos capítulos anteriores acabam utilizando convenções de código de alguma forma. O componente que executa as rotinas de validação considera o prefixo `validar` como forma de identificar os métodos de validação e a implementação da interface `Validador` para

identificação dos atributos com objetos que precisam ser executados. Um outro padrão de código amplamente utilizado pelos componentes é o Java Beans (JAVA COMMUNITY PROCESS, 1997), que define que métodos de recuperação de informações devem começar com `get` e métodos para inserir valores em propriedades devem ser começados com `set`, conforme ilustrado na próxima listagem. Essa convenção foi utilizada no exemplo da geração do mapa de propriedades e é usada em diversos outros componentes e frameworks.

Exemplo de convenção de código de um Java Bean:

```
public class JavaBean {
    private String propriedade;
    public String getPropriedade(){
        return propriedade;
    }
    public void setPropriedade(String s){
        this.propriedade = s;
    }
}
```

Um outro exemplo de convenção de código muito utilizada foi o prefixo `test` no JUnit 3. Nessa versão e nas anteriores, um método com esse prefixo, como ilustrado na próxima listagem era considerado um método de teste e invocado pelo framework. Em versões posteriores, o uso desse prefixo foi substituído por uma anotação.

Definição de um teste no JUnit 3:

```
public class TesteSomador extends TestCase {

    //identificado como teste pelo prefixo "test"
    public void testSoma(){
        Somador s = new Somador();
        int resultado = s.somar(2,2,4);
```

```
        assertEquals(8,resultado);
    }
}
```

Porém, não existem somente convenções que utilizam prefixos e sufixos nos nomes. Um exemplo comum é usar o próprio nome do elemento de código para que ele corresponda ao nome utilizado em alguma outra entidade que ele está mapeando. No exemplo do gerador de mapa de propriedades, o nome do método foi utilizado para gerar o nome da propriedade. De forma análoga, é possível mapear o nome de uma classe para o nome de uma tabela no banco de dados que ela representa ou para uma URL de uma aplicação web que ela precisa tratar.

As interfaces de marcação foram uma prática comum para definir um metadado antes de Java ter suporte nativo a anotações. A interface de marcação é uma interface sem métodos, a qual as classes implementam para terem um significado especial para algum componente. Um exemplo disso é a `Serializable` , que marca as classes que são serializáveis. Ela não possui métodos e é adicionada à classe apenas com o intuito de marcá-la como tendo uma determinada propriedade. A próxima listagem mostra uma classe válida que implementa essa interface, não precisando implementar nenhum método. Em outras palavras, essa interface serve mais para adicionar um metadado do que para realmente representar uma abstração.

Código válido que implementa a interface Serializable:

```
public class Serializável implements Serializable {
    public int numero;
    public String texto;
    //sem métodos omitidos
}
```

Uma das características das convenções de código é utilizar mecanismos da própria linguagem para a definição dos metadados. O lado positivo disso é que os desenvolvedores não precisam adicionar nada para que o componente entenda os metadados daquela classe, bastando seguir as convenções na definição dos nomes e elementos do código. O lado ruim dessa característica é que os metadados ficam, de certa forma, implícitos e precisam ser compartilhados por toda a equipe para seu uso ser efetivo. Por exemplo, pode não ser claro para um desenvolvedor que a mudança de nome de um método pode causar uma mudança de comportamento em um componente que invoca aquela classe.

Devido à facilidade do uso de convenções de código, essa prática é bastante utilizada por frameworks e APIs, porém essa é raramente uma técnica que é utilizada sozinha. Um dos motivos é que a expressividade dessa técnica é bastante limitada e apenas metadados simples conseguem ser representados dessa forma. Sendo assim, o que costumam ser utilizadas são algumas convenções de código que são complementadas com alguma das outras formas de definição de metadados. No exemplo da geração do mapa de propriedades, uma convenção de código considerava que o nome da chave do mapa seria igual ao nome utilizado no método getter, porém uma anotação permitia que se usasse um nome diferente.

Definição programática

Uma outra forma de definição de metadados é através de uma rotina que insere as informações relativas aos metadados diretamente ao componente. Seria algo como a chamada de um método que inserisse a informação de que uma classe possui uma

determinada informação, ou que algum de seus métodos possuem um dado associado. Essa abordagem tira a responsabilidade do programa de ler os metadados, pois é o programa que deve inseri-los. Em um programa que use esse componente, a rotina de inserção desses metadados deve ocorrer antes que esse componente utilize uma classe, em algum código que normalmente é colocado na inicialização da aplicação.

Não existem muitos exemplos de frameworks e APIs que fazem uso dessa abordagem. Do ponto de vista de quem utiliza o componente, pessoalmente não acho muito intuitivo utilizar um código imperativo para configurar informações que são estáticas. Um framework brasileiro que usa esse tipo de técnica é o Mentawai.

Na listagem a seguir está um exemplo obtido do tutorial desse framework que faz o mapeamento de uma classe para a base de dados. Neste exemplo, a classe `Usuario` é mapeada para a tabela `Usuarios` e seus atributos são mapeados para as tabelas. Cada chamada do método `field()` passa como parâmetro o nome do atributo, opcionalmente o nome da coluna caso seja diferente do atributo e o tipo do campo no banco de dados. Apesar de serem chamadas de método, é possível perceber que eles estão definindo informações a respeito dos elementos da classe.

Exemplo de configuração programática de metadados de persistência no framework Mentawai:

```
@Override
public void loadBeans() {
    bean(Usuario.class, "Usuarios")
        .pk("id", DBTypes.AUTOINCREMENT)
        .field("login", DBTypes.STRING)
        .field("senha", DBTypes.STRING)
```

```
        .field("dataNascimento", "nascimento", DBTypes.DATE);
}
```

Apesar de essa definição programática de metadados não ser diretamente utilizada pelos desenvolvedores, ela acaba existindo internamente em diversos componentes e frameworks. Isso acontece porque se houver a funcionalidade de ler os metadados definidos de alguma outra forma, a classe que faz isso precisará dessa interface programática para poder inseri-los.

Fontes externas

Outra forma de definir os metadados é utilizando fontes externas de dados. Dessa maneira, o componente ou framework deve acessar essa fonte externa de dados e realizar a leitura dessas informações. De alguma forma, elas devem referenciar os elementos de código, como classes, métodos e atributos, para que quem realizar essa leitura possa referenciar a metainformação a quem ela pertence. Essa leitura também deve ser feita antes que o componente seja utilizado, pois nesse momento essas informações já devem estar presentes.

Uma das formas mais comuns de se definir metadados em uma fonte externa é através de arquivos de configuração, sendo que arquivos XML acabam sendo os mais utilizados. Eles costumam possuir um formato bem definido para poderem ser lidos pelos componentes. Apesar de XML ser considerado um formato de arquivo "amigável para pessoas", muitas vezes a configuração das informações é complicada e tediosa. O fato de as informações precisarem referenciar os elementos do código faz com que o arquivo acabe ficando verboso, sendo que, como são raros os casos em que existe algum suporte de ferramenta para trabalhar com ele,

é muito fácil cometer um erro em um nome de classe ou método.

Outra alternativa para armazenar os metadados são os bancos de dados. Se eles podem armazenar dados da aplicação, por que não armazenar dados sobre a aplicação? Apesar de essa não ser uma alternativa muito utilizada por frameworks de mercado devido à dificuldade de configuração inicial, em componentes caseiros, ou seja, desenvolvidos dentro da aplicação, essa acaba sendo uma alternativa bastante utilizada. Principalmente para metainformações que podem ser alteradas em tempo de execução, como permissões de segurança de métodos de negócio, ter um controle transacional e centralizado desses dados pode ser uma boa ideia.

Para exemplificar a configuração dos metadados em fontes externas ao código, a próxima listagem traz os metadados de um mapeamento objeto-relacional feito pelo framework Hibernate. Observe que o elemento `<class>` referencia a classe `com.casadocodigo.Usuario` e, em seguida, os elementos `<property>` referenciam os atributos, associando-os à respectiva coluna na base de dados. Os metadados representados aqui em XML possuem o mesmo objetivo dos definidos de forma programática na listagem anterior.

Exemplo de definição de metadados de persistência em arquivos XML:

```
<hibernate-mapping>
     <class name="com.casadocodigo.Usuario" table="USUARIO">
          <id column="USER_ID" name="id" type="java.lang.Long">
               <generator
             class="org.hibernate.id.TableHiLoGenerator">
                    <param name="table">idgen</param>
               <param name="column">NEXT</param>
```

```
            </generator>
        </id>
        <property column="LOGIN" name="login"
          type="java.lang.String"/>
      <property column="SENHA" name="senha"
          type="java.lang.String"/>
      <property column="NASCIMENTO" name="dataNascimento"
          type="java.util.Date"/>
    </class>
</hibernate-mapping>
```

Uma das desvantagens da definição de metadados em fontes externas é que os dados precisam referenciar os elementos do código tornando essa configuração tediosa e sujeita a erros, principalmente se não houver uma ferramenta de apoio. No uso dessa abordagem acaba havendo uma distância entre os elementos do código e os metadados. Isso causa dificuldades na sua configuração e, principalmente, quando for necessária, em alguma manutenção. Uma alteração inofensiva nos elementos de uma classe pode quebrar uma referência nos arquivos de configuração e gerar um bug indesejado na aplicação.

Os metadados, de uma certa forma, acabam sendo responsáveis por parte do comportamento da aplicação. Desse modo, eles devem ser versionados junto com a aplicação para poderem gerar o comportamento desejado. Para documentos XML isso não é um grande problema, mas para bancos de dados pode ser algo bastante complicado. Principalmente quando os usuários de alguma maneira podem modificar os metadados e é preciso fazer uma junção ou adaptação dos metadados devido a novas classes ou modificações em classes existentes.

Por outro lado, a definição de metadados em fontes externas é uma boa alternativa quando é preciso dar suporte à configuração de metadados para classes de terceiros e dos quais não se tem

acesso de modificação do código-fonte. Nesse caso, essa desvinculação entre os metadados e as classes é uma vantagem. Adicionalmente, essa abordagem também é adequada para permitir que os metadados possam ser alterados sem a necessidade de recompilação da aplicação. Isso é ideal para quando são necessários ajustes nos metadados em tempo de deploy ou em tempo de execução.

Anotações

As anotações são um recurso da linguagem Java introduzidas na JDK 5 para permitir a adição de metadados diretamente no código. Essa prática também é conhecida como programação orientada a atributos. A partir desse recurso, é possível manter o código e os metadados no mesmo local, simplificando sua administração. Essa abordagem acaba ainda diminuindo a verbosidade de sua definição, visto que, como os metadados são adicionados nos próprios elementos de código, não é necessário referenciá-los. Esse recurso de linguagem também está presente em outras linguagens como no C#, onde é chamado de *attributes*. Python possui um mecanismo um pouco diferente chamado de *decorator*, mas que também é utilizado para a adição de metadados diretamente no código.

Na linguagem Java, a programação orientada a atributos começou antes das anotações, com a ferramenta XDoclet. No Java EE 1.4, que na época ainda recebia o nome de J2EE, a criação de EJBs demandava a criação de diversos descritores XML e interfaces. Eu cheguei a desenvolver para essa plataforma e posso afirmar que realmente era muito complicado desenvolver sem uma boa ferramenta de apoio. Nessa época, uma alternativa que surgiu

para o desenvolvimento de EJBs foi a utilização da ferramenta XDoclet.

O XDoclet é uma engine de geração de código que processa o código-fonte permitindo a adição de metadados em tags JavaDoc, que são adicionadas dentro de comentários. Como resultado desse processamento, é possível gerar descritores XML e o código-fonte de outras classes. Existia um grande conjunto de tags, por exemplo, que eram destinadas ao desenvolvimento para a plataforma J2EE. No caso de um EJB de sessão, por exemplo, eram geradas as interfaces auxiliares e os descritores XML em que aquela classe deveria estar presente. A listagem a seguir mostra o exemplo de uma classe anotada com as tags JavaDoc do XDoclet.

Utilização do XDoclet para geração de EJBs:

```
/**
 * @ejb.bean
 *  name="CustomerService"
 *  jndi-name="CustomerServiceBean"
 *  type="Stateless"
**/
public abstract class CustomerServiceBean implements
 SessionBean {

  /**
   * @ejb.interface-method tview-type="both"
   */
  public void createCustomer(CustomerVO customer) {
  }

  /**
   * @ejb.interface-method tview-type="both"
   */
  public void updateCustomer(CustomerVO customer) {
  }
}
```

Não sei exatamente qual foi a motivação para a introdução das

anotações no Java 5, mas existiam diversas forças que acabaram empurrando a linguagem nessa direção. Por um lado, havia uma grande critica devido à grande quantidade de descritores do Java EE e a programação orientada a atributos com o XDoclet se mostrava como uma alternativa viável. Por outro lado, a linguagem C# já possuía o mecanismo de atributos e este vinha sendo utilizado com sucesso para o desenvolvimento de aplicações corporativas.

As anotações foram anunciadas na linguagem Java como o recurso que iria simplificar o desenvolvimento de aplicações corporativas. Hoje é possível ver que realmente elas conseguiram cumprir essa promessa, eliminando a obrigatoriedade de diversos descritores XML que tornavam o desenvolvimento desse tipo de aplicação bastante burocrático. Atualmente, grande parte dos frameworks e APIs para a linguagem Java faz o uso de anotações. Esse recurso de linguagem acabou popularizando a utilização de frameworks baseados em metadados, pois facilitou a definição dos metadados para os seus usuários, tornando essa abordagem mais viável.

Para exemplificar a utilização de anotações para definição de metadados, considere a listagem a seguir, que mostra como o mapeamento para um banco de dados é feito com as anotações da API JPA. Observe que nada precisou ser feito para os atributos `login` e `senha`, que nos exemplos anteriores precisaram ser referenciados explicitamente na definição de metadados. A anotação `@Table`, por exemplo, só precisa ser adicionada caso o nome da tabela seja diferente do nome da classe. O mesmo se aplica à anotação `@Column`, que também só é necessária quando a coluna tem o nome diferente do atributo.

Mapeamento objeto-relacional com JPA utilizando anotações:

```
@Entity
@Table(name="Usuarios")
public class Usuario {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

     private String login;
     private String senha;

     @Column(name="nascimento")
     private Date dataNascimento;

    //métodos get e set omitidos
}
```

Apesar de serem mais fáceis de configurar por ficarem próximas aos elementos de código, essa própria característica também pode ser uma desvantagem das anotações. Quando elas são utilizadas, para a alteração dos metadados é necessário recompilar as classes, o que pode não ser viável para ajustes em tempo de implantação ou alteração em tempo de execução. Isso também pode dificultar a reutilização da classe em um contexto em que as anotações não sejam necessárias ou em que outros metadados precisem ser configurados.

O uso de anotações nas classes talvez seja familiar a grande parte dos desenvolvedores, porém a criação de novas anotações e o desenvolvimento de componentes que recuperam e fazem processamentos baseados nelas talvez não sejam tão populares. O resto deste capítulo se dedica a mostrar como as anotações podem ser criadas e como utilizar a API Reflection para recuperá-las.

Misturando mecanismos de definição de metadados

Como pode ser visto, cada forma de definição de metadados possui suas vantagens e desvantagens, e muitas vezes fica difícil equilibrar os requisitos a partir de uma das formas. Por exemplo, enquanto as anotações oferecem uma forma menos verbosa de definir os metadados, as fontes de dados externas permitem a alteração dos metadados sem a recompilação das classes. Felizmente, os componentes não precisam escolher apenas uma forma de definição de metadados, sendo possível combinar metadados de diferentes fontes. O capítulo *Consumindo e processando metadados* fala como estruturar o componente para permitir a combinação de metadados de diferentes fontes.

3.2 CRIANDO ANOTAÇÕES

As anotações em Java são definidas em arquivos separados com extensão `.java` que precisam ter seu mesmo nome, assim como as classes, as interfaces e as enumerações. Para definir uma nova anotação, é preciso utilizar a palavra chave `@interface`. A listagem a seguir mostra um exemplo simples de como se definir uma anotação.

Definição de uma simples anotação chamada Metadado:

```
public @interface Metadado {
}
```

Uma anotação pode possuir atributos, ou seja, você pode

agregar informações aos metadados, porém não é qualquer classe que é aceita como tipo do atributo. Uma anotação não pode possuir comportamento, sendo uma informação estática que é adicionada a classes. Existem outras que podem ser adicionadas às próprias anotações para parametrizar como elas devem ser tratadas pelo compilador e pela máquina virtual. As próximas subseções irão apresentar detalhes a respeito de como elas são criadas.

Uma coisa importante das anotações é: **elas não fazem nada!!!** Para quem utiliza um framework que faz uso de anotações, pode parecer que a configuração de anotações adiciona comportamentos na classe, porém a verdade é que elas somente estão agregando novas informações sobre os elementos da classe. Para que elas tenham algum efeito sobre o funcionamento do programa, alguém precisa recuperá-las e fazer alguma coisa a respeito, pois sozinhas as anotações não fazem nada.

Definindo até quando a anotação está disponível

Uma das principais configurações de uma anotação é até quando ela vai estar disponível para recuperação. Dependendo do uso que será feito dela e do momento em que essa informação será consumida, diferentes tipos de retenção podem ser necessários. Segue uma lista com os três diferentes tipos de retenção que uma anotação pode possuir:

- `SOURCE` : uma anotação com esse tipo de retenção fica disponível apenas no código-fonte. No momento em que a classe é compilada, a anotação não é transferida para o arquivo `.class` . Esse tipo de anotação é normalmente utilizado para fins de documentação e

para uso de ferramentas que fazem processamento direto de código-fonte. Um exemplo são ferramentas que fazem validações em tempo de compilação.

- `CLASS` : anotações com esse tipo de retenção são mantidas nos arquivos `.class` , porém não são carregadas pela máquina virtual. Nesse caso, elas ficam disponíveis até o momento do carregamento da classe. Esse tipo de anotação é utilizado por ferramentas que fazem processamento do bytecode da classe, podendo ser feito de forma estática como uma etapa posterior à compilação, ou no momento do carregamento das classes. Essa prática será vista no capítulo *Manipulação de bytecode*.

- `RUNTIME` : para uma anotação estar disponível para a recuperação em tempo de execução, ela precisa ter esse tipo de retenção. Nesse caso, a máquina virtual vai carregá-la em memória e torná-la acessível através da API Reflection. Esse tipo de anotação é o tipo utilizado por frameworks que precisam ter acesso às anotações em tempo de execução. Será o tipo que iremos utilizar com maior frequência aqui neste livro.

Para configurar o tipo de retenção de uma anotação, é preciso anotá-la com `@Retention` . Ela recebe como parâmetro uma enumeração do tipo `RetentionPolicy` , que pode possuir os valores `SOURCE` , `CLASS` ou `RUNTIME` descritos anteriormente. A listagem a seguir exemplifica como realizar essa configuração. Neste capítulo iremos abordar somente as anotações com retenção do tipo `RUNTIME` , ficando as do tipo `CLASS` para o capítulo

Manipulação de bytecode. O processamento de anotações diretamente no código-fonte como parte do processo de compilação, que utilizaria anotações do tipo `SOURCE`, está fora do escopo do livro.

Configurando o tipo de retenção de uma anotação:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Metadado {
}
```

Caso uma anotação não possua uma retenção configurada, por definição ela será do tipo `SOURCE`. Sendo assim, se você estiver desenvolvendo um código que consome uma anotação e por algum motivo ela não estiver sendo encontrada, muito provavelmente é porque você esqueceu de configurar a retenção para `RUNTIME`. Eu já criei diversos frameworks que consomem anotações, e mesmo assim de tempos em tempos isso acaba acontecendo comigo.

Tipo de elementos anotados

Outra configuração importante para uma anotação são os tipos de elementos que podem ser anotados por ela. Se nada for dito, a anotação poderá ser adicionada em qualquer tipo de elemento. Por mais que essa configuração não seja obrigatória, é sempre uma boa prática adicioná-la para evitar que os seus usuários a adicionem no local errado. Por exemplo, uma anotação que deve ser adicionada nos métodos de acesso pode ser adicionada por engano em um atributo, o que vai fazer com que não seja encontrada pela classe que a procurar.

Segue uma lista com os tipos de elemento que podem ser anotados, sendo cada um deles um elemento da enumeração

`ElementType` :

- `TYPE` : qualquer definição de tipo, como classes, interfaces e enumerações;
- `PACKAGE` : pacotes;
- `CONSTRUCTOR` : construtores;
- `FIELD` : atributos;
- `METHOD` : métodos;
- `PARAMETER` : parâmetros de métodos e construtores;
- `LOCAL_VARIABLE` : variáveis locais;
- `ANNOTATION_TYPE` : anotações.

Para configurar o tipo de elemento que uma anotação pode anotar, é preciso configurá-la com a anotação `@Target` . Ela pode receber um array com diversos tipos onde ela faz sentido ser adicionada. A listagem a seguir mostra um exemplo de configuração, onde a anotação pode ser adicionada em um método ou em um atributo.

Configurando o tipo de elemento que pode ser anotado:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.FIELD})
public @interface Metadado {
}
```

Em relação ao tipo `LOCAL_VARIABLE` , como as variáveis locais não são acessíveis por reflexão, esse tipo de anotação só faz sentido com retenção do tipo `SOURCE` ou `CLASS` . Outro tipo curioso é a

anotação do tipo `PACKAGE`, que é utilizada para adicionar metainformações em um pacote. Para adicioná-la, é preciso criar um arquivo no pacote chamado `package-info.java` para definir a anotação. A listagem a seguir mostra como seria o conteúdo desse arquivo.

Conteúdo do arquivo package-info.java para adicionar anotação ao pacote:

```
@Metadado
package org.casadocodigo;

import org.anotacoes.Metadado;
```

Outras definições

Além das anotações `@Target` e `@Retention` existem outras que podem ser adicionadas para configurar como as ferramentas da JDK devem encará-las. Uma delas é a `@Documented`, que deve ser utilizada quando a anotação precisar ser incorporada à documentação dos elementos anotados. Por exemplo, uma anotação que determina os valores válidos para um atributo certamente é interessante de ser adicionada na documentação. Outras anotações muito específicas que, por exemplo, marcam um método para ter suas invocações incluídas no log, talvez não precisem entrar na documentação.

Outra anotação que pode ser adicionada na anotação é `@Inherited`. Nesse caso, ela será propagada para as subclasses da classe que a possuir. Uma nota importante é que isso só irá funcionar com anotações com `@Target` do tipo `TYPE`. Ao contrário do que se esperaria, um método sobrescrito por uma subclasse não irá herdar as anotações do tipo `@Inherited` do

método que está sobrescrevendo.

Atributos da anotação

Em alguns casos, apenas a presença da anotação já é suficiente para configurar uma propriedade do elemento anotado, como se a classe é persistente ou não. Porém, em outros casos, são necessárias informações adicionais, como por exemplo qual o nome da tabela do banco de dados para a qual essa classe é mapeada. Sendo assim, as anotações podem possuir atributos para a configuração de valores que fazem parte do metadado. Diferentemente de um atributo de classe, apenas alguns tipos podem ser tipos de atributos de uma anotação. Esses são tipos primitivos, enums, `Class`, `String`, outras anotações e arrays de qualquer um desses tipos.

Todo atributo de uma anotação precisa possuir um valor. Dessa forma, se o atributo estiver presente, ele precisa de um valor. É possível configurar um valor *default* para o atributo e quando isso é feito, aquele valor não precisa obrigatoriamente ter um valor atribuído quando a anotação é utilizada. Os atributos são configurados através de pares nome e valor adicionados na anotação no seguinte formato: `@Anotacao(nome=valor)`. Existe um nome especial de atributo, `value`, que, caso seja o único atributo configurado da anotação, pode ter seu nome omitido da seguinte forma: `@Anotacao(valor)`.

Para entender melhor o que é válido e o que não é, nada melhor do que ver alguns exemplos. A listagem a seguir, mostra uma anotação com um atributo chamado `value`. Ao definir a anotação sem o atributo em `atributo1`, isso é inválido pois o valor precisa ser definido obrigatoriamente. Como o atributo da

anotação se chama `value`, então ele pode ser explicitamente referenciado ou não, respectivamente na forma `@Metadado(value=13)` ou `@Metadado(13)`, sendo as duas válidas.

Exemplo de anotação com atributo value:

```
//definição da anotação
public @interface Metadado {
    int value();
}

//Exemplos de uso
public class Exemplo{

    @Metadado //inválido pois não definiu atributo
    public String atributo1;

    @Metadado(value=13) //válido
    public String atributo2;

    @Metadado(13) //válido
    public String atributo3;
}
```

O exemplo apresentado na próxima listagem mostra uma anotação com o atributo `tipo`, que possui a definição de um valor *default* igual a uma `String` vazia. Nesse caso, a definição da anotação da forma `@Metadado`, sem definição de um valor para o atributo, é válida porque um valor foi definido na própria anotação. Para ela, a omissão do nome do atributo é inválida quando ele é configurado, pois o nome do atributo não é `value`.

Exemplo de anotação com atributo value e valor default:

```
//definição da anotação
public @interface Metadado {
    String tipo() default "";
}
```

```
//Exemplos de uso
public class Exemplo{

    @Metadado //válido
    public String atributo1;

    @Metadado(tipo="OK") //válido
    public String atributo2;

    @Metadado("OK") //inválido porque o atributo não chama value
    public String atributo3;
}
```

Como último exemplo de definição de atributos, a próxima listagem mostra uma anotação com um atributo `value` e outro chamado `tipo` com valor *default*. A anotação em `atributo1` é inválida pois `value` precisa ter um valor definido e a anotação em `atributo2` é válida, porque, quando somente o atributo `value` é definido, seu nome pode ser omitido. Nos dois exemplos de uso seguintes, podemos ver que `value` só pode ser omitido quando é o único atributo definido na anotação. Sendo assim, quando `tipo` é definido, o nome do atributo `value` precisa ser definido explicitamente.

Exemplo de anotação com atributo value e valor default:

```
//definição da anotação
public @interface Metadado {
    Class value();
    String tipo() default "";
}

//Exemplos de uso
public class Exemplo{

    @Metadado //inválido porque value não tem valor default
    public String atributo1;

    @Metadado(String.class) //válido
```

```
    public String atributo2;

    @Metadado(value=String.class, tipo="OK") //válido
    public String atributo3;

    @Metadado(String.class, tipo="OK")
    //inválido pois value só pode ser omitido quando é o único
    //atributo
    public String atributo4;
}
```

3.3 LENDO ANOTAÇÕES EM TEMPO DE EXECUÇÃO

Depois de ver na seção anterior como criar anotações, esta seção mostra como recuperá-las através da API Reflection. Mais uma vez ressalto que para isso ser possível é preciso que a anotação possua a definição `@Retention` com o valor `RUNTIME`, pois, caso contrário, a máquina virtual não irá carregar a definição do metadado junto com a classe.

Recuperando anotações

Existe uma interface chamada `AnnotatedElement`, que define os métodos para a recuperação de anotações. Essa interface é implementada por todas as classes que representam elementos de código que podem ser anotados, como as classes `Class`, `Method`, `Field`, `Package` e `Constructor`. Sendo assim, esses métodos que serão apresentados podem ser invocados em qualquer uma dessas classes.

Existem dois métodos nessa interface que nos permitem trabalhar com uma anotação específica. O primeiro deles é `isAnnotationPresent()`, que pode ser utilizado para verificar se

uma determinada anotação está presente ou não no elemento. Esse método retorna um valor booleano dizendo se ela existe naquele elemento. O outro é `getAnnotation()`, que recebe como parâmetro a classe da anotação e retorna a anotação desejada. A instância retornada possui o tipo da anotação, e seus atributos podem ser recuperados através dos métodos com seus nomes.

A listagem a seguir mostra um exemplo de código onde uma anotação é recuperada e seus atributos impressos no console. Antes de imprimir a anotação, o método `isAnnotationPresent()` é utilizado para confirmar sua presença. Observe que, quando o método `getAnnotation()` é invocado, ele recebe a classe da anotação como parâmetro e utiliza seu tipo genérico para inferir o tipo do retorno, que é o próprio tipo da anotação. A partir dessa instância com o tipo, é possível obter os seus atributos invocando métodos com seus nomes, conforme mostrado no exemplo.

Recuperação da anotação:

```
//definição da anotação
@Retention(RetentionPolicy.RUNTIME)
public @interface Metadado {
    String nome();
    int numero();
}

//Uso e recuperação de anotação
@Metadado(nome="classe", numero=17)
public class RecuperaAnotacao {

    public static void main(String[] args) {
        Class<RecuperaAnotacao> c = RecuperaAnotacao.class;
        if(c.isAnnotationPresent(Metadado.class)){
            Metadado m = c.getAnnotation(Metadado.class);
            System.out.println("Propriedade nome = "+ m.nome());
            System.out.println("Propriedade numero ="
```

```
                                + m.numero());
        }
    }
}
```

Os outros métodos definidos em `AnnotatedElement` para recuperação de anotações retornam a lista de anotações de um elemento. O método `getAnnotations()` irá retornar uma lista com todas as anotações de um determinado elemento de código e o método `getDeclaredAnnotations()` irá retornar somente as declaradas no elemento, excluindo as com `@Inherited` que foram herdadas da superclasse. Como a herança de anotações funciona somente para classes, para outros tipos de elemento o retorno dos dois métodos será sempre o mesmo.

Anotações em parâmetros

As anotações em parâmetros são recuperadas de forma diferente, pois não existe uma classe na API Reflection que represente um parâmetro. O método para a recuperação de anotações em parâmetros se chama `getParameterAnnotations()` e está presente nas classes `Method` e `Constructor`. Essa chamada retorna um array de duas dimensões de `Annotation`, sendo que a primeira representa os parâmetros, e a segunda, as anotações daquele parâmetro.

As anotações em parâmetros são muito úteis para identificar qual informação precisa ser passada para ele. Dessa forma, quando um método for invocado utilizando reflexão, essa informação pode ser utilizada para descobrir o que se espera receber. Nos exemplos que foram vistos até agora, eram utilizadas convenções para determinar quais seriam os parâmetros dos métodos invocados, porém com essas informações é possível lidar com diversas

combinações de parâmetros, reconhecendo-os em tempo de execução. Como mostrado na seção *Acessando nomes de parâmetros*, no Java 8 ainda é possível recuperar os nomes dos parâmetros, o que também pode ser um metadado valioso para esse tipo de processamento.

Para exemplificar esse procedimento, a próxima listagem apresenta a definição de uma anotação de parâmetros. Essa anotação se chama `@Param` e o objetivo é que um nome para o parâmetro seja definido em seu atributo `value`.

Anotação para nomear os parâmetros:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.PARAMETER)
public @interface Param {
    String value();
}
```

Na próxima listagem está um método chamado `invocarMetodo()`, que recebe um `Method`, um objeto e um mapa de informações como parâmetro, e executa o método recuperando os seus argumentos do mapa de acordo com o nome configurado na anotação. Observe que ele inicialmente recupera as anotações dos parâmetros pelo método `getParameterAnnotations()` e, em seguida, cria um array chamado `paramValues` para armazenar os valores a serem passados na chamada do método. Para cada parâmetro, o método auxiliar `getNomeParâmetro()` recebe a lista de anotações daquele parâmetro e busca entre elas a anotação `@Param`, retornando o seu valor. Depois, é recuperado do mapa o objeto com a chave tendo o mesmo nome configurado na anotação. Após recuperar todos os parâmetros, o método é invocado passando o array

resultante.

Invocação de método recuperando os parâmetros de um mapa:

```
public static Object invocarMetodo(Method m, Object obj,
        Map<String,Object> info) throws Exception{
    Annotation[][] paramAnnot = m.getParameterAnnotations();
    Object[] paramValues = new Object[paramAnnot.length];
    for(int i=0; i<paramValues.length; i++){
        String name = getNomeParametro(paramAnnot[i]);
        paramValues[i] = info.get(name);
    }
    return m.invoke(obj, paramValues);
}
public static String getNomeParametro(Annotation[] ans){
    for(Annotation a : ans){
        if(a instanceof Param){
            return ((Param)a).value();
        }
    }
    throw new RuntimeException(
        "Anotação @Param não encontrada");
}
```

A listagem a seguir mostra um exemplo de uso desse método. Um método chamado `metodo` com dois parâmetros anotados foi criado para testarmos a invocação. No método `main()`, inicialmente é criado um mapa com diversos valores para a recuperação dos parâmetros. Em seguida, um referência para `metodo` é recuperada e o método `invocarMetodo()` é invocado passando-o junto com a instância da classe e o mapa.

Testando a invocação de métodos com parâmetros nomeados:

```
public class AnotacaoParametro {

    public static void main(String[] args) throws Exception {
        Map<String,Object> info = new HashMap<>();
```

```
        info.put("inteiro", 13);
        info.put("numero", 23);
        info.put("string", "OK");
        info.put("texto", "NOK");

        AnotacaoParametro ap = new AnotacaoParametro();
        Method m = ap.getClass().getMethod(
            "metodo", Integer.class, String.class);
        invocarMetodo(m, ap, info);
    }
    public void metodo(
            @Param("inteiro") Integer i,
            @Param("texto") String s){
        System.out.println("Parametro inteiro = "+i);
        System.out.println("Parametro texto = "+s);
    }
}
```

Deixo como observação desse exemplo que muitas coisas podem dar errado caso o tipo do objeto recuperado do mapa não seja o mesmo do esperado pelo parâmetro. Fica como exercício ao leitor complementar esse exemplo, verificando se os tipos são compatíveis e fazendo uma conversão quando possível. Como saber como a conversão deve ser feita? Tente utilizar uma anotação no próprio parâmetro para indicar métodos ou classes que devem ser utilizados para fazer essa conversão.

Lendo atributos de uma anotação com reflexão

Da mesma forma que uma classe pode ser lida através de reflexão sem a conhecermos previamente, também é possível obtermos a classe relativa à anotação para descobrirmos seus atributos em tempo de compilação. A principal diferença é que, para obtermos instância de `Class` que representa a anotação, devemos chamar o método `annotationType()` em vez de `getClass()`. Em seguida, os atributos da anotação podem ser

recuperados através dos métodos que são utilizados para retorná-los.

Para exemplificar esse processo, a listagem a seguir mostra um código que recupera as anotações de um elemento e imprime no console com seus atributos. Observe que o método recebe um `AnnotatedElement` como parâmetro, permitindo que receba qualquer classe que represente um elemento que possa ter anotações. Ao recuperar a instância de `Class` referente à anotação utilizando `annotationType()`, o método imprime o nome da anotação e em seguida itera pelos seus métodos imprimindo o nome e o valor retornado. Observe que utilizando `getDeclaredMethods()` recuperamos somente os métodos declarados na anotação, ou seja, os que definem atributos.

Método que imprime os atributos das anotações de um elemento:

```
public static void imprimeAnotacoes(AnnotatedElement ae)
                                                throws Exception{
    Annotation[] ans = ae.getAnnotations();
    for(Annotation a : ans){
        Class<?> c = a.annotationType();
        System.out.println("@"+c.getName());
        for(Method m : c.getDeclaredMethods()){
            Object o = m.invoke(a);
            System.out.println(" |->"+m.getName()+"="+o);
        }
    }
}
```

A listagem a seguir mostra um exemplo de uma classe com duas anotações, para a qual a respectiva instância de `Class` é passada para o método `imprimeAnotacoes()`. Em seguida, é mostrada a saída do console gerada por esse programa, na qual as duas anotações e seus atributos são impressos.

Anotação para nomear os parâmetros:

```
@Metadado(nome="teste",numero=34)
@Anotacao(String.class)
public class ImprimeAnotacoes {

    public static void main(String[] args) throws Exception {
        imprimeAnotacoes(ImprimeAnotacoes.class);
    }
}
```

Saída do programa que imprime as anotações:

```
@Metadado
 |->nome=teste
 |->numero=34
@Anotacao
 |->value=class java.lang.String
```

3.4 LIMITAÇÕES DAS ANOTAÇÕES

"Aceite essas limitações e se submeta a elas em vez de continuar insistindo em fazer a sua vontade." - Emerson Natal

Durante este capítulo muita coisa foi dita a respeito do que se pode fazer com as anotações. Esta seção fala um pouco sobre as limitações dessa funcionalidade da linguagem, mostrando o que não se consegue fazer. No capítulo *Práticas no uso de anotações* serão abordadas algumas técnicas que são frequentemente utilizadas para contornar isso.

A primeira limitação é que só pode haver uma anotação de um determinado tipo para um elemento. Imagine, por exemplo, que uma anotação aponte uma classe para fazer uma validação de seus dados. A partir dessa anotação, não seria possível configurar duas classes para fazer a validação, pois somente uma anotação daquele

tipo pode ser adicionada. Para que isso seja possível, uma alternativa seria a própria anotação dar suporte à configuração de diversas classes. Outra alternativa frequentemente utilizada é uma outra anotação que possua como `value` um array da anotação original, permitindo agregar diversas anotações do tipo anterior, como mostrado na seção *Mais de uma anotação do mesmo tipo*. Vale ressaltar que essa questão foi tratada no Java 8, o que é mostrado na seção *Múltiplas anotações*.

Outra limitação das anotações é que elas não possuem nenhum mecanismo que permita generalizá-las, como herança ou algum outro tipo de abstração. Com esse mecanismo, seria possível, por exemplo, ao buscar uma anotação, procurar por todas que possuem uma determinada abstração. Porém, o principal ponto em que isso prejudica é que não é possível estender as anotações para adicionar novas semânticas e novas informações.

Por exemplo, imagine que uma anotação `@A` possua um atributo do tipo de uma anotação `@B`. Se a herança fosse possível, esse tipo `@B` poderia ser estendido por outras anotações, que poderiam ser adicionadas no lugar dele. Isso permitiria um mecanismo de metadados extensível, onde novas informações poderiam ser agregadas, como em um documento XML. Como isso não é possível, fica-se restrito às informações presentes na anotação `@B`, não sendo possível qualquer tipo de extensão.

3.5 MAPEANDO PARÂMETROS DE LINHA DE COMANDO PARA UMA CLASSE

O objetivo desta seção é mostrar um exemplo completo e realista envolvendo anotações. Um tipo de aplicação comum dos

frameworks baseados em metadados é para realizar o mapeamento entre diferentes representações de uma entidade da aplicação. Como foi mostrado no início deste capítulo, existem diversos frameworks que utilizam diferentes abordagens para mapear classes da aplicação para tabelas do banco de dados. De forma similar, existem outros frameworks que mapeiam classes para entidades de documentos XML, métodos para web services e, até mesmo, classes de diferentes aplicações que representam o mesmo conceito.

Seguindo esse tipo de aplicação, o exemplo que será desenvolvido irá mapear os parâmetros recebidos para aplicação na linha de comando para uma classe com propriedades para facilitar sua recuperação posteriormente. Os parâmetros de linha de comando são recebidos pelo método `main()` com um array de `String` na ordem em que são passados para aplicação. Usualmente, utilizam-se letras na forma `-x` para indicar que a próxima palavra será o valor daquela propriedade. O exemplo que será desenvolvido irá receber como parâmetro o array de `String` com os parâmetros e irá inserir os dados em uma instância, de acordo com as anotações de sua classe, que indicarão que atributo deve receber qual parâmetro.

A próxima listagem mostra a anotação que será utilizada para mapearmos os atributos para os parâmetros. Observe que, como toda anotação que precisa ser consumida em tempo de execução, ela possui a retenção como `RUNTIME` . Outro ponto importante é que essa anotação é para ser utilizada em atributos, portanto é nesses elementos em que o mapeamento deve ser feito. Apesar de o mapeamento ser feito nos atributos, como esses normalmente são privados por questões de encapsulamento, são os respectivos

métodos setters que devem ser utilizados para a inserção do parâmetro na classe.

Anotação para mapear parâmetros para atributos:

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Parametro {
    String value();
}
```

Para explicar melhor como o mapeamento será feito, a classe a seguir mostra o exemplo de uma classe mapeada. O componente desenvolvido dará suporte a três tipos de parâmetros: booleanos, `String` e array de `String`. No caso de parâmetros booleanos, a presença ou não de sua marcação indica se ele é verdadeiro ou falso. Por exemplo, se `-p` estiver presente entre os parâmetros, precisa-se atribuir o valor verdadeiro no atributo `possui`. No caso de uma `String`, deve-se passar a marcação do parâmetro seguida de seu valor, como, por exemplo, `-s console` para configurar o valor do atributo `saída`. O funcionamento será similar para arrays de `String`, exceto pelo fato de diversos valores poderem ser passados depois da marcação do parâmetro.

Exemplo de uma classe anotada para receber os parâmetros:

```
public class ParamApp {

    @Parametro("-p")
    private boolean possui;

    @Parametro("-n")
    private boolean naoPossui;

    @Parametro("-a")
    private String[] arquivos;

    @Parametro("-s")
```

```
    private String saida;

    //métodos get e set omitidos
}
```

Para iniciar o desenvolvimento desse componente de leitura de parâmetros, vamos começar desenvolvendo o seu construtor, que irá receber a classe mapeada com a anotação `@Parametro` e já vai efetuar a leitura dos seus metadados. Como resultado, será gerado um mapa cuja chave é a marcação do parâmetro, que é passado como valor da anotação, e a respectiva instância de `Field`. Observe que a classe `MapeamentoParametros` possui um tipo genérico, que deve ser o mesmo da classe passada como parâmetro. Ele será utilizado para determinar o retorno do método que mapear os parâmetros para uma instância dessa classe.

Exemplo de uma classe anotada para receber os parâmetros:

```
public class MapeamentoParametros<E> {

    private Map<String, Field> parametros;
    private Class<E> clazz;

    public MapeamentoParametros(Class<E> c) {
        parametros = new HashMap<>();
        clazz = c;
        Class<?> current = c;
        while (current != Object.class) {
            for (Field f : current.getDeclaredFields()) {
                if (f.isAnnotationPresent(Parametro.class)) {
                    Parametro p =
                        f.getAnnotation(Parametro.class);
                    parametros.put(p.value(), f);
                }
            }
            current = current.getSuperclass();
        }
    }
}
```

Segundo os requisitos, é desejado que todos os atributos sejam pesquisados, incluindo os que são privados e os da superclasse. Dessa forma, dois laços são utilizados para percorrer os atributos, sendo que o primeiro itera da classe passada como parâmetro seguindo pelas suas superclasses até `Object`, e o segundo itera sobre os atributos de cada classe recuperados com `getDeclaredFields()`.

Seguindo uma abordagem *bottom-up* de desenvolvimento, na qual primeiro se desenvolve as funções auxiliares e depois a função principal, vamos começar por desenvolver um método que retorna a instância de `Method` que representa o método setter do `Field` passado como parâmetro. Como pode ser visto no código da próxima listagem, inicialmente é gerado o nome do método setter a partir do nome do atributo e, em seguida, percorre-se toda a lista de métodos da classe buscando um método com esse nome. Como não se sabe exatamente qual será o tipo do parâmetro, não é possível buscar o método diretamente. Caso o método não seja encontrado, uma exceção será lançada.

Método que recupera o setter referente a um atributo:

```
private Method getSetter(Field f) {
    String nomeMetodo = "set"
            + Character.toUpperCase(f.getName().charAt(0))
            + f.getName().substring(1);
    for(Method m : clazz.getMethods()){
        if(m.getName().equals(nomeMetodo))
            return m;
    }
    throw new MapeamentoException(nomeMetodo +
                                  "() não encontrado");
}
```

Um outro método auxiliar a ser desenvolvido é o que insere na classe o valor do parâmetro. Ele recebe a instância da classe

mapeada, o nome do parâmetro e a lista de valores que foram passados depois dele. Como pode ser visto na listagem a seguir, o método `inserir()` inicialmente recupera o `Field` relacionado com o nome do parâmetro e, em seguida, o método setter relacionado ao `Field` através do método `getSetter()`. Por fim, para que seja possível invocar o método, o método `recuperarValor()` retorna o valor a ser utilizado como parâmetro de acordo com o tipo esperado pelo método.

Método que insere um parâmetro na classe:

```
private void inserir(E instancia, String nomeParametro,
                    List<String> valores) {
    Field f = parametros.get(nomeParametro);
    if(f == null){
        throw new MapeamentoException(nomeParametro +
            ": parâmetro não previsto");
    }
    Method m = getSetter(f);
    Object valor  = recuperarValor(nomeParametro, valores, m);
    try {
        m.invoke(instancia, valor);
    } catch (IllegalAccessException | IllegalArgumentException
            | InvocationTargetException e) {
        throw new MapeamentoException(
                    "Problemas ao invocar "+m.getName(), e);
    }
}
```

A listagem a seguir mostra a recuperação do valor da lista de `String` de acordo com o tipo do parâmetro esperado pelo método. Diferentemente dos exemplos anteriores, este se preocupa em validar se a entrada está de acordo com o esperado. Nesse caso, o número de parâmetros na lista, que são passados depois do marcador, deve ser compatível com o tipo do parâmetro. Sendo assim, se o tipo for booleano não deve haver nenhum parâmetro, se for `String` deve haver um valor, e se for um array de `String`

pode haver qualquer número. Observe que o tipo considerado é o do parâmetro recebido pelo método setter e não o do atributo. Portanto, o atributo poderia ter um tipo diferente e ser convertido a partir do valor recebido pelo método de atribuição.

Método que retorna o valor de acordo com o parâmetro do método setter:

```
private Object recuperarValor(String nomeParametro,
        List<String> valores, Method m) {
    if(m.getParameterTypes()[0] == boolean.class ||
            m.getParameterTypes()[0] == Boolean.class){
        if(valores.size() > 0){
            throw new MapeamentoException(nomeParametro +
                " não pode possuir valor");
        }else {
            return true;
        }
    }else if(m.getParameterTypes()[0].isArray()){
        return valores.toArray(new String[valores.size()]);
    }else{
        if(valores.size() != 1){
            throw new MapeamentoException(nomeParametro +
                " só pode possuir um valor");
        }else {
            return valores.get(0);
        }
    }
}
```

Para finalizar o exemplo, a listagem a seguir mostra o método `mapear()`, que é o método principal para a realização do mapeamento. Inicialmente, ele cria uma instância da classe para inserir os parâmetros, assumindo que ela possui um construtor vazio. Em seguida, percorre o array de `String` recebido como parâmetro buscando por marcadores, que seriam começados com `-`. Ao encontrar um marcador, os próximos valores serão inseridos na lista de valores até que um novo marcador seja

encontrado ou até que se chegue ao final do array. Ao terminar de guardar os valores relacionados com um marcador, o método `inserir()`, mostrado anteriormente, é invocado para inserir o valor na classe corretamente.

Método que recebe array de String e retorna instância de classe mapeada:

```
public E mapear(String[] args) {
    String nomeParametro = null;
    List<String> valores = new ArrayList<>();
    E instancia = null;
    try {
        instancia = clazz.newInstance();
    } catch (InstantiationException | IllegalAccessException e){
        throw new MapeamentoException(clazz.getName()
                + " não pode ser instanciada", e);
    }
    for (int i = 0; i < args.length; i++) {
        String token = args[i];
        if (token.startsWith("-")) {
            nomeParametro = token;
        } else {
            valores.add(token);
        }
        if (args.length == i + 1 || args[i + 1].startsWith("-"))
        {
            inserir(instancia, nomeParametro, valores);
            nomeParametro = null;
            valores.clear();
        }
    }
    return instancia;
}
```

Observe que o código criado nesse exemplo poderia ser utilizado em programas reais de linha de comando para a recuperação e interpretação de seus parâmetros. Sem a configuração de metadados adicionais aos da classe, não seria possível implementar essa funcionalidade, pois as informações da

própria classe não seriam suficientes para saber que parâmetro é relacionado com cada atributo. Versões mais sofisticadas poderiam prover suporte a parâmetros de outros tipos, ou mesmo prover anotações que configuram como a `String` recebida seria convertida para o tipo desejado. Esse incremento fica como exercício para os leitores.

TIPO DE PARA ERROS DE CONFIGURAÇÃO DE METADADOS

Talvez alguns leitores mais atentos tenham observado que, apesar de os métodos lançarem a `MapeamentoException`, ela não é declarada nos métodos com a cláusula `throws`. Isso acontece porque essa é uma exceção do tipo não checada, ou seja, representa uma situação excepcional da qual normalmente não se pode recuperar, e por isso não obriga o cliente da classe a tratá-la.

No caso de o componente detectar um erro na configuração de metadados, recomenda-se o lançamento de exceções não checadas. Um erro nos metadados é um erro de código e que não deve ser tratado. Caso se passe por cima disso e continue a execução, o componente pode não funcionar da forma desejada e o tratamento do erro pode estar mascarando um bug no software. Por isso, erros nos metadados devem ser tratados como erros de código, que são possíveis de se consertar apenas mexendo no código, seja ele uma anotação na classe ou um arquivo XML.

3.6 CONSIDERAÇÕES FINAIS

Este capítulo falou sobre a configuração de metadados adicionais aos da classe para serem consumidos e utilizados por algoritmos que usam reflexão. Inicialmente, o capítulo falou sobre as opções existentes para definição de metadados, ressaltando as vantagens e desvantagens de cada uma. O uso de convenções de código é fácil, mas possui expressividade limitada e nem sempre é suficiente. A definição programática de metadados permite uma maior flexibilidade, mas acaba não sendo produtiva por utilizar código imperativo para uma configuração declarativa. Fontes externas de metadados podem ser mais facilmente modificadas, mas a definição de metadados é mais verbosa e fica distante do código. Por fim, as anotações minimizam a quantidade de configurações, mas complicam o reúso da classe em outros contextos e não podem ser utilizadas para classes das quais não se tem acesso ao código.

Em seguida, o capítulo focou em mostrar como as anotações são definidas e utilizadas em Java. Inicialmente foi mostrado como uma nova anotação pode ser criada e quais são as opções que podem ser configuradas. Depois foi mostrado como as anotações podem ser recuperadas pela API de reflexão para serem utilizadas por uma classe ou componente. Por fim, um exemplo mais completo mostrou como o mapeamento entre representações de uma entidade pode ser feito através da configuração de metadados.

Capítulo 4

PROXY DINÂMICO

"Nunca faça por um intermediário o que você pode fazer você mesmo." - Provérbio italiano

Nos capítulos anteriores foram apresentadas as principais funcionalidades da API de reflexão. Inicialmente foi mostrado como obter a referência para as classes e como utilizar suas informações para manipular suas instâncias. Em seguida, falou-se sobre a definição de metadados e como eles podem ser utilizados para adicionar novas informações sobre os elementos de um programa. A partir dessas funcionalidades, vimos alguns exemplos mais completos de como elas podem ser usadas para gerar componentes reutilizáveis que poderiam ser utilizados em softwares reais.

Uma das funcionalidades mais impressionantes da API Reflection da linguagem Java é a criação de proxies dinâmicos. De uma maneira bem informal, a partir deles é possível criar uma instância que implementa uma interface em tempo de execução. Em outras palavras, ele pode implementar a interface que você quiser, tendo uma forma de tratar as chamadas de método feitas para os métodos dessa interface. A partir disso, é possível ter uma única classe que pode encapsular outras classes com diferentes métodos e interfaces.

Lembro-me que a minha reação ao ver o uso de um proxy dinâmico pela primeira vez foi algo como: "*Mas pode fazer isso? Vale fazer isso?*". Era como descobrir uma nova regra no meio do jogo, que certamente poderia me ajudar a vencê-lo de forma bem mais fácil e mais inteligente. A partir de alguns exemplos, este capítulo irá mostrar não somente como utilizar essa técnica, mas também como ela pode ser usada para introduzir novos comportamentos nas classes de forma transparente. É essa a *magia* que alguns frameworks utilizam para que certas coisas simplesmente aconteçam ao invocarmos um método das classes da própria aplicação.

Além do uso da API Reflection para gerar proxies dinâmicos a partir de interfaces, este capítulo também vai mostrar o uso da ferramenta CGLib para gerar proxies dinâmicos a partir de classes, o que é muito útil para classes como Java beans. Ele também irá falar sobre o uso de anotações para auxiliar no desenvolvimento da funcionalidade do proxy dinâmico e como essa funcionalidade pode ser utilizada para gerar em tempo de execução um implementação de uma interface, sem haver nenhuma classe compilada realmente que a implemente. Como os capítulos anteriores, ao final será apresentado um exemplo mais completo que consolida o conhecimento em proxies dinâmicos, relacionando-os com o que vimos nos capítulos anteriores.

4.1 O QUE É UM PROXY?

Antes de entrarmos no detalhe de como funciona um proxy dinâmico, primeiro vamos revisar o conceito de proxy. O proxy é um padrão de projeto que tem o objetivo de proteger o acesso de objetos, de forma transparente às classes que o utilizam. O padrão

Decorator, apesar de possuir um objetivo diferente, que é de adicionar novas funcionalidades à classe, possui uma estrutura muito similar. Ambos os padrões encapsulam a classe original, intermediando o acesso a ela. Dessa forma, enquanto o controle estiver com a classe intermediária, ela pode executar verificações de segurança ou outras funcionalidades, antes de repassar o controle da execução para a classe original.

A palavra *proxy* em inglês significa *intermediário*, o que é uma metáfora adequada para esse padrão, visto que o proxy intermedeia o acesso entre o objeto cliente e o objeto original. Muitas pessoas estão acostumadas com o uso do nome *proxy* para os proxies de rede, que de certa forma possuem um comportamento análogo. Nesse caso, quando existe um proxy, em vez de enviar os pacotes de rede para o endereço de destino, eles são enviados para o proxy, que então redireciona para o endereço original servindo como um intermediário. Tanto para o cliente, quanto para o destino original, é transparente a existência do proxy. Porém, apesar dessa transparência, o proxy pode prestar diversos serviços estando no meio do caminho, como a filtragem de conteúdos, controle do uso de banda, autenticação, entre outros.

A figura seguinte ilustra o funcionamento de um proxy. O primeiro ponto importante é que o objeto que faz o papel do proxy deve possuir a mesma interface da classe original, de forma que ele possa assumir o lugar dela na classe cliente de modo transparente.

Outro ponto que deve ser ressaltado é que o proxy deve possuir uma referência para um objeto da classe encapsulada, de forma a poder repassar as chamadas para ele quando necessário. Sendo

assim, quando a classe cliente invocar algum método no proxy, este irá executar alguma funcionalidade e, se necessário, irá repassar a chamada para o objeto da classe encapsulada. Vale ressaltar que qualquer objeto que implementar a interface pode ser encapsulado dentro do proxy, o que permite que diversos proxies sejam encadeados.

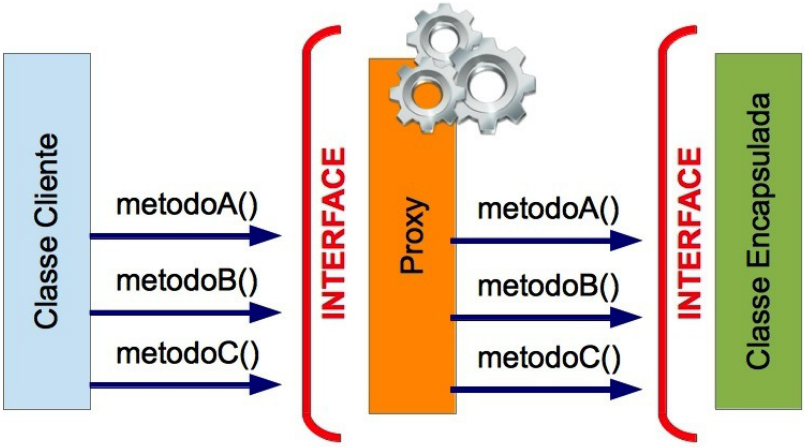

Figura 4.1: Funcionamento de um proxy

Exemplos desse tipo de proxy foram mostrados na seção *Ainda falta alguma coisa na orientação a objetos?*, mais especificamente com as classes `LoggingAsyncProxy` e `ProdutoDAOAsync` , que implementavam proxies que executavam o método da classe encapsulada em uma thread diferente. Observe que elas implementam a mesma interface que o objeto que elas estão encapsulando e recebem em seu construtor. Em cada método, o proxy executa a sua funcionalidade, que no caso é criar uma nova thread, e depois invoca o método no objeto original. Para os clientes que esperam um objeto que implemente a interface, é

transparente o fato de o proxy estar presente ou não. Dessa maneira, é como se essa funcionalidade pudesse ser plugada no objeto original.

O proxy implementado assim funciona muito bem quando precisa ser aplicado a classes que compartilham uma mesma interface, pois, nesse caso, uma única implementação do proxy poderia ser utilizada para qualquer classe. O problema dessa abordagem está quando as classes que o proxy precisa encapsular não possuem uma interface em comum. Nesse caso, para cada interface diferente precisaria ser criada uma classe de proxy, o que poderia gerar repetição de código como no caso das classes `LoggingAsyncProxy` e `ProdutoDAOAsync`, que possuem um código muito similar.

4.2 PROXY DINÂMICO COM A API REFLECTION

A API Reflection possui um recurso que permite a criação de um objeto que implementa uma interface em tempo de execução. Esse objeto é criado por uma classe da API, porém todas as chamadas de método recebidas por ele são redirecionadas para uma implementação da interface `InvocationHandler`. Essa implementação é quem contém a funcionalidade do proxy e redireciona as chamadas para a classe original. A seguir é mostrada passo a passo a criação de um proxy dinâmico e explicado com mais detalhe como ele funciona.

Receita para criar um proxy dinâmico

Vamos mostrar nesta seção passo a passo como criar um proxy

dinâmico com a API de reflexão. Como exemplo, será mostrada a implementação de um proxy que executa os métodos da classe encapsulada de forma assíncrona, ou seja, em uma nova thread. A ideia é que essa classe implemente a mesma funcionalidade das classes `LoggingAsyncProxy` e `ProdutoDAOAsync`, porém de modo a poder ser reutilizada para qualquer interface.

O primeiro passo para implementar o proxy dinâmico é a implementação da interface `InvocationHandler`, como mostrado na listagem a seguir. Ela tem apenas um método chamado `invoke()`, para o qual serão direcionadas todas as chamadas de método. Esse método recebe os três parâmetros a seguir: uma instância do objeto gerado dinamicamente (que não é o objeto encapsulado); o método que foi invocado; e um array de objetos com os parâmetros que foram passados para ele.

Implementação da interface InvocationHandler:

```
public class AsyncProxy implements InvocationHandler {

    public Object invoke(Object proxy, Method method,
     Object[] args) throws Throwable {
        return null;
    }
}
```

O segundo passo é encapsular o objeto original nessa classe. Observe na listagem a seguir, que complementa o código anterior, que foi adicionado um atributo para armazenar o objeto encapsulado, o qual é passado no construtor. Esse construtor foi definido como privado, para que ele só possa ser instanciado pelo método que cria o proxy dinâmico que será definido no próximo passo. A implementação do método `invoke()` agora delega a execução para o objeto encapsulado, chamando nele o mesmo

método com os mesmos argumentos.

Implementando o encapsulamento do objeto original:

```
public class AsyncProxy implements InvocationHandler {

    private Object obj;

    private AsyncProxy( Object obj) {
        this.obj = obj;
    }
    public Object invoke(Object proxy, Method method,
     Object[] args) throws Throwable {
        return method.invoke(obj,args);
    }
}
```

O terceiro passo é criar um método estático que efetivamente cria o proxy dinâmico. Esse método recebe como parâmetro um objeto e retorna-o encapsulado com o proxy. A criação do proxy pode ser feita a partir do método estático `newProxyInstance()` da classe `Proxy` .

Esse método recebe como primeiro parâmetro uma instância de `ClassLoader` , para a qual normalmente é passada o mesmo da classe do objeto que será encapsulado, recuperado a partir do método `getClassLoader()` de `Class` .

O segundo parâmetro são as interfaces as quais o proxy dinâmico deve ser aplicado. No exemplo, todas as interfaces do objeto são recuperadas a partir do método `getInterfaces()` e passadas como parâmetro. Porém, pode-se passar apenas as interfaces cujos métodos devem ser interceptados pelo proxy. O último parâmetro é uma instância de `InvocationHandler` , que no caso é uma instância da própria classe que estamos criando, encapsulando o objeto passado como parâmetro.

Método que retorna objeto encapsulado no proxy dinâmico:

```
public class AsyncProxy implements InvocationHandler {

    public static Object criarProxy(Object obj){
        return Proxy.newProxyInstance
              (obj.getClass().getClassLoader(),
                obj.getClass().getInterfaces(),
                new AsyncProxy(obj));
    }

    //parte já mostrada omitida
}
```

Como último passo, falta apenas adicionar ao método `invoke()` a funcionalidade de execução assíncrona aos métodos do objeto encapsulado. Nesse exemplo, o método só será executado em uma thread diferente caso retorne `void`, porque nesse caso não é necessário aguardar sua execução para poder retornar. O código a seguir mostra a implementação dessa funcionalidade complementando o código já iniciado nas listagens anteriores.

Adicionando a funcionalidade de invocação assíncrona:

```
public class AsyncProxy implements InvocationHandler {

    //método de criação do proxy omitido

    private Object obj;

    private AsyncProxy( Object obj) {
        this.obj = obj;
    }
    public Object invoke(Object proxy, Method method,
     Object[] args) throws Throwable {
        if(method.getReturnType() == void.class){
            new Thread(
                public void run(){
                    method.invoke(obj,args);
```

```
                }
            ).start();
            return null;
        } else {
            return method.invoke(obj,args);
        }
    }
}
```

É isso! São esses os passos para se criar um proxy dinâmico com a API Reflection! Os três primeiros passos são sempre muito similares e o que muda é o último passo, no qual a funcionalidade adicionada pelo proxy é implementada. Essa funcionalidade, dependendo do caso, pode ser implementada antes, depois ou ao redor da invocação do método do objeto original. Observe que, como temos controle dos parâmetros que são passados para o objeto original e do retorno que é dado à invocação do método, esses podem ser modificados e manipulados pelo proxy.

Repassando exceções do objeto original

Um erro comum ao se implementar um proxy dinâmico está em não se propagar corretamente as exceções lançadas pelo objeto original. Pode parecer que, somente chamando o método no objeto encapsulado e passando os mesmos argumentos, o resultado seria o mesmo, porém isso não é verdade devido às exceções. É preciso lembrar que a exceção lançada pelo método original será encapsulada em uma `InvocationTargetException`.

Sendo assim, a próxima listagem mostra como seria um proxy que não faz nada, somente repassando as chamadas para o objeto encapsulado e repassando as suas respostas. Para isso, além de devolver o retorno da chamada de `invoke`, é preciso capturar a exceção `InvocationTargetException` e lançar a exceção contida

`getTargetException()` . Dessa forma, caso o objeto original lance uma exceção, o proxy irá a repassar para a classe cliente.

Lançando as mesmas exceções que o objeto original:

```
public class NaoFAzNadaProxy implements InvocationHandler {

    private Object obj;

    private NaoFAzNadaProxy(Object obj) {
        this.obj = obj;
    }
    public Object invoke(Object proxy, Method method,
     Object[] args) throws Throwable {
        try{
            return method.invoke(obj,args);
        } catch(InvocationTargetException e) {
            throw e.getTargetException();
        }
    }
}
```

Funcionamento de um proxy dinâmico

Para entender melhor como funciona o proxy dinâmico, a figura adiante apresenta uma representação de como as chamadas são realizadas. Uma confusão comum é achar que o proxy dinâmico é a implementação da interface `InvocationHandler` . Na verdade, ele é uma classe criada pela máquina virtual a partir dos parâmetros passados para o método `newProxyInstance()` da classe `Proxy` . A instância do proxy será utilizada no lugar do objeto original de forma transparente para a classe cliente, visto que implementa a interface esperada por ele. Sendo assim, o proxy irá receber as chamadas de métodos normalmente como qualquer classe.

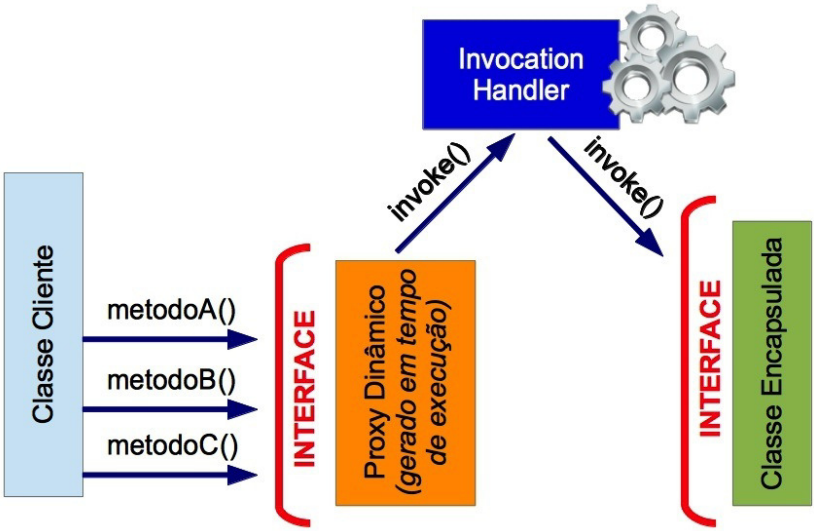

Figura 4.2: Funcionamento do proxy dinâmico

Ao receber essa chamada de método, o método `invoke()` da implementação de `InvocationHandler` é chamado. O primeiro parâmetro que é passado para esse método, como já foi dito anteriormente, é o objeto do proxy que foi gerado dinamicamente. Além disso, esse método também recebe o método e seus parâmetros, dando condições de replicar a chamada do método no objeto encapsulado. Sendo assim, a implementação de `InvocationHandler` pode executar a sua funcionalidade e repassar a chamada para o objeto encapsulado quando for adequado.

4.3 GERANDO A IMPLEMENTAÇÃO DE UMA INTERFACE

Um proxy dinâmico normalmente encapsula um objeto para

acrescentar uma funcionalidade a ele, independente da sua interface. Porém, como pôde ser visto no exemplo mostrado na seção anterior, todo esse encapsulamento é feito na implementação de `InvocationHandler`. Sendo assim, seria possível gerar um proxy dinâmico para uma interface, tratando as chamadas para todos os seus métodos, sem realmente estar encapsulando uma implementação concreta dessa interface.

Talvez você esteja se perguntando nesse momento: como vou saber qual deve ser o comportamento para os métodos de uma interface? A resposta é simples: através de seus metadados! O nome da classe, o nome do método, seus parâmetros e suas anotações podem fornecer informações suficientes para que o proxy saiba a lógica que ele precisa executar quando um determinado método for invocado. Dessa forma, a definição da assinatura do método juntamente com metadados adicionais será responsável por definir seu comportamento. É como se eles definissem uma DSL interna para a definição do comportamento daquelas classes.

O QUE É UMA DSL INTERNA?

O acrônimo DSL significa *Domain Specific Language*, que em português seria traduzido como *linguagem específica de domínio*. Diferentemente de uma linguagem de propósito geral, como Java, que pode ser utilizada para desenvolver qualquer tipo de programa, uma DSL possui um propósito mais específico, sendo utilizada somente dentro de um domínio. Um exemplo de uma DSL é a linguagem SQL, que possui o propósito de executar operações em bases de dados relacionais.

Uma DSL interna é uma linguagem específica de um domínio que é criada dentro de uma linguagem de propósito geral já existente. No caso da criação de um proxy que gera uma implementação baseada na assinatura de um método, os elementos dessa assinatura passam a possuir uma semântica específica do domínio do proxy, que vai gerar, nesse caso, um comportamento em tempo de execução. A limitação dessa linguagem interna acaba sendo a limitação da sintaxe da própria linguagem que está sendo utilizada como base. Esse "jeito" de criar a assinatura do método, nada mais é que uma nova linguagem mais específica que está sendo criada para o domínio do proxy.

Para exemplificarmos a utilização dessa abordagem, vamos ver um exemplo de um proxy que gera a implementação de uma interface que a partir dos nomes dos métodos retorna uma

propriedade de sistema. Por exemplo, se a interface possuir um método `getUserHome()`, ele irá retornar a propriedade de sistema `user.home`. Dessa forma, o proxy precisa interpretar o nome do método para extrair o nome da propriedade que precisa retornar.

O primeiro método auxiliar desenvolvido separa a `String` com o nome do método em um array de `String`, com os pedaços que são separados por letras maiúsculas. Por exemplo, o código transformaria `"getCamelCase"` em `{"get", "camel", "case"}`. A implementação está apresentada na listagem a seguir. Os caracteres da `String` são percorridos e adicionados na variável `corrente`, que é adicionada à lista e zerada ao se encontrar um caractere maiúsculo.

Método que separa a String por suas letras maiúsculas:

```
public String[] separaPorMaiusculas(String nome){
    List<String> lista = new ArrayList<>();
    String corrente = "";
    for(int i=0; i<nome.length(); i++){
        if(Character.isUpperCase(nome.charAt(i))){
            lista.add(corrente);
            corrente = "";
            corrente += Character.toLowerCase(nome.charAt(i));
        }else{
            corrente += nome.charAt(i);
        }
    }
    lista.add(corrente);
    return lista.toArray(new String[lista.size()]);
}
```

O segundo método auxiliar, apresentado a seguir, é o que recebe a `String` dividida e retorna o nome esperado para a propriedade. Ele pega cada `String` e adiciona em uma única, dividindo-as por um `"."`. Observe que o bloco `for` começa sua

iteração pelo segundo elemento para pular a primeira `String`, que seria o `"get"`. Por exemplo, o código transformaria `{"get", "camel", "case"}` em `"camel.case"`.

Transformação do array de String em nome da propriedade:

```
public String nomePropriedade(String[] strs){
    String nomeProp = "";
    for(int i=1; i<strs.length; i++){
        if(i != 1){
            nomeProp += ".";
        }
        nomeProp += strs[i];
    }
    return nomeProp;
}
```

Por fim, a classe `SystemPropertiesRetriever` na próxima listagem realiza a implementação do proxy propriamente dito. O método `invoke()` chama o método de separação por maiúsculas, extrai o nome da propriedade e a retorna, utilizando a chamada `System.getProperty()`. O método estático `criar()` que faz a criação do proxy, recebe como parâmetro o `Class` que representa a interface e retorna um objeto que a implementa dinamicamente. Observe que, na chamada de `newProxyInstance()`, um array com a própria interface é passado como o segundo parâmetro e uma instância da própria classe é passada como terceiro parâmetro.

Implementação do proxy que retorna a propriedade do sistema de acordo com o nome do método:

```
public class SystemPropertiesRetriever implements
 InvocationHandler {

    public static <E> E criar(Class<E> interf){
        return (E) Proxy.newProxyInstance
```

```
            (interf.getClassLoader(),
             new Class[]{interf},
             new SystemPropertiesRetriever());
    }

    @Override
    public Object invoke(Object proxy, Method method,
     Object[] args) throws Throwable {
        String[] split = separaPorMaiusculas(method.getName());
        String nomeProp = nomePropriedade(split);
        return System.getProperty(nomeProp);
    }
}
```

Para exemplificar a utilização do proxy desenvolvido, a listagem a seguir mostra uma interface que possui métodos para recuperação de diversas propriedades do sistema. A seguir, é apresentado um método `main()` que cria o proxy baseado nessa interface e imprime as propriedades do sistema no console. Vale observar que essa implementação não faz muitas verificações adicionais em relação ao método recebido como parâmetro. O método poderia, por exemplo, possuir parâmetros ou um tipo de retorno não adequado. Deixo a implementação desse tratamento de erro como exercício aos leitores.

Exemplo de uso do proxy desenvolvido:

```
public interface SystemProperties {
    public String getUserCountry();
    public String getUserLanguage();
    public String getUserHome();
    public String getJavaVmSpecificationVersion();
    public String getJavaHome();
    public String getFileSeparator();
}

public class Principal {

    public static void main(String[] args) {
        SystemProperties sp =
```

```
            SystemPropertiesRetriever.criar(
                                        SystemProperties.class);
        System.out.println(sp.getUserHome());
        System.out.println(sp.getUserCountry());
        System.out.println(sp.getUserLanguage());
        System.out.println(sp.getFileSeparator());
        System.out.println(sp.getJavaHome());
        System.out.println(sp.getJavaVmSpecificationVersion());
    }
}
```

FRAMEWORK ESFINGE QUERYBUILDER

Para perceber o potencial dessa prática é interessante ver um exemplo mais avançado. O framework Esfinge QueryBuilder utiliza a assinatura dos métodos para gerar consultas para diferentes tipos de banco de dados. Para isso, são combinadas informações da assinatura do método, anotações nos parâmetros e dos próprios tipos dos parâmetros. É possível, por exemplo, definir através de uma anotação em uma interface novos termos de domínio para serem utilizados nos nomes dos métodos. Devido à diversidade de possibilidades que se chegou com essa DSL interna, estão em desenvolvimento plugins para Eclipse que fazem a verificação do nome dos métodos em tempo de compilação e que estendem o mecanismo de refatoração de renomeação para que os nomes dos métodos sejam também modificados quando nomes nas classes persistentes forem alterados.

4.4 PROXY DE CLASSES COM CGLIB

"Quem não tem cão, caça com gato." - Ditado popular

Um dos problemas do proxy dinâmico da API Reflection da linguagem Java é que ele só pode ser aplicado para métodos de interfaces. Até mesmo classes que possuem uma interface não podem ser interceptadas pelo proxy dinâmico em métodos fora dessa interface. Isso é um problema especialmente para classes nos estilo Java Beans, pois não faz muito sentido possuir uma interface para os métodos de acesso aos seus atributos. Imagine precisar criar uma interface para cada classe de domínio com todos seus métodos getters e setters somente para poder utilizar um proxy dinâmico em seus métodos.

Felizmente, existe uma alternativa! A biblioteca CGLib, um acrônimo para *Code Generation Library*, possibilita a criação de proxies dinâmicos para classes a partir de geração de bytecode. Em termos de programação, o desenvolvimento de um proxy dinâmico com CGLib não é muito diferente de um proxy dinâmico com a API Reflection. Não vamos entrar muito em detalhes aqui sobre como funciona o processo de geração de bytecode e carregamento dinâmico de classes em tempo de execução, porém esse processo será mais bem detalhado no capítulo *Manipulação de bytecode*.

Proxy para armazenar histórico de valores

Para utilizar o CGLib em um projeto, basta adicionar o arquivo `cglib-nodep-3.1.jar` em seu classpath. Para exemplificar o uso do proxy, será desenvolvida uma classe que guarda o histórico de valores que foram atribuídos a uma propriedade de um Java Bean, permitindo sua recuperação posterior. Antes de criar o proxy, vamos definir uma interface para permitir a recuperação do histórico de valores de uma propriedade. A definição dessa

interface está apresentada na listagem a seguir. Ela será utilizada para ser incorporada na classe que será gerada dinamicamente. Dessa forma, quando a chamada do método dessa interface for redirecionada para o proxy, ele retorna o valor do histórico ao invés de redirecionar a chamada.

Interface para recuperação do histórico:

```
public interface RecuperadorHistorico {
    public List<Object> getHistorico(String prop);
}
```

Para criar o proxy com o CGLib, a interface que precisa ser implementada é `MethodInterceptor`, que possui apenas o método `intercept()`, o qual será chamado a cada chamada do método. A única diferença do `intercept()` para o método `invoke()` de `InvocationHandler` é seu quarto parâmetro do tipo `MethodProxy`. Essa classe é uma outra forma de acessar métodos provido pelo CGLib. Por enquanto, vamos ignorar esse parâmetro e utilizar a API de reflexão, mas voltaremos a isso no capítulo *Ferramentas: indo além na reflexão*, que irá abordar outras funcionalidades do CGLib.

A próxima listagem apresenta o proxy que armazena o histórico de valores das propriedades de um Java Bean. A classe armazena como atributos o objeto encapsulado, de forma similar ao que foi mostrado para o proxy da API Reflection, e um mapa que para cada propriedade guarda uma lista de valores. Nessa versão do código, o método `intercept()` ainda está incompleta, apresentando em comentários com `PARA FAZER` questões que serão implementadas na próxima listagem. Em sua implementação, o primeiro condicional identifica se o método é da interface `RecuperadorHistorico`, e em caso positivo, assume-se

que o método executado é o `getHistorico()`, e o histórico da propriedade passada como parâmetro deve ser retornado. No condicional seguinte, caso o método interceptado seja um setter, identificado pelo sufixo `"set"` e por receber um parâmetro, o valor do parâmetro é armazenado no histórico. Após isso, o método do objeto encapsulado é executado normalmente.

Proxy para armazenamento e recuperação do histórico:

```
public class Historico implements MethodInterceptor{

    private Object encapsulated;
    private Map<String, List<Object>> historico =
                                                new HashMap<>();

    public Historico(Object encapsulated) {
        this.encapsulated = encapsulated;
    }
    public Object intercept(Object obj, Method m, Object[] args,
            MethodProxy proxy) throws Throwable {
        if(m.getDeclaringClass().equals(
            RecuperadorHistorico.class)){
            //PARA FAZER: retorna o histórico
        }
        if(m.getName().startsWith("set") &&
                    m.getParameterTypes().length == 1){
            String prop = deSetterParaPropriedade(m.getName());
            //PARA FAZER: armazena valor no histórico para
            //propriedade
        }
        try {
            return m.invoke(encapsulated, args);
        } catch (InvocationTargetException e) {
            throw e.getTargetException();
        }
    }
    private static String deSetterParaPropriedade(
     String nomeSetter){
        StringBuffer retorno = new StringBuffer();
        retorno.append(nomeSetter.substring(3,4).toLowerCase());
        retorno.append(nomeSetter.substring(4));
        return retorno.toString();
```

```
    }
}
```

Uma questão a ser ressaltada é o tratamento especial caso o método seja da interface `RecuperadorHistorico`, que será adicionada no proxy independente da classe o implementar ou não. Esse caso especial exemplifica situações em que desejamos adicionar métodos no proxy que será gerado. Como esses métodos não podem ser repassados ao objeto encapsulado, eles devem ser identificados dentro do `intercept()` e uma lógica adequada deve ser executada em resposta. O resultado seria um valor do proxy que deve ser retornado. Apesar de esse exemplo utilizar a CGLib, uma estratégia parecida pode ser seguida quando for necessário adicionar métodos no proxy gerado com a API Reflection.

A listagem a seguir, mostra o código do método `intercept()` completo. Dentro do primeiro condicional, que verifica se o método é da interface `RecuperadorHistorico`, é retornada a lista de valores armazenada no mapa. Observe que, como nesse caso o método interceptado é conhecido, `getHistorico()`, é possível saber que o primeiro parâmetro se refere à propriedade, e utilizá-lo como chave de recuperação no mapa. No segundo condicional, que deve fazer o armazenamento do histórico, é verificado se já existe uma lista com os valores da propriedade. Caso não exista, uma lista é criada e inserida no mapa, e caso já exista, é simplesmente recuperada. Em posse da lista, o valor recebido pelo método setter como parâmetro é inserido no histórico.

Completando o método intercept() do proxy Historico:

```
public Object intercept(Object obj, Method m, Object[] args,
        MethodProxy proxy) throws Throwable {
```

```
    if(m.getDeclaringClass().equals(RecuperadorHistorico.class))
    {
        return historico.get(args[0]);
    }
    if(m.getName().startsWith("set") &&
            m.getParameterTypes().length == 1){
        String prop = deSetterParaPropriedade(m.getName());
        List<Object> list = null;
        if(!historico.containsKey(prop)){
            list = new ArrayList<Object>();
            historico.put(prop, list);
        }else{
            list = historico.get(prop);
        }
        list.add(args[0]);
    }
    try {
        return m.invoke(encapsulated, args);
    } catch (InvocationTargetException e) {
        throw e.getTargetException();
    }
}
```

A listagem a seguir mostra o método de criação do proxy, chamado de `guardar()`, onde o CGLib gera dinamicamente a classe que irá direcionar as chamadas para o `MethodInvoker`. A classe `Enhancer` gera, de certa forma, uma versão incrementada de uma determinada classe. Observe que é configurado para que ele utilize a classe do objeto encapsulado como superclasse, que tenha a interface `RecuperadorHistorico` e que utilize uma instância de `Historico` para redirecionar as chamadas. Em seguida, é retornado um objeto dessa nova classe criada em tempo de execução. Apesar de ser diferente da geração do proxy dinâmico com a API Reflection, essa é uma receita de bolo que muda muito pouco entre diferentes proxies.

Método de criação do proxy com CGLib:

```
public static <E> E guardar(E obj) {
```

```
    try {
        Historico proxy =  new Historico(obj);
        Enhancer e = new Enhancer();
        e.setSuperclass(obj.getClass());
        e.setInterfaces(
            new Class[]{RecuperadorHistorico.class});
        e.setCallback(proxy);
        return (E) e.create();
    } catch (Throwable e) {
        throw new Error(e.getMessage());
    }
}
```

Finalizando o exemplo, a listagem a seguir mostra uma utilização do proxy criado. A classe `Produto`, um Java Bean comum e sem interfaces, será utilizada como base para a geração do proxy. Observe que, para a obtenção da instância utilizada no exemplo, a chamada `Historico.guardar()` foi realizada para gerar o proxy dinâmico e encapsular o objeto passado como parâmetro. Nesse código, o valor do preço do produto é alterado diversas vezes para testarmos se o histórico está sendo realmente armazenado.

Exemplo de utilização do proxy:

```
public class Principal {

    public static void main(String[] args) {
        Produto p = Historico.guardar(new Produto());
        p.setNome("Design Patterns com Java");
        p.setMarca("Casa do Código");
        p.setPreco(59.90);

        //blackfriday
        p.setPreco(49.90);
        //normal
        p.setPreco(59.90);
        //natal
        p.setPreco(54.90);
```

```
        List<Object> lista =
            ((RecuperadorHistorico)p).getHistorico("preco");
        for(Object valor : lista){
            System.out.println(valor);
        }
    }
}
```

Apesar de `Produto` não implementar a interface `RecuperadorHistorico`, o proxy gerado implementa. Então, para poder acessar os métodos dessa interface, basta fazer um *cast* e invocar o método desejado, como mostrado na listagem. Executando o método `main()`, é possível observar que todo o histórico de valores da propriedade `preco` será impresso no console.

Esse exemplo pode parecer simples, mas com pequenas mudanças pode-se adicionar uma referência de data ao histórico de valores, registrando o momento da modificação. A partir dessas informações, é possível gerar uma cópia do objeto com os valores das propriedades que ele possuía em um determinado momento. Dessa forma, consegue-se recuperar a versão que um determinado objeto possui a qualquer momento no tempo.

Funcionamento de um proxy gerado com CGLib

O funcionamento de um proxy dinâmico com CGLib é análogo ao proxy dinâmico criado com a API Reflection, como pode ser visto na figura seguinte. A principal diferença é que, nesse caso, não existe uma interface para ser uma abstração comum entre o proxy e o objeto encapsulado. Dessa forma, a solução é que a classe gerada pelo `Enhancer` estenda a encapsulada, sendo, dessa forma, uma subclasse dela. Vale ressaltar que a classe `Enhancer` permite que diversas outras questões sejam

configuradas para a classe gerada, porém não está no escopo deste livro abordar essas questões.

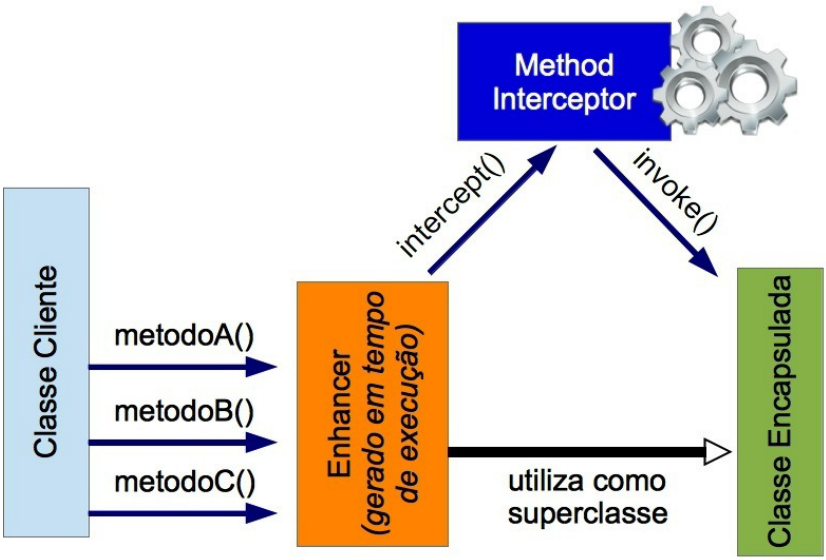

Figura 4.3: Proxy dinâmico com CGI

Cada método invocado será interceptado pela instância de `MethodInterceptor` passado no método `setCallback()` da classe `Enhancer`. De forma similar às implementações da interface `InvocationHandler`, essa classe é quem efetivamente implementa a funcionalidade do proxy dinâmico e redireciona as chamadas ao objeto encapsulado.

Maturidade do Projeto CGLib

Antes de utilizar alguma nova biblioteca em um projeto, um questionamento comum se faz a respeito da maturidade daquela biblioteca. Muito provavelmente, se você desenvolve para aplicações corporativas, já tem o CGLib no classpath da sua aplicação, pois ele é utilizado por frameworks de grande aceitação no mercado, como o Hibernate e o Spring. Sendo assim, fique tranquilo porque o CGLib é uma biblioteca muito madura e já foi utilizada em frameworks usados no desenvolvimento de aplicações de grande porte.

4.5 CONSUMINDO ANOTAÇÕES EM PROXIES

Nas soluções de proxy dinâmico que foram apresentadas, todas as chamadas são direcionadas para o mesmo método de tratamento. Ele recebe o método invocado na classe original e seus parâmetros. Devido a não saber mais sobre ele do que o que está nos seus metadados, o proxy acaba tratando todos eles da mesma forma. Nos exemplos, algumas convenções de código foram utilizadas para distinguir os métodos, porém, como foi dito na seção *Definição de metadados*, a expressividade dessa abordagem é limitada.

Sendo assim, a configuração de metadados é uma excelente alternativa para distinguir os métodos de uma classe, permitindo que o proxy tenha um comportamento distinto para cada um deles. Vamos imaginar o exemplo de um proxy que armazene o

resultado de chamadas de métodos e retorne o mesmo resultado caso uma chamada semelhante seja realizada. Porém, considere também que existam métodos que alteram a classe encapsulada internamente e invalidam os resultados que já foram armazenados. Ao desenvolvê-lo, o primeiro passo seria definir anotações que configuram se deve ser feito cache dos retornos de um método, ou se esse cache deve ser invalidado a partir da chamada daquele método. Essas anotações estão apresentadas na listagem a seguir.

Anotações para marcar os métodos interceptados pelo proxy:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Cache {
}

@Retention(RetentionPolicy.RUNTIME)
public @interface InvalidaCache {
}
```

Para armazenar os dados de uma invocação de métodos para o armazenamento do cache será utilizado um mapa. Esse mapa deve utilizar uma chave que identifica unicamente as chamadas de método, incluindo o que foi invocado e os seus parâmetros. Dessa forma, o método `gerarChave()`, apresentado na próxima listagem, cria uma `String` a partir do nome do método e do valor de seus parâmetros para serem utilizados como chave. Deixo como nota que essa geração de chaves vai depender muito dos objetos terem uma implementação adequada do método `toString()`. Para arrays, por exemplo, essa geração de chaves não funcionaria muito bem.

Método para geração da chave de armazenamento da invocação do método:

```
private String gerarChave(Method method, Object[] args){
    StringBuilder sb = new StringBuilder();
    sb.append(method.getName());
    for(int i=0; i<args.length; i++)
        sb.append(args[i]);
    return sb.toString();
}
```

A listagem a seguir mostra a implementação do proxy que faz o cache se baseando nas anotações dos métodos. Observe que, no método `invoke()`, que nessa versão ainda está incompleto, existe um comando condicional que realiza diferentes ações dependendo da anotação presente no método. O primeiro condicional verifica a presença da anotação `@InvalidaCache`, e o cache das chamadas de método armazenadas até o momento deve ser limpado quando ele é chamado. O segundo condicional verifica a presença da anotação `@Cache`; caso exista, deve ser retornado o valor do cache, ou então executado o método no objeto encapsulado e armazenado seu resultado. Observe que se não houver nenhuma anotação presente, o método é executado normalmente.

Implementação do proxy que faz o cache das chamadas de método:

```
public class CacheProxy implements InvocationHandler {

    private Object obj;
    private Map<String,Object> cache = new HashMap<>();

    public CacheProxy(Object obj) {
        this.obj = obj;
    }
    public Object invoke(Object proxy,
     Method method, Object[] args) throws Throwable {
        try {
            if(method.isAnnotationPresent(InvalidaCache.class)){
                //PARA FAZER: limpar cache
            }else if(method.isAnnotationPresent(Cache.class)){
```

```
                //PARA FAZER: armazenar ou retornar objeto do
                //ache
            }
            return method.invoke(obj, args);
        } catch (InvocationTargetException e) {
            throw e.getTargetException();
        }
    }
    public static Object criar(Object obj){
        return Proxy.newProxyInstance (
            obj.getClass().getClassLoader(),
            obj.getClass().getInterfaces(),
            new CacheProxy(obj));
    }
}
```

A listagem a seguir, apresenta o método `invoke()` da classe `CacheProxy` finalizado. Observe que o método `clear()` do mapa que armazena o cache é chamado para limpar o cache na presença da anotação `@InvalidaCache`. Já na presença da anotação `@Cache`, o primeiro passo é gerar a chave relativa à invocação utilizando o método `gerarChave()` mostrado anteriormente. Em seguida, é verificado se o mapa que armazena o cache já possui a chave. Caso ela exista, o valor armazenado no mapa é retornado e o método no objeto encapsulado nem é invocado. Por outro lado, se a chave não existir, o objeto encapsulado é invocado, o seu retorno e armazenado no mapa e, por fim, esse valor é retornado pelo proxy.

Implementação do método invoke() do CacheProxy:

```
public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
    try {
        if(method.isAnnotationPresent(InvalidaCache.class)){
            cache.clear();
        }else if(method.isAnnotationPresent(Cache.class)){
            String chave = gerarChave(method, args);
            if(cache.containsKey(chave)){
```

```
                return cache.get(chave);
            }else{
                Object retorno = method.invoke(obj, args);
                cache.put(chave, retorno);
                return retorno;
            }
        }
        return method.invoke(obj, args);
    } catch (InvocationTargetException e) {
        throw e.getTargetException();
    }
}
```

Para finalizar o exemplo, vamos mostrar como uma interface seria anotada para a utilização do proxy. A listagem a seguir mostra a interface `Operacao` que possui as anotações em seus métodos. A ideia é que essa interface seja implementada por classes que armazenam um valor internamente e executem operações nesse valor, retornando o resultado. Nesse caso, métodos `somar()` e `multiplicar()` que retornam um valor baseado no valor interno e no parâmetro podem ter seus retornos armazenados no cache. Por outro lado, o método `mudar()` que altera o valor interno deve invalidar o cache, pois os retornos armazenados internamente deixam de ser válidos.

Anotações para marcar os métodos interceptados pelo proxy:

```
public interface Operacao<E> {
    @InvalidaCache public void mudar(E obj);
    @Cache public E somar(E obj);
    @Cache public E multiplicar(E obj);
}
```

Esse proxy desenvolvido nesse exemplo pode ser muito útil para classes que realizam chamadas remotas ou executam operações em uma base de dados. Nesse caso, chamadas que

recuperam dados baseadas no estado do banco ou do objeto remoto podem ser adicionadas no cache, e as que alteram esses estados devem ser utilizadas para invalidar o cache. De qualquer forma, antes de decidir utilizar um cache desse tipo em uma aplicação real, é importante fazer uma medição de desempenho baseada em cenários de uso do software, pois apesar de o cache poder acelerar o tempo médio de retorno de um método, os resultados armazenados ocupam memória. Sendo assim, para valer a pena, o ganho em velocidade deve compensar esse espaço utilizado.

E SE EU QUISER AS ANOTAÇÕES DO MÉTODO DA CLASSE E NÃO DA INTERFACE?

Uma observação importante em relação ao proxy dinâmico da API Reflection é que a instância de `Method` passada como parâmetro é da interface que está sendo encapsulada, e não do objeto que foi encapsulado. Como eles possuem a mesma assinatura, muitas vezes isso não tem muita importância, porém essa questão passa a ser relevante quando estamos buscando anotações, pois nesse caso importa se foram adicionadas na interface ou na classe. Porém, o fato de o método passado ser o da interface, não significa que devemos sempre utilizar anotações na interface. Quando os metadados forem referentes à abstração, devem ser adicionados na interface, porém, quando forem específicos da implementação, devem ser adicionados na classe.

Com base no método recebido como parâmetro, não é difícil de recuperar o mesmo método declarado na classe. Imaginando que a variável `obj` represente o objeto encapsulado e a variável `m` represente o recebido como parâmetro, o método da classe poderia ser recuperado com a chamada `obj.getClass().getMethod(m.getName(),m.getParameterTypes())`. Assim, as anotações poderiam ser facilmente recuperadas da declaração do método na classe e utilizadas no proxy.

4.6 OUTRAS FORMAS DE INTERCEPTAR MÉTODOS

"As laranjas não são as únicas frutas." - Título de um livro de Jeanette Winterson

Apesar de estar fora do escopo deste livro, acho importante dizer que existem outras formas de interceptar a execução de métodos em Java. Dependendo da plataforma para a qual você está desenvolvendo ou do tipo de componente que está sendo criado, a própria API pode fornecer uma alternativa para interceptação dos métodos. Nesse caso, os mesmos conceitos apresentados para a criação dos proxies dinâmicos vistos aqui podem ser utilizados. O objetivo desta seção é mostrar alguns exemplos dessas outras soluções que podem ser utilizadas na interceptação da execução de métodos.

Por exemplo, na plataforma Java EE, existe um tipo de componente chamado *interceptor* que funciona de forma similar a um proxy dinâmico, porém para instâncias que são gerenciadas pelo container Java EE. Para ser um interceptor, a classe precisa receber a anotação `@Interceptor` e possuir um método anotado com `@AroundInvoke`, que necessariamente precisa receber um parâmetro do tipo `InvocationContext` e retornar `Object`. A instância de `InvocationContext` é que contém todas as informações relacionadas à invocação, como o método interceptado e os parâmetros recebidos.

Exemplo de Interceptor da plataforma Java EE:

```
@Interceptor
public class ExemploInterceptor {
```

```
        @AroundInvoke
        public Object interceptaMetodo(InvocationContext ctx)
                throws Exception {
            //implementa lógica a ser acrescentada
        }
}
```

Como os objetos do container Java EE são criados por ele, não é necessário implementar nenhum método que insere o *interceptor* na classe. Uma forma de ligar um interceptor em uma classe é através da anotação `@Interceptors`, que recebe as classes de interceptors que devem atuar em uma classe ou em um método. Uma outra forma mais sofisticada é através da criação de anotações que fazem a ligação entre os interceptors e as classes interceptadas. Para criar essa anotação de ligação, ela precisa ser anotada com `@InterceptorBinding` e deve ser adicionada na própria classe do interceptor. Sendo assim, as classes e métodos que receberem essa anotação serão interceptadas.

Uma outra forma de interceptação de métodos é utilizando a programação orientada a aspectos. A orientação a aspectos possui outras coisas além da interceptação de métodos, porém aqui vamos focar mais exclusivamente nesta parte. A principal implementação da orientação a aspectos em Java é o **AspectJ**, o qual disponibiliza duas diferentes opções de sintaxe, sendo uma delas uma sintaxe própria e a outra baseada em anotações. A listagem a seguir exemplifica a sintaxe baseada em anotações.

Exemplo de aspecto com AspectJ utilizando a sintaxe de anotações:

```
@Aspect
public class AspectoRegistro{

    @Before("execution(* br.casadocodigo.*.*(..))")
```

```
    public void antes(JoinPoint jp) {
        //executa antes
    }
    @After("execution(* br.casadocodigo.*.*(..))")
    public void depois(JoinPoint jp) {
        //executa depois
    }
    @Around("execution(* br.casadocodigo.*.*(..))")
    public Object emVolta(ProceedingJoinPoint jp) throws
     Throwable{
        //executa antes
        Object returno = jp.proceed();
        //executa depois
        return returno;
    }
}
```

As anotações `@Before`, `@After` e `Around` configuram o método do aspecto, que nesse caso é chamado de adendo ou *advice*, para executar respectivamente antes, depois ou a volta do método interceptado. Cada anotação dessa recebe como parâmetro uma expressão que define um conjunto de junção, ou *pointcut*, que nada mais é que o conjunto de métodos que serão interceptados por aquele adendo do aspecto. O grande diferencial dos aspectos em relação às outras técnicas é que os métodos interceptados são definidos no aspecto, havendo um desconhecimento da aplicação da sua existência. A ligação das classes com os aspectos pode ocorrer na compilação, quando elas são carregadas, ou mesmo em tempo de execução com o uso do framework Spring. O capítulo *Ferramentas: indo além na reflexão* irá apresentar uma outra funcionalidade do AspectJ que pode ser utilizada para o mapeamento de anotações.

Quero mais uma vez deixar claro que esta seção foi mesmo para apresentar a existência de outras abordagens, que acabam sendo um pouco mais específicas. Não é o objetivo ensinar a

utilização dessas técnicas como no restante do capítulo. Acredito que entendendo melhor essas outras implementações é possível ter um panorama geral de como esse tipo de funcionalidade é implementado em diferentes contextos. Assim, se você estiver trabalhando com alguma plataforma ou framework que dê suporte a esse tipo de funcionalidade, vai saber mais facilmente os conceitos vistos aqui.

4.7 CONSIDERAÇÕES FINAIS

Os proxies dinâmicos são um recurso poderoso da reflexão que pode ser utilizado para adicionar comportamento em classes existentes, e ainda fazer isso de forma reutilizável para qualquer interface. Como foi visto, para implementar um proxy dinâmico não é muito complicado, bastando seguir uma receita. Este capítulo procurou explorar diferentes tipos de proxy dinâmico, desde o que não encapsula nenhum objeto até o que faz uso de metadados para diferenciar os métodos interceptados. No final do capítulo, mostrei rapidamente algumas outras abordagens para a interceptação de métodos.

Com este capítulo, finalizamos a primeira parte do livro onde os conceitos básicos de reflexão e do uso de metadados são apresentados. Nestes primeiros capítulos também foram apresentadas as APIs básicas em Java para a utilização desses conceitos. Chegando neste ponto, espero que os leitores já estejam familiares ao uso da reflexão e já consigam dar suas primeiras pinceladas em componentes que a utilizam. Prepare-se, pois na próxima parte do livro você vai aprender o que se precisa para fazer uma obra prima...

Boas práticas

Nos capítulos anteriores do livro foram abordadas as APIs e recursos da linguagem que podem ser empregados para o uso de reflexão da linguagem Java. Porém, se sua intenção é realmente criar um componente ou framework reutilizável com reflexão e metadados, conhecer as classes e métodos que serão utilizados é só o começo. Faço um paralelo de quando se aprende herança e polimorfismo em Java, pois saber somente isso não é suficiente para criar um bom design orientado a objetos.

Esta parte do livro apresenta boas práticas que podem ser utilizadas no desenvolvimento desse tipo de componente. Os próximos capítulos compilam de uma forma fluida e didática algumas práticas descobertas como parte da pesquisa que realizei e venho realizando. Essa pesquisa envolve não somente a análise das práticas utilizadas em diversos frameworks existentes, como também seu uso no desenvolvimento de novos frameworks para confirmar a adequabilidade da aplicação de cada uma delas.

O capítulo *Testando classes que usam Reflexão* aborda a criação de testes automatizados para classes e componentes que utilizam reflexão e metadados, trabalhando inclusive com o teste de proxies dinâmicos. Em seguida, o capítulo *Práticas no uso de anotações* demonstra práticas para a definição de metadados a partir de anotações, mostrando soluções para a representação de diferentes tipos de informação. Por fim, esta parte é finalizada com o capítulo *Consumindo e processando metadados*, que apresenta práticas para a estruturação de um componente ou framework baseado em

metadados, de forma a dar suporte à extensão do comportamento em diferentes pontos.

Capítulo 5

TESTANDO CLASSES QUE USAM REFLEXÃO

"Um desenvolvedor como eu se importa com escrever código novo e fazê-lo o mais interessante e eficiente possível. Mas muito poucas pessoas querem fazer os testes." - Linus Torvalds

Como foi dito anteriormente, existem vários erros que são pegos já em tempo de compilação que, quando lidamos com reflexão, podem acontecer em tempo de execução. Sendo assim, pelos próprios exemplos que foram mostrados nos capítulos anteriores, é possível perceber que, se a classe ou método não possui a estrutura esperada pelo componente, muitas coisas podem dar errado. Por exemplo, pode-se tentar instanciar uma classe com um construtor que ela não possui ou invocar um método com a quantidade errada de parâmetros. As classes desenvolvidas com reflexão devem estar preparadas para lidar com qualquer tipo de classe, e existem muitas coisas possíveis de acontecer. Uma premissa errada ou alguma estrutura inesperada pode causar um bug sério na aplicação que utiliza o componente.

O fato de um componente ou classe que utilize reflexão poder ser reutilizado em diferentes contextos dentro de uma aplicação, apesar de ser uma coisa muito boa, aumenta a preocupação com a

qualidade desse código. Muitas vezes, mesmo que a classe já tenha sido reutilizada várias vezes, uma situação distinta ainda pode causar erros. Além disso, é preciso não somente que o componente que utiliza reflexão esteja correto, como também que a classe da aplicação tenha a estrutura adequada e possua os metadados configurados de forma a gerar o comportamento desejado.

Dessa forma, este capítulo se dedica a abordar questões relativas ao teste de classes e componentes que utilizam reflexão, abordando também o teste das aplicações que utilizam esses componentes. Vou considerar que os leitores estejam familiarizados com os conceitos de testes automatizados e como criá-los com o JUnit 4 (para os que não conhecem, leiam o quadro). Porém, se você está lendo um livro avançado de programação como esse e não sabe criar testes de unidade ainda, acho que você está em apuros! Sugiro fortemente que corra atrás desse conhecimento, apesar de acreditar que com um conhecimento básico de JUnit é possível acompanhar o livro.

JUnit 4 em menos de um minuto!

Para criar testes de unidade com o JUnit 4, o primeiro passo é adicionar sua biblioteca no classpath. Para implementá-los, crie uma classe comum e adicione métodos que realizam os testes. Para ser considerado um teste, o método precisa ser anotado com `@Test`. Cada teste deve preparar o cenário do teste, executar a funcionalidade cujo comportamento deseja-se testar e verificar se foi o esperado. Para isso, a classe `Assert` possui diversos métodos estáticos que permitem fazer comparações entre valores. A listagem a seguir mostra um exemplo simples de classe de teste.

Exemplo simples de funcionamento do JUnit:

```
public class ClassDeTeste {
    @Test
    public void metodoDeTeste() {
        ClassTestada testada = new ClassTestada();
        String resultado = testada.metodoTestado();
        assertEquals("valor esperado",resultado);
    }
}
```

O JUnit possui um grande suporte a todos os IDEs existentes, com o qual, para rodar uma classe de testes, basta clicar nela e solicitar sua execução. Isso é o suficiente para acompanhar os exemplos! Caso algo diferente seja utilizado, a própria seção irá explicar o funcionamento.

Para mostrar como fazer os testes automatizados desse tipo de classe, este capítulo vai focar muito em exemplos. No caso, vamos

pegar classes desenvolvidas nos capítulos anteriores e mostrar como poderia ser feito o teste para elas. Os exemplos escolhidos possuem características distintas, justamente para mostrar a diversidade de situações que se pode encontrar ao se testar esse tipo de componente de software.

5.1 VARIANDO ESTRUTURA DA CLASSE PARA TESTE

Quando temos uma classe ou componente baseado em reflexão e metadados, normalmente o seu comportamento vai depender da classe passada como parâmetro. Sendo assim, para testá-los é preciso passar classes com diferentes estruturas como parâmetro. Dessa forma, o cenário de cada teste vai envolver a criação de uma nova classe e, quando cabível, vai criar um objeto dessa classe. A ação que define a funcionalidade nesse teste envolve a invocação do componente que utiliza reflexão. Por fim, a verificação irá checar se o comportamento foi o esperado, o que pode envolver a verificação do valor de um retorno, do estado de um objeto, ou mesmo se um determinado método foi invocado.

Para começar, vamos implementar os testes automatizados do gerador de mapas apresentado na seção *O primeiro contato com a API Reflection*. Para esse componente, o teste mais simples seria gerar o mapa de propriedades de um Java Bean simples, com algumas propriedades e sem anotações. A próxima listagem mostra como esse primeiro teste seria implementado. Inicialmente, definimos a classe **Bean** que será consumida pela classe de geração de mapas. Observe que ela possui dois atributos com métodos de acesso getters e setters. Em seguida é criada uma instância dessa

classe, as suas propriedades são configuradas e, por fim, o objeto é passado ao método `gerarMapa()`. Para verificar se o comportamento foi correto, são inseridas duas asserções que verificam se o mapa retornado possui as propriedades esperadas.

Teste da funcionalidade básica do gerador de mapas:

```
public class TesteGeradorMapa {
    @Test
    public void mapaDeClasseSimples() {

        class Bean{
            private String prop1;
            private int prop2;
            public String getProp1() {
                return prop1;
            }
            public void setProp1(String prop1) {
                this.prop1 = prop1;
            }
            public int getProp2() {
                return prop2;
            }
            public void setProp2(int prop2) {
                this.prop2 = prop2;
            }
        }

        Bean b = new Bean();
        b.setProp1("teste");
        b.setProp2(25);
        Map<String,Object> mapa = GeradorMapa.gerarMapa(b);
        assertEquals("teste",mapa.get("prop1"));
        assertEquals(25,mapa.get("prop2"));
    }
}
```

Como foi descrito, é preciso definir uma classe para a realização da verificação. No exemplo mostrado, ela é criada dentro do próprio método de teste, sendo válida sua utilização somente dentro de seu escopo. As principais vantagens dessa

abordagem é que a classe definida fica próxima ao seu uso, e que o mesmo nome pode ser utilizado em outros métodos sem haver conflito. Porém, essa é uma das possibilidades, e ela pode ser definida em outros escopos, como fora do método ou em seu próprio arquivo. Uma classe definida em um contexto mais amplo pode ser utilizada em mais métodos de testes, mas por outro lado, a definição dentro do mesmo arquivo aumenta a legibilidade do código, tornando mais fácil de ver o cenário que está sendo definido. Sendo assim, é preciso pesar essas questões para ver qual o local mais adequado para sua definição.

Continuando a definição dos testes, a próxima listagem verifica a utilização da anotação `@Ignorar`. A diferença na definição da classe `Bean` para esse teste está na adição da anotação `@Ignorar` no método `getProp1()`. Observe que a criação do objeto e a definição da propriedade são iguais às do método anterior. Nas asserções, é verificado se o mapa não possui a chave `"prop1"` através do método `assertFalse()`, e a outra verificação é a mesma que já havíamos feito.

Testando o uso de anotação @Ignorar:

```
@Test
public void ignoraPropriedade() {

    class Bean{
        private String prop1;
        private int prop2;
        @Ignorar
        public String getProp1() {
            return prop1;
        }
        public void setProp1(String prop1) {
            this.prop1 = prop1;
        }
        public int getProp2() {
```

```
            return prop2;
        }
        public void setProp2(int prop2) {
            this.prop2 = prop2;
        }
    }

    Bean b = new Bean();
    b.setProp1("teste");
    b.setProp2(25);
    Map<String,Object> mapa = GeradorMapa.gerarMapa(b);
    assertFalse(mapa.containsKey("prop1"));
    assertEquals(25,mapa.get("prop2"));
}
```

Por fim, o teste seguinte exercita o caso em que a anotação `@NomePropriedade` é utilizada para mudar o nome da chave usada no mapa. Observe que, neste caso, as asserções verificam se existe a entrada no mapa com o nome configurado na anotação e se não existe a chave com o nome do atributo.

Testando a mudança de nome da propriedade do mapa:

```
@Test
public void mudaNomePropriedade() {

    class Bean{
        private String prop1;
        private int prop2;
        public String getProp1() {
            return prop1;
        }
        public void setProp1(String prop1) {
            this.prop1 = prop1;
        }
        @NomePropriedade("propriedade2")
        public int getProp2() {
            return prop2;
        }
        public void setProp2(int prop2) {
            this.prop2 = prop2;
        }
```

```
    }

    Bean b = new Bean();
    b.setProp1("teste");
    b.setProp2(25);
    Map<String,Object> mapa = GeradorMapa.gerarMapa(b);
    assertEquals("teste",mapa.get("prop1"));
    assertEquals(25,mapa.get("propriedade2"));
    assertFalse(mapa.containsKey("prop2"));
}
```

Esse exemplo demonstra o processo básico de se testar uma classe que utiliza reflexão. Como foi mostrado nesses três testes, a criação do objeto e a invocação dos métodos acabam sendo as mesmas, e o que muda no processo de teste é a estrutura da classe passada como parâmetro. A ideia é utilizar uma classe como base e ir modificando a estrutura de forma a mudar o comportamento esperado.

Verificando chamadas de método

Um tipo de funcionalidade frequentemente feita por classes que utilizam reflexão envolve a invocação de métodos em classes que ela só reconhece em tempo de execução. Sendo assim, um teste que precisará ser feito para validar esse tipo de funcionalidade envolve, como parte da verificação, assegurar que os métodos corretos foram invocados. Isso implica não somente em verificar a invocação dos métodos certos, mas também em se certificar de que outros métodos não foram invocados.

Aqui, utilizaremos como exemplo o validador de objetos apresentado da seção *Procurando métodos e atributos para validação*. Uma das funcionalidades dessa classe envolve a invocação de métodos começados com `"validar"` e que não recebam parâmetros. Sendo assim, a próxima listagem mostra dois

testes para verificar essa funcionalidade. Observe que o primeiro teste verifica se o método correto é invocado, e o segundo verifica se somente métodos que obedecem aos requisitos são invocados. Para verificar se a invocação é feita ou não, é criada uma variável booleana na classe para teste, que é alterada no momento em que o método é invocado. Sendo assim, vendo o valor dessa variável conseguimos ver se o método foi invocado ou não.

Testando a invocação de métodos no validador:

```
public class TesteValidador {

    @Test
    public void testInvocaMetodo() throws ValidacaoException {
        class ParaValidar{
            public boolean invocou = false;
            public void validarInformacao(){
                invocou = true;
            }
        }
        ValidadorObjetos v = new ValidadorObjetos();
        ParaValidar pv = new ParaValidar();
        v.validarObjeto(pv);
        assertTrue(pv.invocou);
    }
    @Test
    public void naoInvocaMetodo() throws ValidacaoException {
        class ParaValidar{
            public boolean invocouA = false;
            public boolean invocouB = false;
            public void validarParamErrado(String s){
                invocouA = true;
            }
            public void nomeErrado(String s){
                invocouB = true;
            }
        }
        ValidadorObjetos v = new ValidadorObjetos();
        ParaValidar pv = new ParaValidar();
        v.validarObjeto(pv);
        assertFalse(pv.invocouA);
```

```
        assertFalse(pv.invocouB);
    }
}
```

Uma outra funcionalidade da classe `ValidadorObjetos` envolve a invocação de atributos cujo tipo implemente a interface `Validador`. Nesse caso, a ideia é a mesma do teste mostrado anteriormente, porém, como a invocação é feita no objeto que está no atributo, o mecanismo deve ser adicionado nesse objeto, e não na classe que está sendo criada. A listagem a seguir mostra a implementação do teste dessa funcionalidade.

Observe que a classe `ValidadorTeste` é criada para simular uma classe que implementa a interface `Validador` e possui um atributo booleano para verificar se o método foi invocado ou não. Em seguida, dentro do método de teste, é criada uma classe com três atributos, dos tipos `ValidadorTeste`, `Validador` e `Object`, sendo que todos recebem uma instância de `ValidadorTeste`. Segundo os requisitos, para ser invocado, o tipo do atributo deve implementar a interface `Validador`, o que é refletido pelas asserções, onde somente o atributo do tipo `Object` não deve ser invocado.

Teste da invocação de atributos do tipo Validador:

```
//Declaração fora da classe
public class ValidadorTeste implements Validador {
    public boolean invocou;
    public void validar(Object o) throws Exception {
        invocou = true;
    }
}

//método de teste
@Test
public void invocaAtributos() throws ValidacaoException {
    class ParaValidar{
```

```
        public ValidadorTeste v1 = new ValidadorTeste();
        public Validador v2 = new ValidadorTeste();
        public Object v3 = new ValidadorTeste();
    }
    ValidadorObjetos v = new ValidadorObjetos();
    ParaValidar pv = new ParaValidar();
    v.validarObjeto(pv);
    assertTrue(pv.v1.invocou);
    assertTrue(((ValidadorTeste)pv.v2).invocou);
    assertFalse(((ValidadorTeste)pv.v3).invocou);
}
```

Posso utilizar mock objects nesse tipo de teste?

Se você é um desenvolvedor familiar com a criação de testes, talvez deva estar se perguntando se essa criação de classes mostrada nas listagens anteriores não seria o mesmo que criar um *mock object*. Um mock object é um objeto que é criado para simular uma dependência da classe que está sendo testada. Isso envolve dar as respostas para simular diferentes cenários de teste e verificar se as invocações de método foram feitas de acordo com as esperadas. Em uma classe que não utiliza reflexão, a dependência possui um contrato esperado, determinado por uma classe ou uma interface. Sendo assim, o mock object implementa esse mesmo contrato de forma a poder substituir essa dependência.

No caso do teste de classes que utilizam reflexão, o objeto utilizado como dependência não possui uma interface definida. Nesse caso, o que precisa ser feito em cada caso de teste é a definição de classes com estruturas diferentes, e não comportamentos diferentes para a mesma estrutura. Apesar de isso não ser um termo tão usual, um artigo científico definiu esse tipo de classe criada para teste como *mock class* (FERNANDES; GUERRA; SILVEIRA, 2010). Sendo assim, o mock object se refere à simulação de comportamento para a dependência da classe

testada e mock class, à simulação de estruturas diferentes de classes para teste. De certa forma, é possível criar um mock object de um mock class, pois testes distintos podem usar uma mesma estrutura de classe mas com diferentes comportamentos. Isso é comum quando a mock class representa uma interface e o mock object é uma implementação dela.

Nos exemplos mostrados, cada teste criava sua mock class já com o comportamento esperado para o teste. Muitas vezes, como uma nova classe precisará ser definida de qualquer forma, é mais fácil já defini-la fazendo o necessário para gerar o cenário do teste. O melhor exemplo de mock object nos testes já apresentados neste capítulo foi a classe `ValidadorTeste` , pois, nesse caso, a interface esperada é fixa, por mais que os atributos sejam lidos por reflexão.

No caso dos mock objects, uma alternativa à implementação manual de novas classes para a simulação de comportamento são os frameworks de criação de mocks. Eles normalmente utilizam proxies dinâmicos para gerar o comportamento esperado pela classe e verificar se as chamadas realizadas foram as esperadas. Exemplos de frameworks Java que implementam esse tipo de funcionalidade são o **JMock**, o **EasyMock** e o **Mockito**. A vantagem em utilizá-los está em não necessitar criar diversas classes para simular diferentes comportamentos para os testes, permitindo que a definição do comportamento e verificações do mock sejam feitas no próprio método de teste.

Os exemplos aqui apresentados irão utilizar o JMock para mostrar a utilização desse tipo de framework em testes de classe que utilizam reflexão, porém está fora do escopo deste livro explicar o funcionamento desse framework.

JMock em menos de cinco minutos!

Para utilizar o JMock, a classe de teste deve possuir algum atributo que tenha um tipo derivado da classe `Mockery`, que será utilizado na criação dos mocks. Em versões mais antigas do JUnit, a classe de teste era anotada com `@RunWith(JMock.class)` para que as verificações dos mocks rodassem depois do método de teste. Em versões mais novas, basta criar um atributo do tipo `JUnitRuleMockery` e o anotar com `@Rule`. Para criar o mock de uma interface basta utilizar o método `mock()` passando o `Class` correspondente, sendo que se for criar mais de um mock da mesma classe precisa-se passar um parâmetro adicional com uma `String` que nomeia a instância do mock.

Para definir as expectativas e comportamento do mock, a chamada `checking(new Expectations(){{ ... }})` é utilizada, sendo que todas as definições são feitas no lugar dos três pontos. O JMock utiliza uma DSL interna para definir as chamadas esperadas e seu comportamento, incluindo o número de invocações, ordem de invocação, parâmetros esperados e o resultado da invocação, como retorno ou lançamento de exceção. As expectativas utilizadas em cada exemplo do livro serão explicadas com detalhes, possibilitando que mesmo quem não conheça o framework possa acompanhá-los. No site do projeto JMock, existe um resumo de sua sintaxe em uma *cheat sheet* no endereço http://jmock.org/cheat-sheet.html. A listagem a seguir mostra a estrutura básica de uso do JMock.

Exemplo de uso do JMock:

```
public class TesteComJMock {

    @Rule
```

```
    public JUnitRuleMockery ctx = new JUnitRuleMockery();

    @Test
    public void testeComMock() throws Exception {
        ClasseMock mock= ctx.mock(ClasseMock.class);
        ClasseTestada c = new ClasseTestada();
        c.setClasseMock(mock);

        ctx.checking(new Expectations() {{
            //expectativas do mock
            //retornos dados pelo mock
        }});

        c.metodoTestado();
    }
}
```

Para vermos como o teste usando mock object pode ser feito simulando os objetos da interface `Validador`, a próxima listagem mostra como o framework JMock poderia ser utilizado. Para que os mocks possam ser criados e suas expectativas verificadas no final de cada teste, a classe de teste deve ter um atributo do tipo `JUnitRuleMockery` anotada com `@Rule`. No método de teste, observe que a mock class `ParaValidar` é criada recebendo mock objects em seus dois atributos, a partir da chamada do método `mock()` da class do JMock. Em seguida, são definidas as expectativas: o método `validar()` do atributo de tipo `Validador` deve ser invocado uma vez e o método `validar()` do atributo de tipo `Object` não deve ser invocado.

Uso de framework de mock para criação de teste com Validador:

```
public class TesteValidador {

    @Rule public JUnitRuleMockery ctx = new JUnitRuleMockery();

    @Test
```

```
    public void invocaAtributosJMock() throws Exception {

        class ParaValidar{
            public Validador v1 =
                ctx.mock(Validador.class, "v1");
            public Object v2 = ctx.mock(Validador.class, "v2");
        }

        ValidadorObjetos v = new ValidadorObjetos();
        final ParaValidar pv = new ParaValidar();

        ctx.checking(new Expectations() {{
            oneOf (pv.v1).validar(pv);
            never (((Validador)pv.v2)).validar(pv);
        }});

        v.validarObjeto(pv);
    }
}
```

Lidando com testes que envolvem exceções

As exceções são um tipo de retorno que uma invocação de método pode dar como resultado, significando a existência de algum tipo de erro na execução da funcionalidade. A exceção lançada pode indicar uma situação em que a funcionalidade não pode ser executada, mas que faz parte de um cenário comum de negócio. Exemplos são informações inválidas fornecidas pelo usuário, uma autenticação falha ou a tentativa de acesso a uma funcionalidade à qual o usuário não é autorizado. A ocorrência dessas exceções não significa que existe um erro no software, mas que uma execução da funcionalidade não pôde ser completada devido a uma situação excepcional prevista.

Outro tipo de exceção é aquele que indica que o software não está funcionando corretamente devido a um problema interno, que representa um erro ou um bug. Muitas classes e componentes

possuem uma abordagem defensiva, segundo a qual não confiam em outras classes e verificam se as chamadas estão sendo feitas da forma correta. Nesse caso, o lançamento de uma exceção significa que o uso daquela classe não está sendo feito da forma correta. É importante que a mensagem de erro seja bem descritiva para possibilitar que o desenvolvedor detecte a falha e possa corrigi-la. Em caso de classes que utilizam reflexão, esse tipo de verificação é muito importante pois a classe recebida como parâmetro pode possuir uma estrutura inadequada ou erros na configuração de metadados.

Para exemplificar o primeiro tipo de exceção, serão utilizados os próprios testes da classe `ValidadorObjetos()`. Como foi mostrado em seu desenvolvimento, essa classe deve pegar as exceções lançadas pelos métodos e classes de validação, encapsulando-as em uma lista que é lançada em uma `ValidaçãoException`. A listagem a seguir mostra um teste que simula um cenário em que uma exceção é lançada de um método de validação, chamado `validarInformacao()`, e uma implementação de `Validator` da qual é criado um mock com o framework JMock. Observe que, para fazer o mock lançar a exceção, nas expectativas é utilizada a chamada `will(throwException(...))`.

Verificando informações das exceções recebidas:

```
@Test
public void capturaExcecoes() throws Exception {

    class ParaValidar{
        public Validador v = ctx.mock(Validador.class);
        public void validarInformacao() throws Exception{
            throw new Exception("mensagem A");
        }
```

```
    }

    ValidadorObjetos v = new ValidadorObjetos();
    final ParaValidar pv = new ParaValidar();

    ctx.checking(new Expectations() {{
        oneOf (pv.v).validar(pv);
        will(throwException(new Exception("mensagem B")));
    }});

    try {
        v.validarObjeto(pv);
        fail();
    } catch (ValidacaoException e) {
        List<Exception> erros = e.getErros();
        assertEquals("mensagem A", erros.get(0).getMessage());
        assertEquals("mensagem B", erros.get(1).getMessage());
    }
}
```

Nesse caso, como existem verificações para serem feitas na exceção lançada, o método `validarObjeto()` precisa ser envolto em um bloco `try`. Como é esperado que uma exceção seja lançada, o método `fail()` é invocado logo após a chamada de `validarObjeto()`, pois se ela não for lançada o teste deve falhar. Dentro do bloco `catch`, a lista de exceções é recuperada e em seguida é verificado, através da mensagem de erro, se as exceções lançadas pelo método da classe e pelo mock estão na lista.

Como exemplo de teste de exceções que são lançadas por uma estrutura inadequada da classe, serão utilizados testes da `MapeamentoParametros` apresentada na seção *Mapeando parâmetros de linha de comando para uma classe*. Esse exemplo foi escolhido porque foram feitas várias verificações no formato da classe, lançando exceções quando a sua estrutura não condizia com o que era esperado. A listagem a seguir mostra o exemplo de dois testes que fazem essa verificação. O primeiro verifica o caso em

que o tipo da propriedade seja inteiro, sendo que o esperado é somente `String` , e o segundo explora o caso de o método setter receber dois parâmetros.

Exemplo de testes que verificam classes com estrutura inadequada:

```
public class TesteMapeamentoParametros {

    @Test(expected=MapeamentoException.class)
    public void testTipoErrado() {

        class Param{
            @Parametro("-i")
            private int info;
            public int getInfo() {
                return info;
            }
            public void setInfo(int info) {
                this.info = info;
            }
        }

        String[] a = {"-i","23"};
        MapeamentoParametros<Param> map =
            new MapeamentoParametros<>(Param.class);
        Param p = map.mapear(a);
    }

    @Test(expected=MapeamentoException.class)
    public void testQuantidadeParamErrado() {

        class Param{
            @Parametro("-i")
            private String[] info;
            public String[] getInfo() {
                return info;
            }
            public void setInfo(String infoA, String infoB) {
                this.info = new String[]{infoA, infoB};
            }
        }
```

```
        String[] a = {"-i","textoA","textoB"};
        MapeamentoParametros<Param> map =
            new MapeamentoParametros<>(Param.class);
        Param p = map.mapear(a);
    }
}
```

Nesse caso, não existe nenhuma verificação adicional que precisa ser feita em relação às informações da exceção lançada pelo método `mapear()`. Sendo assim, pode ser utilizado o parâmetro `expected` da anotação `@Test`. Nele, deve ser passada a classe da exceção que se espera que seja lançada. Dessa forma, o teste só irá passar caso a exceção seja lançada dentro do método de teste. Uma ressalva em relação ao uso do atributo `expected` é que o teste irá passar se a exceção for lançada em qualquer ponto do método de teste. Portanto, recomenda-se utilizar essa abordagem apenas no caso de a exceção poder ser lançada em apenas uma das chamadas de método do teste.

Finalizo esta seção ressaltando a importância da realização desse tipo de teste para classes que utilizam reflexão. É importante verificar como o componente está lidando com diferentes estruturas de classe, seja fazendo algum tratamento especial, ou lançando uma exceção informativa dizendo o motivo de aquela classe não ser aceita. Quando a classe precisar lidar com tipos, é sempre importante incluir testes envolvendo tipos primitivos e arrays, pois eles, na maioria das vezes, requerem um tratamento especial. A criação de testes automatizados que abrangem diversos cenários diferentes é primordial para aumentar a qualidade da implementação, o que é especialmente importante se ela for reutilizada em diferentes contextos.

PARA QUE COMPLICAR SE O TESTE PODE SER SIMPLES?

Os exemplos apresentados para os testes até o momento neste livro possuem a estrutura mais básica possível para exercitar a funcionalidade desejada. Pode parecer que isso está sendo feito para manter as listagens pequenas ou para poder ser mais didáticos, porém na verdade é uma boa prática que os testes sejam simples e objetivos dentro da funcionalidade que está sendo exercitada. Assim, por que criar um Java Bean com diversas propriedades, se duas já são o suficiente? O ideal é, a princípio, manter cada teste simples e focado em apenas uma funcionalidade, e posteriormente criar um teste mais complexo que envolve a execução de diversas funcionalidades combinadas.

5.2 TESTE DE PROXIES DINÂMICOS

Um outro tipo de teste importante quando trabalhamos com reflexão é o de proxies dinâmicos. A dificuldade de teste do proxy dinâmico está em sua própria natureza de encapsular um outro objeto. Sendo assim, uma importante parte dele envolve a invocação de métodos nesse objeto encapsulado. Desse modo, dependendo da funcionalidade do proxy, é importante ver quais e quantas chamadas foram realizadas, quais parâmetros foram passados à classe encapsulada e até mesmo simular diferentes cenários quando ela retornar diferentes valores e lançar exceções.

Vale ressaltar que, como o proxy é dinâmico, ele precisa estar

preparado para lidar com diferentes tipos de classe. Portanto, devemos considerar que os métodos que vão ser recebidos pelo proxy podem possuir qualquer tipo de estrutura, com diferentes quantidades e tipos de parâmetros e diferentes tipos de retorno. Caso você decida não dar suporte a determinados tipos de método, é importante verificar isso e lançar um erro se a interface possuir um método em um formato não aceito.

A estratégia básica para o teste de um proxy dinâmico é criar uma interface que capture uma característica que se queira testar e criar um mock object dessa interface. Em seguida, criar o proxy encapsulando o mock e utilizá-lo para as verificações relacionadas às invocações de método do proxy no objeto encapsulado.

Para exemplificar esse tipo de teste, esta seção apresenta testes da classe `CacheProxy` mostrada na seção *Consumindo anotações em proxies*. Neste caso, o mock irá ajudar a verificar as situações em que o objeto encapsulado é invocado e os casos em que o resultado é obtido do cache armazenado pelo proxy. Para iniciar o teste, a listagem a seguir apresenta a interface `CacheMe` , que irá servir como a mock class para os testes criados.

Verificando informações das exceções recebidas:

```
public interface CacheMe{
    @Cache int metodoComCache(int param);
    @InvalidaCache void anulaCache();
}
```

O primeiro teste implementado no método `cacheSimples()` da próxima listagem verifica se o proxy armazena e retorna o valor do cache corretamente para chamadas de métodos com diferentes parâmetros. As expectativas do mock object é que `metodoComCache()` seja invocado uma vez com o parâmetro `1` e

uma vez com o parâmetro `2`, sendo que ele vai retornar respectivamente os valores `10` e `20`. Como ação do teste, o método `metodoComCache()` será invocado duas vezes com cada parâmetro e precisa receber os valores corretos como retorno. Dessa forma, será verificado se o método do mock object foi invocado somente na primeira vez, e se o cache armazenado foi retornado corretamente.

Verificando informações das exceções recebidas:

```
public class TesteCacheProxy {

    @Rule public JUnitRuleMockery ctx = new JUnitRuleMockery();

    @Test
    public void cacheSimples() {
        final CacheMe mock = ctx.mock(CacheMe.class);
        CacheMe proxy = (CacheMe) CacheProxy.criar(mock);

        ctx.checking(new Expectations() {{
            oneOf (mock).metodoComCache(1);
            will(returnValue(10));
            oneOf (mock).metodoComCache(2);
            will(returnValue(20));
        }});

        assertEquals(10, proxy.metodoComCache(1));
        assertEquals(20, proxy.metodoComCache(2));
        assertEquals(10, proxy.metodoComCache(1));
        assertEquals(20, proxy.metodoComCache(2));
    }
}
```

O segundo método de teste verifica a invocação de um método que anula o cache armazenado pelo proxy e está apresentado na listagem a seguir. Desta vez, iremos invocar o método `metodoComCache()` duas vezes, e em seguida, o método `anulaCache()` e, por fim, o método `metodoComCache()` mais

duas vezes. A expectativa é que `metodoComCache(0)` seja invocado apenas duas vezes no mock object, retornando o valor `10` na primeira vez e `20` da segunda vez, isso além da chamada do método `anulaCache()`. Observe que existem asserções que verificam se as duas primeiras chamadas retornam `10` e se as duas últimas retornam `20`. Esse teste verifica se o cache é limpo depois da chamada de um método anotado com `@InvalidaCache`.

Verificando informações das exceções recebidas:

```
@Test
public void invalidaCache() {
    final CacheMe mock = ctx.mock(CacheMe.class);
    CacheMe proxy = (CacheMe) CacheProxy.criar(mock);

    ctx.checking(new Expectations() {{
        exactly(2).of(mock).metodoComCache(1);
        will(onConsecutiveCalls(returnValue(10),
                                returnValue(20)));
        oneOf(mock).anulaCache();
    }});

    assertEquals(10, proxy.metodoComCache(1));
    assertEquals(10, proxy.metodoComCache(1));
    proxy.anulaCache();
    assertEquals(20, proxy.metodoComCache(1));
    assertEquals(20, proxy.metodoComCache(1));
}
```

É importante ficar claro que esses dois testes apenas ilustram a abordagem para o teste de um proxy dinâmico. Em uma suíte de testes mais completa para esse proxy, seriam testados cenários mais complexos, como uma interface que possui métodos com o mesmo nome, métodos com vários parâmetros, métodos com quantidade variável de parâmetros, retorno e parâmetros com objetos complexos, entre outros. Por exemplo, imagine dois métodos com mesmo nome que recebem respectivamente um e

dois inteiros como parâmetro. Será que o proxy usaria o valor do retorno da chamada com parâmetros `(1,1)` para a chamada com `(11)`? Deixo como exercício para o leitor implementar esse teste, e corrigir a implementação do proxy se necessário.

5.3 TESTANDO A CONFIGURAÇÃO DE METADADOS

As seções anteriores deste capítulo abordaram a criação de testes que verificam os componentes ou classes que utilizavam reflexão e metadados. Porém, quem garante que esse componente vai se comportar conforme o desejado para uma determinada classe? Por exemplo, se você quer que os parâmetros da sua aplicação sejam mapeados para uma classe, é preciso saber se isso está sendo feito da forma correta. Da mesma forma, se você deseja que o cache seja feito para uma determinada classe utilizando a `CacheProxy`, é preciso verificar se a estrutura criada para a classe e as anotações configuradas refletem esse comportamento.

Quando estamos utilizando um componente que utiliza reflexão e metadados, a própria estrutura da classe e os metadados vão determinar o seu comportamento quando essa classe ou um de seus objetos forem passados como parâmetro. Por mais que pareça estranho testarmos algo declarativo, é importante saber se os metadados estão declarados de forma a gerar o comportamento adequado. Enquanto os testes da classe ou do componente são feitos pelos seus desenvolvedores, esse tipo de teste dos metadados deve ser feito por quem o está utilizando.

Como testar o comportamento gerado pelos

metadados

Para criarmos um teste que verifica se os metadados estão configurados corretamente, deve ser checado o comportamento da classe que utiliza reflexão ao receber uma determinada classe da aplicação. O teste é bem parecido com os que foram mostrados nas seções anteriores, com a diferença de que não será utilizada uma mock class e sim uma classe real da aplicação. O objetivo também é diferente, pois não queremos verificar se a classe que utiliza reflexão funciona corretamente quando recebe uma classe com uma determinada estrutura e anotações; mas sim, assumindo que o componente foi testado e funciona corretamente, deseja-se verificar quais são a estrutura e as anotações que a classe precisaria ter para se obter um determinado comportamento.

Imagine que, em minha aplicação, eu precise receber como parâmetros na linha de comando uma lista de arquivos de entrada e apenas um arquivo de saída. Para facilitar a implementação da leitura desses parâmetros, será utilizado o componente de mapeamento de parâmetros apresentado na seção *Mapeando parâmetros de linha de comando para uma classe*. Sendo assim, o teste envolverá invocar a classe `MapeamentoParametros` com a classe da minha aplicação e ver se ela recebe as informações da forma correta. A listagem a seguir mostra o teste que poderia ser feito para essa verificação.

Verificando se o mapeamento foi feito de forma correta:

```
@Test
public void arquivosEntradaSaida() {
    String[] a =
        {"-in","entradaA.txt","entradaB.txt","-out",
         "saida.txt"};
    MapeamentoParametros<ParametrosArquivo> map =
```

```
            new MapeamentoParametros<>(ParametrosArquivo.class);
    ParametrosArquivo p = map.mapear(a);

    assertEquals("entradaA.txt", p.getArquivosEntrada()[0]);
    assertEquals("entradaB.txt", p.getArquivosEntrada()[1]);
    assertEquals("saida.txt", p.getArquivoSaida());
}
```

Na posição de quem está implementando a aplicação, é desejável que seja lançada uma exceção quando forem passados mais de um arquivo de saída. Dessa forma, mais um método de teste é adicionado para verificar esse cenário. A listagem a seguir mostra a verificação para saber se uma exceção é lançada no caso de dois arquivos de saída serem passados como parâmetro.

Verificando se o componente lança exceção com parâmetros errados:

```
@Test(expected=MapeamentoException.class)
public void maisDeUmArquivoDeSaida() {
    String[] a =
        {"-in","entradaA.txt","-out","saidaA.txt","saidaB.txt"};
    MapeamentoParametros<ParametrosArquivo> map =
        new MapeamentoParametros<>(ParametrosArquivo.class);
    ParametrosArquivo p = map.mapear(a);
}
```

Uma questão importante nesse tipo de teste é que a verificação é feita de acordo com os requisitos da aplicação. Sendo assim, o comportamento desejado pode ser obtido através de uma combinação entre metadados e o comportamento do componente que utiliza reflexão. Por exemplo, imagine que essa aplicação possua um parâmetro opcional para o timeout do processamento, que é do tipo inteiro, porém o componente só aceita parâmetros do tipo `String`, array de `String` e booleano. A listagem a seguir mostra como seria o teste e a implementação para esse exemplo.

Verificando se o componente lança exceção com parâmetros errados:

```
@Test
public void parametroTimeout() {
    String[] a = {"-time","5000"};
    MapeamentoParametros<ParametrosArquivo> map =
            new MapeamentoParametros<>(ParametrosArquivo.class);
    ParametrosArquivo p = map.mapear(a);
    assertEquals(5000, p.getTimeout());
}

//implementação
public class ParametrosArquivo {

    @Parametro("-time")
    private int timeout;

    public void setTimeout(String timeout) {
        this.timeout = Integer.parseInt(timeout);
    }
}
```

Observe que o teste expressa somente o comportamento esperado, não se importando se a funcionalidade será implementada somente pelo componente ou se vai precisar ser complementada nas classes da aplicação. No caso, é esperado que seja retornado um valor inteiro pelo método de acesso na classe `ParametrosArquivo`. Sendo assim, para gerar esse resultado esperado, como mostrado na listagem, essa classe precisaria fazer a conversão do valor no momento que recebesse a `String` da classe `MapeamentoParametros`.

Como reutilizar código de teste

Quando utilizamos componentes e classes baseados em reflexão e metadados, é possível reutilizá-los para executar

funcionalidades para diversas classes da aplicação. Apesar de esse reúso poupar tempo de desenvolvimento, os testes do funcionamento do componente para cada classe da aplicação podem ser trabalhosos e tomar muito tempo. Porém, se avaliarmos o teste do mesmo componente para diferentes classes, é possível perceber que eles possuem muito em comum, como a chamada do próprio método do componente. Nesta seção, vamos ver como é possível reutilizar código desse tipo de teste, para que se possa poupar tempo de desenvolvimento também nessa etapa.

A listagem a seguir mostra o exemplo de um teste que verifica a geração do mapa para a classe `Produto`. Neste caso, o atributo `descricao` precisa ser ignorado e o atributo `categoria` deve ser inserido no mapa como `"tipo"`. Observe que as asserções verificam as informações que estão no mapa e os atributos que não devem estar no mapa. Como já foi dito, é importante lembrar que o que está sendo testado não é o `GeradorMapa`, mas os metadados configurados.

Testando a geração do mapa para a classe Produto:

```
public class TesteMapaProduto {

    @Test
    public void geracaoMapa() {
        Produto p = new Produto();
        p.setNome("Tablet APX10");
        p.setPreco(500.00);
        p.setDescricao("10 pol, 16GB, 800x600");
        p.setCategoria("Eletrônicos");

        Map<String,Object> mapa = GeradorMapa.gerarMapa(p);

        assertEquals(mapa.get("nome"), "Tablet APX10");
        assertEquals(mapa.get("preco"), 500.00);
        assertEquals(mapa.get("tipo"), "Eletrônicos");
        assertFalse(mapa.containsKey("descricao"));
```

```
        assertFalse(mapa.containsKey("categoria"));
    }
}
```

Para ser possível reutilizar a classe de teste, o primeiro passo é separar a parte dele que é mais geral do que a que é específica para a classe `Produto`. A ideia é que métodos auxiliares forneçam as informações específicas para serem utilizadas em um método de teste genérico. Na listagem a seguir, isso foi feito para a classe de teste mostrada anteriormente. O método `getObjeto()` cria e popula o objeto que será utilizado para o teste, e os métodos `getConteudoMapa()` e `getExcluidos()` retornam respectivamente as entradas esperadas para o mapa e as chaves que não devem estar presentes no mapa.

Separando a parte geral do teste da parte específica para a classe Produto:

```
public class TesteMapaProduto {

    @Test
    public void geracaoMapa() {
        Object o = getObjeto();
        Map<String,Object> mapa = GeradorMapa.gerarMapa(o);
        Map<String,Object> esperado = new HashMap<>();
        getConteudoMapa(esperado);
        for(String s : esperado.keySet()){
            assertEquals(esperado.get(s), mapa.get(s));
        }
        List<String> excluidos = new ArrayList<>();
        for(String s : excluidos){
            assertFalse(mapa.containsKey(s));
        }
    }
    protected Object getObjeto() {
        Produto p = new Produto();
        p.setNome("Tablet APX10");
        p.setPreco(500.00);
        p.setDescricao("10 pol, 16GB, 800x600");
```

```
        p.setCategoria("Eletrônicos");
        return p;
    }
    protected void getConteudoMapa(
     Map<String, Object> esperado){
        esperado.put("nome", "Tablet APX10");
        esperado.put("preco", 500.00);
        esperado.put("tipo", "Eletrônicos");
    }
    protected void getExcluidos(List<String> excluidos){
        excluidos.add("descricao");
        excluidos.add("categoria");
    }
}
```

A seguir, como mostrado na próxima listagem, pode-se extrair uma superclasse abstrata que contém a implementação do método de teste e a declaração abstrata dos métodos mais específicos. Dessa forma, a classe `TesteMapaProduto` apenas implementa esses métodos auxiliares, provendo informações para o teste que foi definido na superclasse. Como a chamada de método é a mesma, é possível fazer isso, pois o que muda é apenas o objeto passado para o método e o que é esperado para o retorno.

Criando uma classe modelo para a criação de testes do gerador de mapas:

```
public abstract class TesteMapaGenerico {

    @Test
    public void geracaoMapa() {
        Object o = getObjeto();
        Map<String,Object> mapa = GeradorMapa.gerarMapa(o);
        Map<String,Object> esperado = new HashMap<>();
        getConteudoMapa(esperado);
        for(String s : esperado.keySet()){
            assertEquals(esperado.get(s), mapa.get(s));
        }
        List<String> excluidos = new ArrayList<>();
        for(String s : excluidos){
```

```
            assertFalse(mapa.containsKey(s));
        }
    }
    protected abstract Object getObjeto();
    protected abstract void getConteudoMapa(Map<String,
                                            Object> esperado);
    protected abstract void getExcluidos(
                                        List<String> excluidos);
}
```

Por mais que a quantidade de linhas de código não mude muito, ter apenas métodos simples para implementar certamente aumenta a velocidade de criação dos testes de outros objetos. Esse efeito se potencializa quando existem outros métodos e outros cenários a serem testados.

5.4 GERANDO CLASSES COM CLASSMOCK

Uma das dificuldades de testar classes e componentes que utilizam reflexão é o fato de precisar a cada teste definir uma classe para ser passada como parâmetro. Algumas vezes, a diferença entre classes definidas em diferentes testes é apenas a assinatura de um método ou a presença de uma anotação. Infelizmente, não dá para reutilizar a definição de uma para a outra, por mais que elas sejam muito similares. Isso acaba tornando a definição de teste para esse tipo de classe mais trabalhosa e mais verbosa, pois as classes precisam ser definidas sempre por inteiro.

Visando simplificar esse tipo de teste, o framework **ClassMock** (FERNANDES; GUERRA; SILVEIRA, 2010) gera classes em tempo de execução para serem passadas como parâmetro. Esse framework pode ser baixado no endereço http://classmock.sf.net. Em vez de criar uma classe da forma usual, como foi visto nas seções anteriores, a partir da classe `ClassMock` são inseridas as

informações da classe que se deseja criar, como propriedades (que incluem atributo e métodos de acesso), interfaces, superclasse, métodos, anotações, entre outros. Dessa forma, é possível reutilizar uma definição inicial como base para gerar as variações necessárias para um método, ou mesmo criar métodos que insiram determinadas estruturas em uma mock class. O objetivo desta seção é apresentar a abordagem utilizada pelo ClassMock, sem necessariamente apresentar todos os métodos disponíveis em sua API. Para um conhecimento mais aprofundado de todas as funcionalidades dessa ferramenta, sugere-se acessar a documentação disponível no site do projeto.

Utilizando o ClassMock no gerador de mapas

Para exemplificar o uso do ClassMock, a listagem a seguir mostra os mesmos testes desenvolvidos para a classe `GeradorMapa` na seção *Variando estrutura da classe para teste*, criados agora utilizando esse framework. Observe que o método `criarMockClass()` é anotado com `@Before` e por isso será executado antes de todos os métodos de teste. Nele, é criada uma classe base para os testes chamada de `Bean` e que possui as propriedades `prop1` do tipo `String` e `prop2` inteira, da mesma forma que no exemplo original. Um atributo da classe de teste chamado `mockClass` do tipo `ClassMock` é utilizado para guardar essas informações.

Testes do gerador de mapas utilizando o ClassMock:

```
public class TesteGeradorMapaClassMock {

    private ClassMock mockClass;

    @Before public void criarMockClass(){
```

```
        mockClass = new ClassMock("Bean");
        mockClass.addProperty("prop1", String.class)
        .addProperty("prop2", int.class);
    }
    @Test public void mapaDeClasseSimples() {
        Object instance = ClassMockUtils.newInstance(mockClass);
        ClassMockUtils.set(instance,"prop1","teste");
        ClassMockUtils.set(instance,"prop2",25);
        Map<String,Object> mapa =
            GeradorMapa.gerarMapa(instance);
        assertEquals("teste",mapa.get("prop1"));
        assertEquals(25,mapa.get("prop2"));
    }
    @Test public void ignoraPropriedade() {
        mockClass.addAnnotation(
            "prop1", Ignorar.class, Location.GETTER);
        Object instance = ClassMockUtils.newInstance(mockClass);
        ClassMockUtils.set(instance,"prop1","teste");
        ClassMockUtils.set(instance,"prop2",25);
        Map<String,Object> mapa =
            GeradorMapa.gerarMapa(instance);
        assertFalse(mapa.containsKey("prop1"));
        assertEquals(25,mapa.get("prop2"));
    }
    @Test public void mudaNomePropriedade() {
        mockClass.addAnnotation("prop2", NomePropriedade.class,
            Location.GETTER, "propriedade2");
        Object instance = ClassMockUtils.newInstance(mockClass);
        ClassMockUtils.set(instance,"prop1","teste");
        ClassMockUtils.set(instance,"prop2",25);
        Map<String,Object> mapa =
            GeradorMapa.gerarMapa(instance);
        assertEquals("teste",mapa.get("prop1"));
        assertEquals(25,mapa.get("propriedade2"));
        assertFalse(mapa.containsKey("prop2"));
    }
}
```

Em seguida, em cada método de teste é criado um novo objeto a partir da mock class definida, no qual são configuradas propriedades. Como a classe é definida em tempo de execução, é preciso utilizar reflexão para manipular seus métodos e a

instanciar, porém o próprio ClassMock possui uma classe chamada `ClassMockUtils` com métodos auxiliares que podem ser utilizados para isso. No exemplo, são usados os métodos `newInstance()`, que retornam uma instância de uma classe definida com `ClassMock`, e `set()`, que insere um valor em uma propriedade. Outros métodos dela são `get()`, que recupera informações, e `invoke()`, que executa métodos. Caso o método testado não receba uma instância da classe, mas uma instância de `Class` representando a classe, ela pode ser recuperada a partir do método `createClass()` da própria `ClassMock`.

Observe no exemplo de classe de teste mostrada que os métodos de teste `ignoraPropriedade()` e `mudaNomePropriedadea()` adicionam uma anotação na mock class antes de criarem uma instância dela. Dessa forma, é possível aproveitar a estrutura inicial da classe definida no método anotado com `@Before` para depois fazer as modificações necessárias nos métodos de teste. Peço ao leitor para voltar na seção *Variando estrutura da classe para teste* e ver como dessa forma é possível definir os testes de forma mais objetiva e menos verbosa.

ClassMock + JMock = testes de proxies dinâmicos

A ideia principal do ClassMock é criar a estrutura de mock classes para serem utilizadas para teste. Quando são adicionadas propriedades, o ClassMock gera não só o atributo, mas também a implementação dos dois métodos de acesso. Porém, não é o seu objetivo gerar o comportamento de métodos, mas somente a sua estrutura. Quando é necessário simular algum comportamento, é preciso utilizar algum framework de mock objects, como o JMock, para criar o mock object da mock class definida. Esse tipo de

prática, como foi visto na seção *Teste de proxies dinâmicos*, é o caminho que deve ser utilizado para a criação de teste para proxies dinâmicos.

A listagem a seguir mostra como essa abordagem seria utilizada combinando o uso dos frameworks ClassMock e JMock para o teste da classe `CacheProxy`. Inicialmente, a classe `ClassMock` é utilizada para criar uma interface chamada `CacheMe`. O segundo parâmetro `true` indica que uma interface será criada. Em seguida, é adicionado um método abstrato chamado `comCache()` que retorna inteiro, indicado pelo primeiro parâmetro, e recebe um inteiro, indicado pelo último parâmetro. Por fim, é adicionada a anotação `@Cache` no método `comCache()`. Todos esses parâmetros serão utilizados para a geração dinâmica da interface para ser utilizada pelo teste, que é feita no momento em que o método `createClass()` é chamado.

Testes do gerador de mapas utilizando o ClassMock:

```
public class TesteCacheProxyClassMock {

    @Rule public JUnitRuleMockery ctx = new JUnitRuleMockery();

    @Test
    public void cacheSimples() throws Throwable {
        ClassMock cm = new ClassMock("CacheMe", true);
        cm.addAbstractMethod(int.class, "comCache", int.class)
            .addMethodAnnotation("comCache", Cache.class);
        Class<?> interf = cm.createClass();
        final Object mock = ctx.mock(interf);
        Object proxy = CacheProxy.criar(mock);

        ctx.checking(new Expectations() {{
            ClassMockUtils.invoke(one(mock),"comCache",1);
            will(returnValue(10));
        }});
```

```
        assertEquals(10, ClassMockUtils.invoke(proxy,
                                               "comCache",1));
        assertEquals(10, ClassMockUtils.invoke(proxy,
                                               "comCache",1));
    }
}
```

O próximo passo depois da criação da mock class é a criação do respectivo mock object para definir as expectativas e o comportamento do objeto que será encapsulado pelo proxy dinâmico. O método `mock()` é chamado recebendo a interface criada dinamicamente, e retorna uma instância dela controlada pelo JMock. Em seguida, o proxy é criado a partir do encapsulamento do mock object. Depois disso, a lógica do teste é a mesma, porém, como a interface foi criada de forma dinâmica, é preciso utilizar o método `invoke()` da classe `ClassMockUtils` para fazer a invocação dos métodos. Note que esse método é utilizado nas asserções e inclusive para definir as expectativas do mock object.

5.5 CONSIDERAÇÕES FINAIS

Este capítulo apresentou as técnicas básicas para o teste de classes que utilizam reflexão e metadados para seu processamento. Inicialmente foi mostrada a criação de classes para simular diferentes estruturas que serão consumidas pelos testes e, em seguida, como as invocações de métodos poderiam ser verificadas. Baseado nos exemplos apresentados, foi introduzido o conceito de **mock class** que para o teste desse tipo de classe é diferente e complementar aos **mock objects**. Também foi abordado o teste de proxies dinâmicos, que combinam os conceitos de mock class e mock object para a definição do teste. Por fim, vimos o teste da

configuração de metadados, que deve ser feito pelas aplicações que utilizam esses componentes para verificar se os metadados estão configurados corretamente para que o comportamento seja o esperado.

Em geral, o teste desse tipo de componente é difícil de ser realizado pois depende de fatores dinâmicos nas classes, que muitas vezes são difíceis de serem previstos e simulados. Porém, essa dificuldade é um obstáculo que não pode impedir a criação desses testes, pois, por ser reutilizado em diversas partes do sistema, é preciso que ele possua uma boa qualidade. Frameworks como o JMock ou o ClassMock ajudam a diminuir algumas dificuldades inerentes a esse tipo de teste, porém, dependendo da funcionalidade da classe, novas dificuldades podem surgir, e podem demandar até a criação de outras classes auxiliares para viabilizar a criação do teste.

Capítulo 6

PRÁTICAS NO USO DE ANOTAÇÕES

"Antes de o software ser reutilizável, ele precisa ser utilizável." - Ralph Johnson

Os exemplos apresentados nas seções anteriores utilizaram metadados bem simples, que precisaram de anotações sem atributos ou com apenas um atributo. Essa situação nem sempre ocorre quando estamos desenvolvendo um componente baseado em metadados mais complexo. Muitas vezes é necessário ter informações mais elaboradas associadas a um elemento da classe, como, por exemplo, uma regra de segurança. Em outros casos, é preciso repetir configurações para vários elementos da mesma classe, ou do mesmo pacote, como para dizer se uma operação deve ser ou não transacional.

Ao se criar um conjunto de anotações para serem lidas por um componente, o que será chamado de esquema de anotações, deve-se pensar em uma solução que será fácil de ser utilizada pelo usuário. Quando muitas anotações precisam ser adicionadas em um mesmo elemento, ou até quando uma única possui diversos atributos, isso torna o código poluído, causando maus cheiros no código, conhecidos como *Annotation Drowning* (afogamento por

anotações) ou *Crowded Party* (multidão) (SILVEIRA; FERNANDES; CORREIA; GUERRA, 2010). Outro problema que pode ocorrer é a repetição da mesma configuração em diversos elementos da mesma classe, ou de classes diferentes, o que é conhecido como *Popular Configuration* (configuração popular). A próxima listagem mostra como uma classe sobrecarregada de anotações pode ficar.

Classe sobrecarregada de anotações:

```
@NamedQuery(name = "getCrowdedClass",
            query = "from Classes where n > :veryhigh")
@Entity
@Table(name = "LOTADA")
@Inheritance(strategy = SINGLE_TABLE)
@DiscriminatorColumn(name = "_type_",discriminatorType = STRING)
@DiscriminatorValue("sobrecarregada")
@Cache(usage = NONSTRICT_READ_WRITE)
public class Sobrecarregada {

   @Id
   @GeneratedValue(strategy=SEQUENCE, generator = "hibseq")
   @SequenceGenerator(name =”meuGerador",
                          sequenceName = "common_seq")
   private Integer id;
}
```

Ao realizar a modelagem dos metadados que serão consumidos pelo componente, todas essas questões devem ser levadas em consideração, para que o componente não traga problemas à aplicação que o está utilizando. Uma quantidade muito grande de anotações pode piorar a legibilidade do código dificultando sua compreensão. A repetição de uma mesma configuração em vários elementos, por outro lado, dificulta a sua modificação, que precisará ser feita em diversos locais do código. Nesse contexto, a utilização de boas práticas para definição e consumo de metadados, pode simplificar essa tarefa para quem estiver

utilizando o componente.

Este capítulo irá apresentar várias práticas que podem ser utilizadas para a definição de anotações, e como pode ser criado um código para consumir os metadados definidos dessa forma. Quando for possível, será mostrado um método mais geral, que facilita a implementação dessa prática em qualquer componente. Essas práticas foram extraídas de diversos frameworks existentes e já foram utilizadas com sucesso em vários contextos diferentes. Em algumas seções serão apresentados exemplos de APIs e frameworks existentes que implementam a prática apresentada.

6.1 MAIS DE UMA ANOTAÇÃO DO MESMO TIPO

Uma limitação conhecida das anotações em Java antes da versão 8 é que só é possível adicionar uma anotação de cada tipo a cada elemento (a seção *Múltiplas anotações* mostra como o Java 8 lidou com essa questão). Porém, em algumas situações, é necessário poder adicionar várias configurações do mesmo tipo ao mesmo elemento. Quando a configuração é simples, ou seja, a anotação precisa de apenas um atributo, um recurso que pode ser feito é usar um array. Imagine uma anotação que defina diversos arquivos de configuração que precisam ser lidos para a criação de uma determinada classe. A próxima listagem mostra um exemplo de como seria sua configuração.

Utilizando um array para diversas configurações na anotação:

```
@Arquivos({"config.xml", "conf.txt", "alteracoes.log"})
public class ParaConfigurar{
```

```
    ...
}
```

O uso do array de informações no caso de haver apenas um atributo não causa muitos problemas, porém, imagine se outras informações forem necessárias, como o tipo do arquivo e o diretório onde ele se encontra. Nesta situação, se a mesma solução fosse utilizada, cada atributo da anotação teria um array de valores, como ilustrado na próxima listagem. Observe que é a posição do valor no array que define quais são os outros valores correspondentes. Como as informações relativas a um único arquivo não ficam juntas, a definição da anotação fica mais difícil de ser lida.

O uso de arrays para vários atributos:

```
@Arquivos(
    arquivos = {"config.xml", "conf.txt", "alteracoes.log"},
    tipos = {"XML", "TXT", "TXT"},
    diretorios = {"C:\\conf", "C:\\conf", "C:\\log"}
)
public class ParaConfigurar{
    ...
}
```

Anotação vetorial

Uma solução que pode ser adotada nesse caso é definir uma anotação para adicionar um metadado referente a um único arquivo de configuração, e uma outra que possua uma array da anotação do primeiro tipo. Dessa forma, quando for necessário adicionar mais de um arquivo de configuração, a segunda anotação é utilizada para agrupar uma lista de anotações do primeiro tipo. Essa solução será chamada de **Anotação Vetorial**. A listagem a seguir mostra como ficaria o caso dos arquivos de configuração

com essa abordagem. Mesmo ficando mais verboso, devido ao fato de o nome do atributo precisar ser repetido em cada anotação, o código fica mais claro porque as informações que são relacionadas são colocadas juntas.

Uma anotação com um atributo que é um array de anotações:

```
@Arquivos({
    @Arquivo(value="config.xml", tipo="XML",dir="C:\\conf"),
    @Arquivo(value="conf.txt", tipo="TXT",dir="C:\\conf"),
    @Arquivo(value="alteracoes.log", tipo="TXT",dir="C:\\log")
})
public class ParaConfigurar{
    ...
}
```

O uso da anotação vetorial também favorece cenários onde existem atributos opcionais ou onde o valor default vai ser utilizado. Como a definição de cada anotação no array é independente, o fato de um atributo ser omitido em alguma delas não vai influenciar as outras. Isso não é verdade no caso dos atributos com arrays de valores que foi mostrado anteriormente, pois o que liga um valor ao outro é justamente sua posição, o que inviabiliza a omissão de alguma informação que não for necessária.

A listagem a seguir apresenta um método que pode ser utilizado para a recuperação dos arquivos utilizando a anotação `@Arquivos` para agrupar anotações do tipo `Arquivo`. Inicialmente, é criada uma lista para armazenar as anotações que forem encontradas durante a busca. A primeira verificação realizada é se existe a anotação `@Arquivo`, a qual é adicionada na lista se for encontrada. Em seguida, é verificada se a anotação `@Arquivos` está presente e, em caso positivo, todos as anotações

`@Arquivo` presentes no seu atributo `value` são adicionadas na lista para serem retornadas.

Recuperação dos arquivos das anotações:

```
public static List<Arquivo> getArquivos(Class<?> c){
    List<Arquivo> lista = new ArrayList<>();
    if(c.isAnnotationPresent(Arquivo.class)){
        lista.add(c.getAnnotation(Arquivo.class));
    }
    if(c.isAnnotationPresent(Arquivos.class)){
        Arquivos arqs = c.getAnnotation(Arquivos.class);
        for(Arquivo a : arqs.value())
            lista.add(a);
    }
    return lista;
}
```

Na API de mapeamento objeto-relacional JPA, é possível definir uma consulta nomeada, ou seja, uma *named query*, através da anotação `@NamedQuery`. Quando a classe quer definir mais de uma consulta, ela utiliza a solução que foi apresentada nesta seção. A anotação `@NamedQueries` recebe em seu atributo `value` um array da `@NamedQuery`.

Solução reutilizável

Essa solução que foi apresentada é específica para as anotações `@Arquivo` e `@Arquivos`, e caso quiséssemos utilizar uma solução similar para outras, seria necessário criar um outro método para sua recuperação. Para elaborar a solução, devemos determinar que informação precisamos ter na anotação principal, para poder recuperar a sua anotação vetorial. Nesse caso, se soubermos qual a classe da anotação vetorial correspondente, poderíamos procurá-la e recuperar o array com as anotações originais.

Para poder configurar esse metadado, vamos criar uma anotação para ser adicionada na própria anotação principal, que irá configurar qual a classe corresponde à anotação vetorial. Sua definição, chamada `@Vetorial`, está apresentada na próxima listagem. Observe que, pela configuração de `@Target`, ela só pode ser adicionada em `ANNOTATION_TYPE`. Adicionalmente, o atributo `value`, pelo seu tipo genérico, só pode receber classes que estendam `Annotation`, ou seja, anotações.

Recuperação dos arquivos das anotações:

```
@Target(ElementType.ANNOTATION_TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Vetorial {
    Class<? extends Annotation> value();
}
```

Na próxima listagem é apresentado um método mais geral, que possibilita a recuperação de uma lista de anotações, fazendo também a busca em sua respectiva anotação vetorial. Observe que, como na solução mais específica, é criada uma lista de anotações que será utilizada para adicionar as anotações encontradas e ser retornada pelo método. Inicialmente a solução verifica se a própria anotação está presente, e, em caso positivo, adiciona-a na lista. Em seguida, o método verifica se a anotação está anotada com `@Vetorial` e, nesse caso, chama o método `recuperaVetorial()`.

Recuperação dos arquivos das anotações:

```
public static List<Annotation> getAnnotations(
 AnnotatedElement ae, Class<? extends Annotation> anot) throws
 Exception{
    List<Annotation> lista = new ArrayList<>();
    if(ae.isAnnotationPresent(anot)){
        lista.add(ae.getAnnotation(anot));
```

```
    }
    if(anot.isAnnotationPresent(Vetorial.class)){
        recuperaVetorial(ae, anot, lista);
    }
    return lista;
}
```

A listagem a seguir apresenta o método que recupera a anotação `@Vetorial` e, a partir dessa informação, busca a lista de anotações na anotação vetorial correspondente. A primeira ação desse método é recuperar de `@Vetorial` a classe configurada para a anotação vetorial, que será armazenada na variável `vetAnot`. Em seguida, verifica-se se essa anotação está presente no elemento anotado e, em caso positivo, os valores precisarão ser recuperados através de reflexão, pois não se conhece em tempo de compilação o tipo dessa anotação. Sendo assim, é recuperado da classe da anotação vetorial o método correspondente ao atributo `value`, assumindo que ele será um array de anotações. Em seguida, esse array é percorrido e seus elementos são inseridos na lista.

Recuperação dos arquivos das anotações:

```
protected static void recuperaVetorial(AnnotatedElement ae,
        Class<? extends Annotation> anot,List<Annotation> lista)
        throws Exception {
    Vetorial vetorial = anot.getAnnotation(Vetorial.class);
    Class<? extends Annotation> vetAnot = vetorial.value();
    if(ae.isAnnotationPresent(vetAnot)){
        Annotation an = ae.getAnnotation(vetAnot);
        Method value = an.annotationType().getMethod("value");
        Annotation[] array = (Annotation[]) value.invoke(an);
        for(Annotation a : array)
            lista.add(a);
    }
}
```

Para ilustrar como a anotação `@Vetorial` seria utilizada na prática, a próxima listagem apresenta como ela seria aplicada no

exemplo da anotação `@Arquivo`. Observe que a `@Vetorial` é adicionada, indicando no atributo `value` a classe que possui a sua anotação vetorial, no caso `@Arquivos`. Sendo assim, o método `getAnnotations()` poderia ser utilizado para recuperar uma lista de anotações `@Arquivo`.

Recuperação dos arquivos das anotações:

```
@Retention(RetentionPolicy.RUNTIME)
@Vetorial(Arquivos.class)
public @interface Arquivo {
    String value();
    String tipo();
    String dir();
}
```

6.2 REDUZINDO A QUANTIDADE DE CONFIGURAÇÕES

"O que diferencia esse framework de todos os outros é a preferência pela convenção sobre a configuração, tornando as aplicações mais fáceis de desenvolver e entender." - Sam Ruby, sobre Ruby on Rails

Um objetivo que se precisa ter em mente ao se modelar um esquema de anotações é possibilitar ao desenvolvedor definir os metadados com o mínimo de configurações possível. No primeiro contato mais profundo com reflexão, no capítulo *Java Reflection API*, foram utilizadas convenções de código para obter mais informações sobre os elementos da classe. Porém, no capítulo *Metadados e anotações* em seguida, vimos que isso nem sempre era suficiente, e que em muitos casos é necessário configurar metadados adicionais para que o processamento do componente

seja possível. Em um componente real, é preciso encontrar um equilíbrio entre essas duas estratégias, para permitir a configuração de exceções sem exigir um número excessivo de configurações.

Convenções de código

Para exemplificar, vamos considerar o caso do gerador de mapas a partir das propriedades de JavaBeans apresentado na seção *O primeiro contato com a API Reflection*. Na versão inicial, quando apenas reflexão foi utilizada, os nomes das propriedades eram usados obrigatoriamente como chaves do mapa. A segunda versão, que introduziu o uso de anotações, permitiu a flexibilidade de utilizar outra chave no armazenamento da propriedade do mapa. Porém, é importante observar que, no caso mais comum, em que se usa o nome da propriedade como chave, não é necessária nenhuma configuração. Uma implementação ruim desse mesmo componente, poderia exigir que todas as propriedades fossem utilizadas.

Esse princípio é conhecido como *Convention over Configuration* (convenção sobre configuração) (CHEN, 2006), ou apenas **CoC** para os mais íntimos. Utilizando esse princípio, deve-se tentar utilizar a convenção sempre que possível e usar configurações somente quando a convenção falhar. Isso, além de permitir uma curva de aprendizado mais suave para desenvolvedores não familiarizados com o componente ou framework, também torna o código mais uniforme. O framework Ruby on Rails para desenvolvimento web é conhecido por utilizar e ser um grande defensor desse princípio.

Um cenário de utilização dessa prática é quando existe uma

possibilidade que se aplica à maioria dos casos, porém podem existir algumas exceções daquela regra. Dessa forma, o componente pode tratar todos os elementos sem configuração da forma padrão, e exigir a adição de um metadado adicional para as exceções. Na API JPA para persistência de dados, uma classe anotada com `@Entity` é considerada uma classe persistente, e como caso geral todos os seus atributos serão persistidos. As exceções, que são atributos que não devem ser persistidos, são anotadas com `@Transient`.

Outro cenário é quando parte das informações inerentes ao elemento podem ser utilizadas para inferir alguma propriedade do metadado. O nome de um método de acesso, por exemplo, pode ser utilizado para inferir informações como a coluna em que deverá ser persistido ou o nome do elemento que o irá representar em um documento XML. Os tipos envolvidos com o elemento, como a classe de um atributo ou o retorno de um método, também podem ser utilizados para determinar conversões ou validações a serem feitas. Muitas vezes, podem-se obter informações sobre esse tipo, até mesmo acessando suas anotações, para se determinar alguma informação. Em vários casos, essa prática pode não impedir que um metadado precise ser adicionado, mas pode diminuir a quantidade de propriedades que precisam ser configuradas.

Para se utilizar dessa prática, deve-se verificar se dentre os metadados que foram definidos como necessários para o componente existe algum que será a configuração para a grande maioria dos casos, algo como 80% ou 90% dos elementos. Dessa forma, quando o componente for buscar o metadado, ele deve considerar aquela configuração caso nada seja encontrado. Um

segundo passo é verificar dentre os metadados se existe alguma informação que poderia ser inferida diretamente do elemento sem informações adicionais. Para valer a pena, essa convenção também deve funcionar na maioria dos casos. Dessa forma, quando o metadado for solicitado, caso não exista uma configuração explícita, deve-se extrair a informação dos dados do elemento do código.

Configurações gerais

Existem alguns cenários em que fica difícil de definir uma configuração default que vai cobrir a maioria dos casos em todas as aplicações que forem utilizar o componente. Nessas situações, o comportamento padrão pode depender da aplicação ou mesmo se aplicar a um determinado conjunto de classes dela. Por exemplo, dentro de uma classe com vários métodos, pode ser que exista um padrão de configuração que precisará ser configurado para a maioria deles, por mais que esse padrão não se aplique a outras classes.

Uma forma de permitir a diminuição de configurações nesse caso é permitindo que possam ser feitas configurações mais gerais, que se apliquem a um grupo de elementos. Por exemplo, a configuração que precisaria ser feita individualmente para cada método poderia ser feita na classe, sendo essa a configuração default para todos os seus métodos. De forma análoga, a anotação poderia ser adicionada ao pacote, sendo considerada o valor default para todas as classes no pacote e todos os métodos em suas classes. Caso não se deseje o valor default configurado de forma mais geral, basta o elemento soprepor aquela configuração.

Para termos ideia de como isso pode poupar configurações, imagine uma classe que possua oito métodos, sendo que seis possuem a mesma configuração e outros dois tenham configurações distintas. Definindo uma configuração mais geral para a classe, seriam necessárias apenas três anotações, sendo uma para a classe e duas para os outros métodos, e não oito anotações, uma para cada método. Isso, sendo expandido para um pacote, faria a quantidade de configurações que poderiam ser eliminadas aumentar ainda mais.

A listagem a seguir mostra um método que faz a busca de anotações considerando que ela pode estar definida em um elemento mais geral no código. O método recebe como parâmetro o elemento de código e a classe da anotação que deve procurar. Obviamente, o primeiro local onde é feita a busca é no próprio elemento recebido como parâmetro, mas caso a anotação não seja encontrada, a busca é realizada no elemento de código mais geral onde o elemento recebido está contido.

Recuperação de anotações considerando definições em elementos mais gerais:

```
public static Annotation getAnnotation(
 Class<? extends Annotation> c, AnnotatedElement ae){
    Annotation an = ae.getAnnotation(c);
    if(an == null){
        if(ae instanceof Method){
            return
             getAnnotation(c, ((Method)ae).getDeclaringClass());
        }else if(ae instanceof Field){
            return
             getAnnotation(c, ((Field)ae).getDeclaringClass());
        }else if(ae instanceof Class){
            return getAnnotation(c, ((Class)ae).getPackage());
        }
    }
```

```
    return an;
}
```

Em outras palavras, se o elemento for um método ou um atributo, a busca será feita na respectiva classe, e se o elemento for uma classe, a busca será feita no pacote. Observe que é feita uma chamada recursiva da própria função, passando como parâmetro o elemento mais geral. Se o elemento for um pacote e a anotação não for encontrada, pela implementação apresentada, a busca irá parar e será retornado o valor nulo. Um exercício a ser feito pelos leitores seria continuar a busca no pacote mais geral, até se chegar à raiz.

Quando esse tipo de estratégia de definição de anotações é empregado, é preciso considerar as anotações cujas simples presenças adicionam semântica ao elemento. Sendo assim, se uma anotação desse tipo for adicionada em um elemento mais geral, como não existe uma configuração que defina a negativa daquela anotação, não existirá forma de "remover" a anotação. Como forma de abordar esse problema, a listagem a seguir mostra a anotação `@CancelaAnotacoes`, que cancela as configurações realizadas em elementos mais gerais que o elemento que a contém. Observe que o atributo `value` recebe a lista de classes das anotações para as quais isso deve ocorrer.

Anotação para ignorar definições feitas em elementos mais gerais:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface CancelarAnotacoes {
    Class<? extends Annotation>[] value();
}
```

A listagem a seguir mostra uma nova versão do método

`getAnnotation()` considerando a possível presença da anotação `@CancelaAnotacoes`. A única modificação feita no método principal é a adição da chamada ao método `cancelaParentes()` no condicional que verifica se deve ser feita uma chamada recursiva ao método passando como parâmetro o elemento mais geral. Esse método introduzido verifica se a anotação `@CancelaAnotacoes` está presente no elemento, e em caso positivo verifica também se a classe da anotação procurada está na lista de anotações para serem ignoradas.

Nova versão da recuperação em elementos gerais considerando a presença de @CancelaAnotacoes:

```
public static Annotation getAnnotation(
 Class<? extends Annotation> c, AnnotatedElement ae){
    Annotation an = ae.getAnnotation(c);
    if(an == null && !cancelaParentes(c, ae)){
        if(ae instanceof Method ){
            return
             getAnnotation(c, ((Method)ae).getDeclaringClass());
        }else if(ae instanceof Field){
            return
             getAnnotation(c, ((Field)ae).getDeclaringClass());
        }else if(ae instanceof Class){
            return getAnnotation(c, ((Class)ae).getPackage());
        }
    }
    return an;
}
public static boolean cancelaParentes(
 Class<? extends Annotation> c, AnnotatedElement ae){
    if(ae.isAnnotationPresent(CancelarAnotacoes.class)){
        CancelarAnotacoes ca =
            ae.getAnnotation(CancelarAnotacoes.class);
        for(Class<? extends Annotation> cancelado : ca.value()){
            if(c == cancelado)
                return true;
        }
    }
    return false;
```

```
}
```

6.3 EXPRESSÕES EM ANOTAÇÕES

"É tão difícil se expressar. Eu entendo isso." - Jonathan Safran Foer, Everything Is Illuminated

Existem diversos domínios de componentes e frameworks em que se tem a necessidade de definir regras como parte de metadados. Exemplos de domínios que podem precisar disso são segurança (regras para bloquear ou permitir o acesso), validação (regras que definem se uma informação é válida ou não) e tratamento de eventos (regras para definir quem irá tratar o evento). Nesse caso, a estruturação de regras em anotações é complexa e possui uma série de restrições, principalmente quando você precisa combinar essas regras com operadores lógicos. Expressões matemáticas e combinação de valores também se encaixam nessa categoria de metadados que são difíceis de expressar de forma estruturada.

Representando regras

Para ilustrar esse problema, imagine que se deseja expressar regras de validação de um objeto a partir de anotações. Uma regra é composta por uma propriedade, um operador de comparação (como maior, menor ou igual) e um valor com o qual a propriedade será comparada. Essas regras podem ser combinadas utilizando operadores " `and` " e " `or` " para formar a expressão de validação. A listagem a seguir mostra como seria o uso dessas anotações de regra. No caso, a classe `CidadaoIndependente` precisa possuir o atributo `idade` maior que dezoito ou esse

atributo maior que dezesseis e `emancipado` igual a `true`. Veja que foi utilizada a anotação vetorial `@Regras` para permitir que várias regras sejam agrupadas.

Criando uma estrutura de anotações para definição de regras:

```
@Regras({
    @Regra(prop="idade",valor="18",operador=">",conector="or"),
    @Regra(prop="idade",valor="16",operador=">",conector="and"),
    @Regra(prop="emancipado",valor="true", operador="==")
})
public class CidadaoIndependente {
    private String nome;
    private int idade;
    private boolean emancipado;
    //getters e setters omitidos
}
```

Observando esse código fica muito claro que as anotações não foram feitas para representar esse tipo de informação. A expressão estaria correta caso o operador " `and` " tivesse prioridade, mas tente imaginar como é possível a representação de expressões com parênteses. Além disso, fica claro que o código é sobrecarregado de anotações e de baixa legibilidade. Um fator que contribui para a dificuldade de representação desse tipo de metadado com anotações é o fato de não existir herança, e consequentemente polimorfismo. Dessa forma, não é possível definir que `@Regras` recebe um conjunto de `@Regra` e poder passar outras anotações derivadas. Assim, fica-se preso à definição original, sem possibilidade de extensão.

Uma alternativa à estruturação das informações das anotações é disponibilizar na anotação apenas um campo texto para que o desenvolvedor entre com uma expressão. Veja na listagem a seguir

como essa abordagem gera um código mais limpo e mais legível. Porém, isso tem um preço, pois o desenvolvedor precisa aprender essa linguagem utilizada para expressar a regra dentro da anotação. Adicionalmente, o componente que faz a validação precisará ler, interpretar e executar essa expressão com os dados do objeto, o que não é algo trivial de ser implementado.

Utilização de expressão para definir uma regra:

```
@Regra("idade > 18 || (idade > 16 && emancipado)")
public class CidadaoIndependente {
    private String nome;
    private int idade;
    private boolean emancipado;
    //getters e setters omitidos
}
```

Utilizando a API para Expression Language

Na seção anterior vimos que o uso de expressões dentro de anotações gera um código mais limpo e legível, além de ser uma solução mais flexível que a estruturação das regras nos atributos das anotações. A dificuldade estaria em implementar o interpretador de expressões, porém já existe uma API em Java que faz a interpretação e possui diversas implementações. Trata-se da API para avaliar expressões em páginas web a partir do JSP 2.0, linguagem conhecida como **Expression Language**, ou simplesmente **EL**. Poucos desenvolvedores sabem que essa API pode ser utilizada de forma independente para avaliar outros tipos de expressão. Esta seção vai mostrar brevemente como utilizar essa API para avaliar expressões. Na implementação dos exemplos foram utilizados os arquivos `el-api.jar` e `jasper-el.jar` que podem ser encontrados dentre as bibliotecas do Tomcat.

EL EM MENOS DE 1 MINUTO

Não está no escopo deste livro ensinar a utilizar a EL, mas vai aqui um resumo rápido de como ela funciona. É possível adicionar no escopo da EL variáveis e funções, além de utilizar operadores e valores literais nas expressões. Toda expressão deve estar na forma `${expressao}`. Variáveis que são JavaBeans podem ter suas propriedades acessadas utilizando ponto `bean.propriedade` ou colchetes `bean["propriedade"]`. De forma similar, os mapas também podem ter seus valores acessados da mesma forma, sendo que em vez do nome da propriedade seria utilizada a chave do mapa. Valores de listas e arrays pode ser acessados a partir de seu índice entre colchetes, como `lista[1]` ou `lista["1"]`.

O primeiro passo para se utilizar a API de EL é criar uma classe para armazenar o contexto de avaliação de expressões estendendo a classe `ELContext`, como está apresentado na próxima listagem. Esse contexto precisa ter as funções que estarão disponíveis para uso na expressão, representada pela classe `FunctionMapper`, e as variáveis que poderão ser utilizadas, representada pela classe `VariableMapper`. Observe que a implementação simplesmente recebe as instâncias dessas classes e sobrescreve os métodos da superclasse responsáveis por disponibilizar essas informações.

Definição para um contexto de avaliação de expressões:

```
public class MeuELContext extends ELContext{
```

```
    private FunctionMapper functionMapper;
    private VariableMapper variableMapper;
    private CompositeELResolver elResolver;

    public MyELContext(FunctionMapper functionMapper,
            VariableMapper variableMapper,
            ELResolver ...resolvers){
        this.functionMapper = functionMapper;
        this.variableMapper = variableMapper;
        elResolver = new CompositeELResolver();
        for(ELResolver resolver : resolvers){
            elResolver.add(resolver);
        }
    }
    public ELResolver getELResolver() {
        return elResolver;
    }
    public FunctionMapper getFunctionMapper() {
        return functionMapper;
    }
    public VariableMapper getVariableMapper() {
        return variableMapper;
    }
}
```

A classe `ELResolver` é responsável por interpretar as variáveis e resolver o seu valor. Toda a sintaxe de interpretação de valores de propriedades de JavaBeans, mapas, arrays e listas são interpretadas por implementações dessa interface. Observe que o construtor da classe criada recebe uma lista de `ElResolver` e utiliza a classe `CompositeELResolver` para que todos sejam utilizados.

A listagem a seguir ilustra como mapear as variáveis que podem ser utilizadas na expressão. A ideia é que todo valor que precisar estar disponível no momento da avaliação seja adicionado nessa variável. No caso do método `mapearVariaveis()` apresentado, ele recebe um mapa e adiciona todas as entradas não nulas em uma instância de `VariableMapper` . A classe

`ValueExpressionLiteral` é utilizada para inserir um valor no `VariableMapper`, e deve receber em seu construtor o valor e a respectiva classe. Em seguida, essa instância é inserida junto com o nome com que a variável deve ser referenciada na expressão, no caso a própria chave em que ela estava no mapa.

Fazendo o mapeamento de variáveis para a expressão:

```
public static VariableMapper mapearVariaveis(
                Map<String, Object> vars) {
    VariableMapper mapper = new VariableMapperImpl();
    for (String attributeName : vars.keySet()) {
        if (vars.get(attributeName) != null){
           Class<?> clazz = vars.get(attributeName).getClass();
            ValueExpressionLiteral expression =
                new ValueExpressionLiteral(
                    vars.get(attributeName), clazz);
            mapper.setVariable(attributeName, expression);
        }
    }
    return mapper;
}
```

Depois de mapear as variáveis, é preciso fazer o mapeamento das funções. A próxima listagem mostra um código que cria uma implementação de `FunctionMapper` adicionando todos os métodos estáticos de uma classe recebida como parâmetro. Observe que o método `addFunction()` é utilizado para mapear o método. O primeiro parâmetro, para o qual é passado uma `String` vazia, representa um prefixo que poderia ser utilizado na invocação do método. É importante ressaltar que apesar de esse exemplo adicionar métodos da mesma classe, nada impede métodos de diferentes classes de serem adicionados na implementação de `FunctionMapper`.

Mapeando métodos estáticos de uma classe para funções

para serem utilizadas na expressão:

```
public static FunctionMapper mapearFuncoes(Class<?> clazz) {
    FunctionMapperImpl mapper = new FunctionMapperImpl();
    for (Method m : clazz.getMethods()) {
          if(Modifier.isStatic(m.getModifiers()))
            mapper.addFunction("", m.getName(), m);
    }
    return mapper;
}
```

Para finalizar as funções utilitárias para o uso da Expression Language, a listagem apresentada a seguir mostra o método `criarContexto()`, que faz a criação do `EvaluationContext` a partir dos métodos e das classes mostradas anteriormente. Ele recebe como parâmetro uma classe de onde serão tirados os métodos estáticos e um mapa de objetos de onde serão tiradas as variáveis. Observe que são adicionadas as implementações de `ELResolver` para interpretar arrays, listas, mapas e JavaBeans (a ordem é importante).

Criação do contexto de avaliação de expressões:

```
public static EvaluationContext criarContexto(
 Class<?> functionClass, Map<String, Object> attributeMap) {
    VariableMapper vMapper = mapearVariaveis(attributeMap);
    FunctionMapper fMapper = mapearFuncoes(functionClass);
    MeuELContext context = new MeuELContext(fMapper, vMapper,
            new ArrayELResolver(), new ListELResolver(),
            new MapELResolver(), new BeanELResolver());
    return new EvaluationContext(context, fMapper, vMapper);
}
```

Finalmente, para ilustrar como uma expressão em EL pode ser executada, a próxima listagem mostra o exemplo de avaliação do valor de uma `String`. No mapa de variáveis são adicionados `a` com valor igual a 23 e `b` com valor igual a 10. Para se obter as funções, foi passada como parâmetro a própria classe do exemplo,

que possui o método estático `duplicar()`. A variável `expr` armazena uma expressão que utiliza não somente as duas variáveis, mas também o método definido na classe. O `EvaluationContext` é obtido através do método `criarContexto()` e utilizado para criar a expressão.

Exemplo de avaliação de expressão em EL:

```
public class Principal {

    public static void main(String[] args) {
        Map<String, Object> mapa = new HashMap<>();
        mapa.put("a", 23);
        mapa.put("b",10);

        String expr = "${a+duplicar(b)}";
        EvaluationContext ec =
            criarContexto(Principal.class, mapa);
        ValueExpression result = new ExpressionFactoryImpl()
            .createValueExpression(ec, expr, int.class);
        System.out.println(result.getValue(ec));
    }
    public static int duplicar(int i){
        return 2*i;
    }
}
```

A expressão é criada a partir do método `createValueExpression()` e deve receber como parâmetro o `EvaluationContext`, a `String` com a expressão a ser avaliada e o tipo do retorno esperado, que no caso é inteiro. O resultado da expressão é obtido através da chamada do método `getValue()`. Essa é a receita de bolo para utilizar Expression Language e avaliar as expressões! Basta agora utilizar isso para avaliar as expressões definidas como atributos de anotações.

Implementando o exemplo da regra de validação

A partir das funções auxiliares apresentadas nas listagens anteriores, não é difícil implementar uma classe que faz a validação do objeto através da regra contida na anotação `@Regra`. A listagem a seguir apresenta a classe `Validador` que possui o método `valido()`, que retorna um valor booleano dizendo se o objeto é valido ou não. O primeiro passo é verificar a existência da anotação `@Regra` e recuperar a expressão de validação, que para se encaixar nos padrões da EL precisa estar entre `"${"` e `"}"`.

Classe que executa a regra de validação contida na anotação:

```
public class Validador {

    public boolean valido(Object obj){
        Class<?> c = obj.getClass();
        if(c.isAnnotationPresent(Regra.class)){
            Regra r = c.getAnnotation(Regra.class);
            String expr = "${"+r.value()+"}";
            Map<String,Object> atrib =
                GeradorMapa.gerarMapa(obj);
            EvaluationContext ec =
                criarContexto(Validador.class, atrib);
            ValueExpression result = new ExpressionFactoryImpl()
                .createValueExpression(ec, expr, boolean.class);
            return (Boolean) result.getValue(ec);
        }
        return true;
    }
}
```

O conteúdo do mapa de objetos que deve ser passado para o contexto de avaliação da expressão vai depender do domínio do componente que está sendo desenvolvido. Nesse caso, a expressão referencia os atributos da própria classe e, por isso, deveria ser passado um mapa com essas informações. Felizmente, neste livro, já desenvolvemos a classe `GeradorMapa` que gera um mapa com as propriedades do objeto e, nesse exemplo, iremos utilizá-lo para

gerar o mapa a ser usado na expressão de validação. No caso das funções, adicionamos funções que estariam disponíveis na própria classe `Validador`, que na listagem não possui nenhuma, mas apenas adicionando-as à classe, já estariam disponíveis.

Para finalizar, basta criar o contexto de avaliação e pedir que a expressão seja avaliada. Observe que foi passado como parâmetro para a criação de `ValueExpression`, o valor `boolean.class`, pois é esperado que a expressão retorne um valor booleano. Por esse exemplo é possível perceber que de forma bem simples é possível ter acesso a uma linguagem com vários recursos para a definição de expressões, permitindo expressar regras sofisticadas dentro das anotações.

6.4 ASSOCIANDO COMPORTAMENTO A ANOTAÇÃO

"Entidades de software (classes, módulos, funções etc.) devem ser abertas para extensão, mas fechadas para modificação." - Princípio Aberto/Fechado - Bertrand Meyer

As anotações na linguagem Java são elementos que não possuem comportamento, podendo possuir somente dados. Porém, em algumas situações, é desejável associar um determinado comportamento, ou uma determinada lógica ao metadado que está sendo configurado. Um exemplo disso são as regras de validação apresentadas na seção anterior. De certa forma, o uso da EL dentro de um atributo do tipo `String` nada mais é do que uma forma de definir uma lógica pequena dentro da anotação.

A questão é que nem sempre é possível utilizar uma linguagem

de expressões, principalmente quando é preciso expressar uma lógica mais complexa. Por mais que utilizando a EL seja possível criar funções e disponibilizar para serem utilizadas, de certa forma deve-se saber que tipo de lógica pode ser necessário nas expressões. Em certos casos é preciso dar ao desenvolvedor a liberdade de ligar a anotação à sua própria lógica. Esta seção apresenta estratégias que podem ser utilizadas para se fazer isso.

Adicionando classe à anotação

Uma forma simples de se associar uma lógica a uma anotação é utilizando um atributo do tipo `Class`, para o qual será configurada a classe que possui a lógica desejada. Dessa forma, o componente pode instanciar a classe passada como parâmetro e executar qualquer um de seus métodos. A partir do tipo genérico do objeto `Class`, é possível restringir as classes que podem ser passadas, permitindo somente as que implementem uma determinada interface ou que estendam uma determinada classe. Sendo assim, essa abstração pode exigir que a classe configurada possua um método com assinatura conhecida que possui a lógica que precisa ser associada à anotação. Obviamente, nada também impede que o método a ser invocado seja identificado através de anotações como visto em exemplos de capítulos anteriores.

Para exemplificar, vamos incrementar a classe `Validador` de forma a ela ler uma anotação que recebe uma classe que possui um método de validação. A listagem a seguir mostra a interface `RegraValidacao`, que define o método `validar()`. Observe que esse método recebe o mapa com a lista de atributos da classe. Na mesma listagem é apresentado o exemplo de uma classe que implementa essa interface, no caso, que verifica se o valor do

atributo `idade` no mapa é maior que 18.

Interface para implementação de uma nova regra de validação:

```
public interface RegraValidacao {
    public boolean validar(Map<String,Object> atributos);
}

//Exemplo de implementação
public class MaiorDeIdade implements RegraValidacao {
    public boolean validar(Map<String, Object> atributos) {
        int idade = (int) atributos.get("idade");
        return idade >= 18;
    }
}
```

Já na próxima listagem é apresentada a anotação `@ClasseRegra`, que recebe em seu atributo `value` a classe com a validação. Observe como a cláusula `extends` é utilizada com o tipo genérico para amarrar as classes que podem ser passadas para o atributo da anotação.

Anotação para receber a classe com a regra de validação:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface ClasseRegra {
    Class<? extends RegraValidacao> value();
}
```

Para completar o exemplo, a listagem a seguir mostra um incremento do método `valido()` para considerar a anotação `@ClasseRegra`. Observe que antes de processar é verificado se a anotação está presente e se o objeto já foi considerado não válido pelo processamento da anotação que utiliza expressões. Em seguida é realizada a seguinte sequencia de ações: a anotação é recuperada; a classe é recuperada do atributo da anotação; e a

classe é instanciada. Depois disso, basta invocar na classe o método de validação.

Anotação para receber a classe com a regra de validação:

```
public boolean valido(Object obj) throws Exception{
    Class<?> c = obj.getClass();
    boolean resultado = true;
    Map<String,Object> atrib = GeradorMapa.gerarMapa(obj);
    if(c.isAnnotationPresent(Regra.class)){
        //validação com expressão omitida
    }if(resultado && c.isAnnotationPresent(ClasseRegra.class)){
        ClasseRegra cr = c.getAnnotation(ClasseRegra.class);
        Class<? extends RegraValidacao> clazz = cr.value();
        RegraValidacao rv = clazz.newInstance();
        resultado = rv.validar(atrib);
    }
    return resultado;
}
```

Estendendo o esquema de anotações

Imagine a partir do exemplo apresentado que se deseje fazer uma validação para pessoas idosas, cuja idade é maior ou igual que 65, ou que verifique outro tipo de restrição etária. Poderíamos, em vez de ter uma classe chamada `MaiorDeIdade`, ter uma classe chamada `RestriçãoEtaria`, que recebe como parâmetro a idade mínima para que o objeto seja considerado válido. O problema disso é que a classe de validação é instanciada pelo componente, o qual precisaria setar esse parâmetro de alguma forma. Outra questão é onde esse parâmetro seria definido, pois a anotação só possui atributo para receber a classe, e essa informação é específica dessa classe.

A solução nesse caso é adicionar no componente a capacidade de ler novas anotações criadas pelos desenvolvedores,

possibilitando que seja criada uma anotação para a restrição de idade que receba a idade mínima como parâmetro. Para que isso seja possível, é preciso que a nova anotação indique a classe que saiba interpretá-la e, no caso desse componente, que saiba executar aquela validação. Isso pode ser feito adicionando uma anotação na própria anotação para indicar a classe associada a ela.

Para entendermos como isso seria implementado na prática, vamos implementar essa solução no componente de validação, mostrando como seria feito para criamos uma anotação que validasse se a idade é maior que um valor passado como parâmetro. O primeiro passo seria adicionarmos na interface `RegraValidacao` um método para que a implementação possa ser inicializada com uma anotação, como mostrado na listagem a seguir. A partir do método `inicializar()` a implementação recebe a anotação podendo utilizar suas informações para parametrizar a classe.

Adição do método de inicialização através da anotação:

```
public interface RegraValidacao<E extends Annotation> {
    public void inicializar(E an);
    public boolean validar(Map<String,Object> atributos);
}
```

Em seguida, é necessária a criação da anotação que será utilizada para associar a classe com a regra de validação com a anotação que irá configurar essa regra. Na próxima listagem, a anotação `@AnotacaoRegra` é similar à `@ClasseRegra`, com a diferença de que ela deve ser adicionada na anotação que será adicionada na classe, e não na classe diretamente.

Anotação para associar a classe com a regra de validação à anotação:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface AnotacaoRegra {
    Class<? extends RegraValidacao> value();
}
```

Por fim, é preciso adicionar ao método `valido()` o suporte à leitura desse tipo de anotação. A listagem a seguir mostra como seria a implementação, omitindo a implementação da execução de expressões e da execução da classe configurada na anotação `@ClasseRegra`. Como não é possível recuperar a anotação diretamente, é preciso percorrer todas as anotações da classe, verificando quais possuem a anotação `@AnotacaoRegra`. Quando uma anotação for encontrada, deve-se instanciar a classe que implementa a interface `RegraValidacao` e passar sua anotação original. Em seguida, depois que a instância for inicializada, chama-se o método de validação.

Adição do suporte a um esquema de anotações extensível no método de validação:

```
public boolean valido(Object obj) throws Exception{
    Class<?> c = obj.getClass();
    boolean resultado = true;
    Map<String,Object> atrib = GeradorMapa.gerarMapa(obj);
    if(c.isAnnotationPresent(Regra.class)){
        //validação com expressão omitida
    }
    if(resultado && c.isAnnotationPresent(ClasseRegra.class)){
        //validação com classe omitida
    }
    if(resultado){
        for(Annotation an : c.getAnnotations()){
            Class<? extends Annotation> anotType =
                                            an.annotationType();
            if(anotType.isAnnotationPresent(
             AnotacaoRegra.class)){
                AnotacaoRegra ar =
                    anotType.getAnnotation(AnotacaoRegra.class);
```

```
                Class<? extends RegraValidacao> clazz =
                                                    ar.value();
                RegraValidacao rv = clazz.newInstance();
                rv.inicializar(an);
                resultado = resultado && rv.validar(atrib);
            }
        }
    }
    return resultado;
}
```

Como não existe herança entre anotações, uma forma de agrupar anotações é fazendo com que possuam um metadado em comum, nesse caso, uma anotação. Sendo assim, é adicionada uma anotação para marcar novas anotações que são de interesse de um determinado componente. Como a nova pode possuir atributos específicos que irão parametrizar algum processamento, passa-se como parâmetro para a anotação de marcação a classe responsável por ler a nova anotação. Dessa forma, é possível criar novas anotações para um componente e estender seu processamento ligando classes para leitura e processamento delas.

Para exemplificar a criação de uma nova anotação, na listagem a seguir está apresentada a anotação `@IdadeMaiorQue`, a qual deve ser adicionada para a validação se a idade é maior que um determinado valor. Observe que ela possui um atributo `value` para parametrizar a idade mínima válida. A anotação `@AnotacaoRegra` configura qual é a classe que deverá ser utilizada para a validação, que no caso é `RestricaoEtaria`.

Anotação para receber a classe com a regra de validação:

```
@AnotacaoRegra(RestricaoEtaria.class)
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface IdadeMaiorQue {
    int value();
```

```
}
```

Finalizando o exemplo, a próxima listagem mostra a implementação de `RegraValidacao` ligada à anotação, que será instanciada pelo componente. O tipo genérico definido na interface, que recebe o tipo da anotação `@IdadeMaiorQue`, determina que esse será o tipo recebido pelo método `inicializar()`. Esse método recebe a anotação e parametriza a validação atribuindo o valor da propriedade `value` da anotação para a variável `restricao`. Essa variável é então utilizada no método `validar()` para a verificação da idade mínima.

Anotação para receber a classe com a regra de validação:

```
public class RestricaoEtaria implements
 RegraValidacao<IdadeMaiorQue> {

    private int restricao;

    public void inicializar(IdadeMaiorQue an) {
        restricao = an.value();
    }
    public boolean validar(Map<String, Object> atributos) {
        int idade = (int) atributos.get("idade");
        return idade >= restricao;
    }
}
```

6.5 MAPEAMENTO DE ANOTAÇÕES

Quando a quantidade de anotações se torna muito grande, isso pode gerar poluição no código, dificultando sua leitura e manutenção. Algumas das estratégias apresentadas podem ajudar a diminuir esse problema, principalmente quando as anotações pertencem a um mesmo esquema e são consumidas por um mesmo componente. Porém, imagine uma classe de domínio que

precise ser mapeada para a base de dados, para um documento XML e que possui regras de validação configuradas com anotações. Um outro exemplo, seria uma classe com métodos de negócio que precisa que sejam configuradas questões como transações, logging e segurança. Nesses casos, as anotações podem vir de fontes diferentes e ser consumidas por componentes independentes, dificultando a criação de uma estratégia única para lidar com essas questões. Considere a próxima listagem como exemplo dessa situação.

Classe com várias anotações:

```
@AnotacaoA("info")
@AnotacaoB(propriedade="prop",valor=23)
@AnotacaoC
public class ClasseAnotada{
    //...
}
```

Nesse caso, uma estratégia à qual o componente pode dar suporte é o mapeamento de anotações. Quando essa estratégia é utilizada, as anotações do componente podem ser definidas não somente no elemento de código, mas também em anotações contidas nesse elemento. Dessa forma, é possível representar um conjunto de anotações no elemento com apenas uma anotação, para a qual as outras foram mapeadas. A listagem a seguir mostra um exemplo de uso dessa estratégia. Nesse caso, a `@AnotacaoMapeada` recebe as anotações que deveriam ser adicionadas no elemento de código, o qual recebe somente essa anotação. Essa prática se torna ainda mais interessante quando a mesma configuração é recorrente em diversos elementos.

Solução de mapeamento de anotações:

```
//Definição da anotação de mapeamento
```

```
@AnotacaoA("info")
@AnotacaoB(propriedade="prop",valor=23)
@AnotacaoC
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface AnotacaoMapeada {
}

//Definição da classe
@AnotacaoMapeada
public class ClasseAnotada{
    //...
}
```

Uma importante observação é que, para uma anotação poder ser mapeada, ela deve permitir que sua adição possa ser feita, não somente no tipo de elemento alvo, mas também em outras anotações. Sendo assim, em sua configuração de `@Target` deve ser acrescentado `ANNOTATION_TYPE`, além do tipo original.

Implementando a leitura do mapeamento

Para dar suporte ao mapeamento de anotações, o componente deve buscá-las não somente nos elementos, mas também nas que estão contidas nele. Caso o componente considere que possam existir anotações de mapeamento dentro de anotações de mapeamento, deve-se definir um ponto de parada, como uma profundidade máxima ou anotações em que não se deve procurar, como as do pacote `java.lang.annotation`. Uma forma de buscar as que são procuradas somente nas de mapeamento é criar uma anotação para que o desenvolvedor precise declarar explicitamente que ela é utilizada para mapear outras anotações. A listagem a seguir apresenta a anotação que irá fazer essa marcação no exemplo.

Anotação para marcar as anotações que mapeiam outras:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Mapeamento {
}
```

A listagem a seguir mostra como seria implementada a busca de uma anotação utilizando mapeamento. Inicialmente, verifica-se se a própria anotação está presente diretamente no elemento, e em caso positivo ela é retornada. Em seguida, percorrem-se todas as outras anotações à procura de uma que declare que faz mapeamento, ou seja, que possua a anotação `@Mapeamento`. Nesse caso, o próprio método é chamado de forma recursiva passando o tipo da anotação como parâmetro. Caso essa chamada retorne a anotação, ela é retornada, e se retornar `null` a busca continua nas outras anotações. Por fim, se após percorrer todas as anotações ela não for encontrada, o método retorna `null`.

Método que busca anotação considerando que a possibilidade de mapeamento:

```
public static Annotation getAnnotation(AnnotatedElement ae,
                    Class<? extends Annotation> anot){
    if(ae.isAnnotationPresent(anot)){
        return ae.getAnnotation(anot);
    }
    for(Annotation each : ae.getAnnotations()){
        Class<? extends Annotation> annotationType =
            each.annotationType();
        if(annotationType.isAnnotationPresent(
         Mapeamento.class)){
            Annotation a = getAnnotation(annotationType, anot);
            if(a != null)
                return a;
        }
    }
    return null;
}
```

É importante ressaltar que, nesse caso, o mapeamento é

simples e uma anotação sem atributos é utilizada para representar diversas outras anotações, que podem ou não ter atributos. Seria possível fazer um mapeamento mais complexo, em que valores da anotação de mapeamento seriam utilizados para determinar valores nas anotações mapeadas, porém, até o momento, ainda não vi um framework que tenha utilizado esse tipo mais sofisticado.

Anotações de domínio

As anotações representam uma semântica relacionada ao domínio que elas representam. Por exemplo, uma anotação que diz qual deve ser a estratégia de propagação de transações é relacionada com o domínio de gerenciamento de transações, mas acaba sendo adicionada em uma classe de negócio. Isso, de certa forma, causa uma invasão dessas informações, muitas vezes relacionadas com requisitos não funcionais da aplicação, em classes que são relacionadas ao negócio da aplicação.

Um exemplo dessa situação está mostrado na listagem a seguir. Os métodos `cadastrarUsuario()` e `bloquearUsuario()` recebem uma série de anotações que serão utilizadas por outros componentes. No caso, em termos de logging são funcionalidades críticas de serem registradas, devem ser executadas em um contexto transacional e só podem ser executadas por administradores. Observe que ambos os métodos possuem as mesmas anotações.

Anotações relacionadas com requisitos não funcionais em classe de negócio:

```
@Logging(CRITICO)
@Transacao(OBRIGATORIA)
@PapeisPermitidos("admin")
```

```
public void cadastrarUsuario(Usuario u){...}

@Logging(CRITICO)
@Transacao(OBRIGATORIA)
@PapeisPermitidos("admin")
public void bloquearUsuario(long idUsuario){...}
```

Se fizermos uma análise do negócio da aplicação para avaliar o que esses métodos possuem em comum, será possível perceber que são métodos de administração do sistema. É por esse motivo que eles possuem todos esses requisitos relacionados às outras características não funcionais. Sendo assim, a ideia da anotação de domínio é utilizar nas classes de negócio anotações que possuem uma semântica relacionada com o negócio, e não com outros interesses da aplicação. Dessa forma, essa anotação de domínio pode ser utilizada para fazer um mapeamento para as anotações relacionadas com outros domínios.

A listagem a seguir mostra como a anotação de domínio poderia ser criada e utilizada na classe de negócio. No caso, `@AdministracaoSistema` recebe não somente as anotações relacionadas aos interesses não funcionais, mas também a anotação `@Mapeamento`, para poder ser reconhecida. Dessa forma, os métodos recebem apenas a anotação de domínio, que adiciona ao método uma metainformação relacionada com o domínio da aplicação. Observe que dessa forma fica mais simples alterar configurações de uma forma mais geral, além de simplificar a configuração de outros métodos do mesmo tipo.

Utilizando anotação de domínio:

```
//definição da anotação de domínio
@Logging(CRITICO)
@Transacao(OBRIGATORIA)
@PapeisPermitidos("admin")
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Mapeamento
public @interface AdministracaoSistema {
}

//definição dos métodos
@AdministracaoSistema
public void cadastrarUsuario(Usuario u){...}
@AdministracaoSistema
public void bloquearUsuario(long idUsuario){...}
```

É importante ressaltar que as anotações de domínio só podem ser utilizadas dessa forma por componentes que suportem esse tipo de leitura. Por exemplo, as anotações da API CDI (CORDEIRO, 2013) do Java EE podem ser mapeadas para outras anotações. Mas é importante ressaltar que simplesmente mapear as anotações não é o mesmo que criar uma anotação de domínio, se ela não possuir um significado dentro do negócio da aplicação. A seção *Inserindo e substituindo anotações com AspectJ* mostra como o AspectJ pode ser utilizado para o mapeamento de anotações em frameworks que não suportam essa prática diretamente.

Combinando várias estratégias de leitura de anotações

Este capítulo apresentou diversas práticas que podem ser utilizadas na leitura e configuração de anotações. Foram também apresentadas funções que fazem a leitura das anotações seguindo cada uma das estratégias. Nada impede que essas técnicas sejam combinadas em um determinado componente, caso isso seja adequado. Porém, é preciso criar uma rotina de recuperação de anotações que combine as duas ou mais formas de busca desejadas, o que pode não ser uma tarefa trivial. De qualquer forma, um conselho sempre sábio é testar as diferentes possibilidades, principalmente as combinadas, para verificar se os metadados estão sendo recuperados corretamente.

6.6 CONSIDERAÇÕES FINAIS

Este capítulo trouxe diversas práticas para o uso de anotações, mostrando como diversos problemas podem ser solucionados. Inicialmente, foi abordada a questão da adição de múltiplas configurações semelhantes no mesmo elemento. Em seguida, foram mostradas estratégias para diminuição da quantidade de anotações com o uso de convenções de código e configurações gerais. Também foi falado sobre a adição de comportamento agregado as anotações, apresentando estratégias como o uso de expressões, a definição de atributos para configuração de classes e a abordagem de extensão do esquema de anotações com uma respectiva extensão do comportamento associado às novas

anotações. Finalmente, o capítulo terminou falando sobre o mapeamento de anotações, e como isso pode ser utilizado para a criação de metadados mais próximos ao domínio da aplicação.

É importante ressaltar novamente que essas práticas foram retiradas de diversos frameworks que são amplamente utilizados pela indústria. Como pesquisador dessa área, é possível perceber que esse conhecimento foi amadurecendo ao longo do tempo, havendo diversos problemas nas primeiras soluções criadas com anotações. A API CDI (CORDEIRO, 2013), citada na seção de mapeamento de anotações, é um exemplo de uma API mais madura em termos de utilização de anotações, que utilizou diversos dos conceitos apresentados neste capítulo. Deixo como exercício aos desenvolvedores observarem algumas das APIs e frameworks com anotações que utilizam, buscando pontos onde ela poderia ser melhorada com o uso de alguma dessas práticas.

Capítulo 7

CONSUMINDO E PROCESSANDO METADADOS

"Se você automatiza uma bagunça, você terá uma bagunça automatizada." - Rod Michael

Até o momento, nos capítulos anteriores, foram abordados componentes menores nos quais a leitura dos metadados acaba ficando misturada com o processamento de sua lógica. Para classes pequenas, que acabam consumindo uma ou duas anotações isso pode ser aceitável, porém, quando o esquema de metadados é grande e a lógica é mais complicada, essa mistura pode dificultar bastante não somente a compreensão do código mas também essa evolução.

Este capítulo fala sobre como estruturar um componente baseado em metadados preparando-o para crescer. As práticas apresentadas aqui foram resultado de uma pesquisa que envolveu o estudo de diversos frameworks existentes e do desenvolvimento de diversos outros. A partir delas, são identificados os principais pontos onde um framework que utiliza metadados pode disponibilizar um ponto de extensão, permitindo a adição de novas

funcionalidades sem a necessidade de alterar o que já está pronto.

A estrutura deste capítulo irá seguir um *running example*, que é um exemplo que vai se desenvolvendo ao longo das seções. A primeira seção irá apresentar o exemplo, explicando os detalhes de seu funcionamento. Em seguida, nas seções subsequentes, o código vai sendo refatorado de forma que as práticas apresentadas sejam adicionadas ao exemplo. Dessa forma, será mostrado o caminho que parte de um componente com toda a lógica em uma única classe, para um componente extensível e mais bem estruturado.

7.1 COMPONENTE BASE PARA O EXEMPLO

O componente que será utilizado como base para esse exemplo realiza uma comparação entre as propriedades de dois objetos, retornando uma lista com as diferenças. Um de seus diferenciais é que ele permite que sejam utilizadas anotações para configurar como deve ser realizada a comparação para cada propriedade. Sendo assim, a anotação `@NaoComparar` pode ser adicionada em uma propriedade para que seus valores não sejam comparados pelo algoritmo. Além dessa, também é possível adicionar a anotação `@Tolerancia` para a comparação de valores numéricos, sendo esta a diferença mínima entre os valores para que sejam considerados diferentes. Por fim, a anotação `@Profundo` registra que a propriedade é um objeto composto, cujas propriedades também precisam ser comparadas uma a uma. O método retorna uma lista de objetos da classe `Diferenca` cujas instâncias possuem o valor novo, o valor antigo e o nome da propriedade.

A listagem a seguir mostra o método principal do componente que será apresentado. Ele pode parecer um pouco grande e

complicado a princípio, mas pode ter certeza de que ele pode se tornar bem pior em um componente desse tipo que não se preocupar com a divisão de responsabilidades. A principal parte do corpo desse método é uma iteração pelos métodos da classe dos objetos comparados buscando métodos setter sem a anotação `@NaoComparar`. Para cada propriedade encontrada, inicialmente são recuperados o seu valor nos dois objetos, e, em seguida, de acordo com as anotações que ele possuir, é chamado um método para fazer a comparação propriamente dita.

Código do método principal do componente de comparação:

```
public <E> List<Diferenca> comparar(E velho, E novo){
    List<Diferenca> difs = new ArrayList<Diferenca>();
    Class<?> clazz = novo.getClass();
    for (Method m : clazz.getMethods()) {
        try {
            boolean isGetter = m.getName().startsWith("get");
            boolean semParam =
                (m.getParameterTypes().length == 0);
            boolean naoGetClass =
                !m.getName().equals("getClass");
            boolean paraComparar =
                !m.isAnnotationPresent(NaoComparar.class);
            if (isGetter && semParam && naoGetClass &&
             paraComparar) {
                Object valorVelho = m.invoke(velho);
                Object valorNovo = m.invoke(novo);
                String nomeProp = m.getName().substring(3, 4)
                    .toLowerCase() + m.getName().substring(4);
                if (m.isAnnotationPresent(Tolerancia.class)) {
                    Tolerancia tolerancia = m
                            .getAnnotation(Tolerancia.class);
                    compararTolerancia(difs, tolerancia.value(),
                        valorNovo, valorVelho, nomeProp);
                } else if (m.isAnnotationPresent(Profundo.class)
                 && valorNovo != null && valorVelho != null) {
                    compararProfundo(difs, nomeProp, valorVelho,
                    valorNovo);
                } else {
```

```
                    compararNormal(difs, nomeProp, valorNovo,
                    valorVelho);
                }
            }
        } catch (Exception e) {
            throw new RuntimeException(
                "Erro recuperando propriedade", e);
        }
    }
    return difs;
}
```

A próxima listagem mostra os métodos auxiliares responsáveis por fazer as comparações propriamente ditas. O primeiro método, chamado `compararNormal()`, faz a comparação default entre objetos, primeiramente verificando a presença de valores nulos e depois fazendo a comparação a partir do método `equals()`. Em seguida, o método `comparaTolerancia()` faz a comparação quando a anotação `@Tolerancia` está presente. Nesse caso, é obtido o valor absoluto da diferença entre os valores da propriedade, considerando que existe uma diferença somente quando esta é maior que a tolerância. Por fim, o método `compararProfundo()`, invocado na presença de `@Profundo`, utiliza os valores para chamar recursivamente o método `comparar()`, o qual irá comparar cada um de suas propriedades. As diferenças retornadas por ele são adicionadas na lista, adicionando em sua identificação o nome da propriedade original seguida de um ponto. Observe que não iremos nos preocupar com ciclos de objetos, que podem causar um laço recursivo infinito.

Métodos auxiliares que fazem cada tipo de comparação:

```
private void compararNormal(List<Diferenca> difs, String prop,
        Object newValue, Object oldValue) {
    if (newValue == null) {
        if (oldValue != null) {
            difs.add(new Diferenca(prop, newValue, oldValue));
```

```
        }
    } else if (!newValue.equals(oldValue)) {
        difs.add(new Diferenca(prop, newValue, oldValue));
    }
}
private void compararTolerancia(List<Diferenca> difs,
 double tolerance, Object newValue, Object oldValue,
 String prop) {
    double dif =
        Math.abs(((Double) newValue) - ((Double) oldValue));
    if (dif > tolerance) {
        difs.add(new Diferenca(prop, newValue, oldValue));
    }
}
private void compararProfundo(List<Diferenca> difs,
 String nomeProp, Object valorVelho, Object valorNovo) {
    List<Diferenca> difsProp = comparar(valorNovo, valorVelho);
    for (Diferenca d : difsProp) {
        d.setPropriedade(nomeProp + "." + d.getPropriedade());
        difs.add(d);
    }
}
```

Para ver um exemplo de como ficaria uma classe com essas anotações, a listagem a seguir apresenta a classe `Pessoa`, que possui propriedades anotadas com as anotações do componente. A propriedade `peso` recebeu a anotação `@Tolerancia(0.01)` para configurar que essa deve ser a diferença mínima para que os valores não sejam considerados iguais. Em seguida, a propriedade `endereco` recebeu a anotação `@Profundo` para que suas propriedades, como `rua` e `numero`, sejam incluídas na comparação. Por fim, a propriedade `idade` foi configurada com `@NaoComparar` para que o algoritmo de comparação a ignore.

Exemplo de utilização das anotações do componente:

```
public class Pessoa {

    private String nome;
    private double peso;
```

```
    private int idade;
    private Endereco endereco;
    public Pessoa(String nome, double peso, int idade,
     Endereco endereco) {
        this.nome = nome;
        this.peso = peso;
        this.idade = idade;
        this.endereco = endereco;
    }
    public String getNome() {
        return nome;
    }
    @Tolerancia(0.1)
    public double getPeso() {
        return peso;
    }
    @NaoComparar
    public int getIdade() {
        return idade;
    }
    @Profundo
    public Endereco getEndereco() {
        return endereco;
    }
    //métodos set omitidos
}
```

Para exemplificar o uso do componente e verificar se ele se comporta como esperado para a classe `Pessoa`, a próxima listagem apresenta um teste de unidade que explora diferentes cenários de comparação. O método `inicializacaoVariaveis()` cria duas instâncias da classe `Pessoa` com os mesmos dados, que servirão como base em cada teste. Nesse caso, a classe `Pessoa` pode ser considerada como uma mock class para o teste do componente. Esses testes também serão utilizados para ajudar a verificar se as refatorações no componente irão alterar seu comportamento.

Código de inicialização do teste de unidade que verifica a

execução do componente na classe Pessoa:

```
public class TesteComparacao {

    private Pessoa p1, p2;
    private ComponenteComparacao c;

    @Before
    public void inicializaVariaveis(){
        c = new ComponenteComparacao();
        p1 = new Pessoa("Fulano", 70.5, 23,
           new Endereco("Rua Brasil", "200"));
        p2 = new Pessoa("Fulano", 70.5, 23,
                    new Endereco("Rua Brasil", "200"));
    }
    //testes omitidos
}
```

A listagem a seguir mostra os três primeiros testes do componente. O primeiro deles realiza a verificação das duas instâncias de `Pessoa` sem fazer nenhuma modificação e, como resultado, deve receber uma lista vazia. O teste `nomeDiferente()` modifica o nome de um dos dois objetos e verifica se a diferença é encontrada pelo componente. Por fim, o teste `ignoraIdade()` modifica a propriedade `idade` que, por possuir a anotação `@NaoComparar`, não deve ser listada entre as diferenças.

Testes envolvendo propriedades simples:

```
@Test
public void semDifereca() {
    List<Diferenca> l = c.comparar(p1, p2);
    assertTrue(l.isEmpty());
}
@Test
public void nomeDiferente() {
    p2.setNome("Ciclano");
    List<Diferenca> l = c.comparar(p1, p2);
    assertEquals(1, l.size());
    assertEquals("nome", l.get(0).getPropriedade());
```

```
}
@Test
public void ignoraIdade() {
    p2.setIdade(24);
    List<Diferenca> l = c.comparar(p1, p2);
    assertTrue(l.isEmpty());
}
```

A próxima listagem mostra os testes para a comparação de propriedades que possuem regras especiais. Os dois primeiros testes focam na propriedade `peso` que possui a anotação `@Tolerancia`, sendo que um teste introduz um valor maior que a tolerância e o outro, um valor menor. O último teste foca na propriedade que possui a anotação `@Profundo`, modificando uma propriedade interna ao atributo `endereco` e verificando se essa diferença é encontrada.

Testes de comparação de propriedades com regras especiais:

```
@Test
public void maiorTolerancia() {
    p2.setPeso(70.8);
    List<Diferenca> l = c.comparar(p1, p2);
    assertEquals(1, l.size());
    assertEquals("peso", l.get(0).getPropriedade());
}
@Test
public void menorTolerancia() {
    p2.setPeso(70.55);
    List<Diferenca> l = c.comparar(p1, p2);
    assertTrue(l.isEmpty());
}
@Test
public void diferencaEndereco() {
    p2.getEndereco().setRua("Rua Argentina");
    List<Diferenca> l = c.comparar(p1, p2);
    assertEquals(1, l.size());
    assertEquals("endereco.rua", l.get(0).getPropriedade());
}
```

Observando o código do componente, é possível perceber que existe uma mistura na leitura e processamento de anotações. Para perceber isso, uma forma seria imaginarmos como a adição de novas funcionalidades poderia exigir modificações em todo o componente. Se os metadados, por exemplo, pudessem ser definidos usando XML, isso já traria grandes mudanças para o componente. A adição de uma nova anotação de comparação seria um outro exemplo que exigiria a adição de um novo condicional para a execução da comparação.

7.2 SEPARANDO A LEITURA DO PROCESSAMENTO DE METADADOS

"Eu disse ado a ado, cada um no seu quadrado." - Sharon Acioly, Dança do Quadrado

O primeiro passo na refatoração do componente seria a separação das responsabilidades de leitura e processamento de metadados. A mistura dessas duas responsabilidades distintas nesse tipo de componente dificulta qualquer mudança na estratégia de definição de metadados. A adição de funcionalidades e novas possibilidades acabam resultando na criação de novos condicionais que irão afetar a legibilidade e a qualidade do código do próprio componente.

Para os que estão acostumados com o desenvolvimento de sistemas de informação, é possível fazer um paralelo entre essa questão e o processamento de informações armazenadas em uma base de dados. Se a aplicação é feita de forma acoplada ao acesso ao banco de dados para recuperar as informações, será difícil modificar a forma de armazenamento e até mesmo obter dados de

uma fonte diferente. Por outro lado, se existe uma classe isolada responsável por aquela leitura, ela pode ser facilmente substituída ou estendida sem afetar outras partes do código. Sendo assim, da mesma forma que é positivo separar o processamento da leitura de informação, o mesmo é válido quando tratamos de metadados.

Criando o container de metadados

Para ser possível separar a leitura do processamento de metadados, é preciso ter uma classe na qual se possam armazenar os metadados. Ela será utilizada por classes que fazem a leitura de metadados para inserir os que forem lidos. Adicionalmente, ela também será utilizada para obtenção dos metadados pelas classes que irão utilizá-los para processamento. A essa classe, que será a peça chave na comunicação entre leitores e processadores de metadados, se dá o nome de **container de metadados**, ou, como o nome do padrão em inglês, **metadata container**.

O diagrama apresentado na figura adiante apresenta a estrutura básica do componente. A classe representada como `LeitorMetadados` é responsável por criar a instância de `ContainerMetadados` e a popular com as informações obtidas da classe alvo. Apesar de no diagrama o `ContainerMetadados` ser representado com apenas uma classe, na prática, muitas vezes, a estrutura é formada por mais de uma. É muito comum, por exemplo, ter uma que representa os metadados de uma classe que possui uma lista de instâncias de classes que representam os metadados de seus métodos. A `ComponenteMetadados` é o ponto de entrada na execução do componente ou framework, onde a funcionalidade será invocada por outras classes. Ela obtém instâncias de `ContainerMetadados` da classe `LeitorMetadados`

e a utiliza para acessar os metadados da classe que está sendo processada pelo componente.

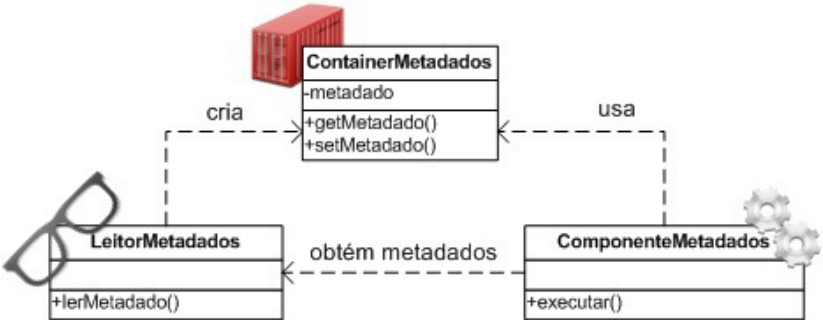

Figura 7.1: Diagrama mostrando a estrutura do componente com o uso do container de metadados

Para entendermos na prática como isso pode ser implementado, será visto como o componente de comparação poderia ser estruturado dessa forma. O primeiro passo é definirmos o container de metadados para guardar as informações necessárias. A listagem a seguir apresenta a classe `ContainerComparacao`, que armazena os metadados de comparação de uma classe. Ela basicamente possui um mapa com uma entrada para cada propriedade, cuja chave é o nome da propriedade e o valor é uma instância de `ComparacaoPropriedade`.

Container de metadados para o componente de comparação:

```
public class ContainerComparacao {

    private Map<String, ComparacaoPropriedade> props =
                                                new HashMap<>();

    public Set<String> getPropriedades(){
        return props.keySet();
```

```
    }
    public ComparacaoPropriedade
     getComparacaoPropriedade(String nome){
        return props.get(nome);
    }
    public void
     addComparacaoPropriedade(ComparacaoPropriedade cp){
        props.put(cp.getNome(), cp);
    }
}
```

A classe `ComparacaoPropriedade`, apresentada na listagem a seguir, armazena as informações de comparação referentes a uma propriedade. Os atributos `tolerancia` e `profundo` armazenam respectivamente a tolerância numérica, sendo zero caso não exista, e se deve ser feita ou não a comparação profunda.

Parte do container de metadados para uma propriedade:

```
public class ComparacaoPropriedade {

    private Method acesso;
    private String nome;
    private double tolerancia;
    private boolean profundo;

    public ComparacaoPropriedade(Method acesso) {
        this.acesso = acesso;
        this.nome =  acesso.getName().substring(3, 4)
                .toLowerCase() + acesso.getName().substring(4);
    }

    //métodos get e set omitidos

}
```

Observe que uma instância do método que deve ser utilizado para o acesso à propriedade também é armazenada. Apesar de ele pode ser recuperado posteriormente a partir do nome da propriedade, é interessante armazená-lo por questões de

desempenho. Como foi visto na seção *Usar reflexão tem um preço?*, a recuperação de uma instância de `Method` pode ser mais demorada que sua própria invocação. Dessa forma, armazenar no container os métodos que podem ser invocados muitas vezes é uma boa ideia.

A listagem a seguir mostra a implementação da leitura dos metadados de comparação para uma determinada classe. Observe que o método `lerMetadados()` recebe uma instância de `Class` como parâmetro e retorna um `ContainerComparacao`. Da mesma forma que as propriedades da classe eram percorridas na versão original do componente para comparação, nessa listagem elas são percorridas para inserir os dados no container de metadados. Um fator interessante a ser notado é que a presença da anotação `@NaoComparar` não resulta na configuração dessa informação, mas simplesmente faz com que essa propriedade não seja adicionada no container.

Implementação do leitor de metadados de comparação:

```
public class LeitorComparacao {

    public ContainerComparacao lerMetadados(Class<?> clazz){
        ContainerComparacao cont = new ContainerComparacao();
        for (Method m : clazz.getMethods()) {
            try {
                boolean isGetter =
                     m.getName().startsWith("get");
                boolean semParam =
                    (m.getParameterTypes().length == 0);
                boolean naoGetClass =
                    !m.getName().equals("getClass");
                boolean paraComparar =
                    !m.isAnnotationPresent(NaoComparar.class);
                if (isGetter && semParam && naoGetClass &&
                 paraComparar) {
                    ComparacaoPropriedade cp =
```

```
                    new ComparacaoPropriedade(m);
                if (m.isAnnotationPresent(
                 Tolerancia.class)) {
                    Tolerancia tolerancia =
                        m.getAnnotation(Tolerancia.class);
                    cp.setTolerancia(tolerancia.value());
                } else if (m.isAnnotationPresent(
                 Profundo.class)) {
                    cp.setProfundo(true);
                }
                cont.addComparacaoPropriedade(cp);
            }
        } catch (Exception e) {
            throw new RuntimeException(
             "Erro recuperando propriedade", e);
        }
    }
    return cont;
  }
}
```

Por fim, a classe representada na próxima listagem apresenta a classe `ComponenteComparacao` segundo essa nova estrutura. Observe que uma das primeiras ações realizadas é a recuperação de uma instância de `ContainerComparacao` a partir de um `LeitorComparacao`. Dessa forma, a partir dessa instância os metadados são recuperados para processamento. Apesar de o método ser recuperado de `ComparacaoPropriedade` para sua invocação, não existe nenhuma recuperação de anotações nessa classe que será responsável por realizar o processamento.

Classe que realiza a comparação utilizando o container de metadados:

```
public class ComponenteComparacao {

    private LeitorComparacao leitor = new LeitorComparacao();

    public <E> List<Diferenca> comparar(E velho, E novo){
        List<Diferenca> difs = new ArrayList<Diferenca>();
```

```
        Class<?> clazz = novo.getClass();
        ContainerComparacao cont = leitor.lerMetadados(clazz);
        for (String prop : cont.getPropriedades()) {
            try {
                ComparacaoPropriedade cp =
                 cont.getComparacaoPropriedade(prop);
                Method m = cp.getAcesso();
                Object valorVelho = m.invoke(velho);
                Object valorNovo = m.invoke(novo);
                if (cp.getTolerancia() != 0) {
                    compararTolerancia(difs, cp.getTolerancia(),
                        valorNovo, valorVelho, cp.getNome());
                } else if (cp.isProfundo() && valorNovo != null
                 && valorVelho != null) {
                    compararProfundo(difs, cp.getNome(),
                        valorVelho, valorNovo);
                } else {
                    compararNormal(difs, cp.getNome(),
                        valorNovo, valorVelho);
                }
            } catch (Exception e) {
                throw new RuntimeException(
                    "Erro recuperando propriedade", e);
            }
        }
        return difs;
    }
    //métodos de comparação omitidos
}
```

Comparando o componente estruturado dessa forma com o anterior, a princípio parece que não foi uma boa escolha, pois o código acabou ficando muito maior. O leitor de metadados precisa iterar pelos métodos da classe buscando as anotações e inserindo os dados no container de forma condicional e, de forma similar, a classe que faz a comparação itera pelas propriedades do container de metadados e também faz diferentes comparações de forma condicional. Peço ao leitor que aguarde as próximas seções, em que serão mostradas algumas práticas que só são possíveis de serem aplicadas devido a esse desacoplamento da leitura do

processamento de metadados. É a partir dessa separação que cada um desses processos poderá evoluir de forma independente!

Adicionando um repositório de metadados

A representação dos metadados específicos do componente na instância do container de metadados torna possível seu armazenamento e posterior reutilização. Sendo assim, se duas invocações distintas da funcionalidade do componente reutilizarem o mesmo container, isso irá poupar o tempo de processamento utilizado na leitura duplicada dos metadados da mesma classe. Uma forma de se fazer isso, é passar a classe que será utilizada no processamento no construtor do componente. Dessa forma, ele pode obter o container de metadados do leitor e armazená-lo para ser utilizado em diversas chamadas. Apesar de dessa forma haver uma reutilização da mesma instância do container, haveria leituras duplicadas caso diferentes instâncias do componente fossem criadas passando a mesma classe no construtor.

Uma outra forma de reaproveitar a leitura de metadados já feita para uma classe é a criação de um repositório de metadados. Ele seria acessado pelo componente e armazenaria internamente as instâncias do container de metadados que já tivessem sido criadas. A figura a seguir mostra como um repositório seria inserido na estrutura apresentada anteriormente. Observe que agora é a classe `RepositorioMetadados` que acessa `LeitorMetadados` para obter o container. Porém, esse acesso será feito apenas nos casos em que a instância do container da classe não estiver armazenada dentro do repositório.

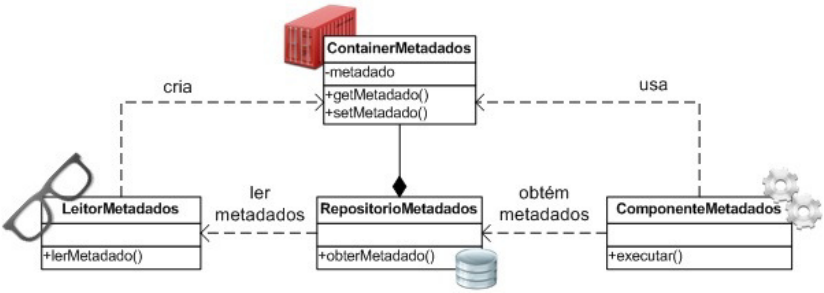

Figura 7.2: Diagrama que mostra a introdução do repositório de metadados

A próxima listagem apresenta a implementação de um repositório de metadados para o componente de comparação. Como a mesma instância do repositório deve ser compartilhada por todos os componentes, o padrão Singleton é utilizado para a recuperação do repositório. Observe que é definido um construtor privado para impedir que uma outra instância seja criada externamente. O repositório possui um mapa que é utilizado para associar as classes com seus respectivos containers. Caso seja solicitado a ele os metadados de uma classe que ainda não está no mapa, o leitor de metadados é utilizado para recuperar essa instância, a qual em seguida é armazenada no mapa e retornada.

Implementação do repositório de metadados para o componente de comparação:

```
public class RepositorioMetadados {

    private static RepositorioMetadados instancia =
            new RepositorioMetadados();
    public static RepositorioMetadados getInstance(){
        return instancia;
    }

    private Map<Class<?>, ContainerComparacao> cache =
        new HashMap<>();
```

```
    private LeitorComparacao leitor = new LeitorComparacao();

    private RepositorioMetadados(){}

    public ContainerComparacao getMetadados(Class<?> clazz){
        if(cache.containsKey(clazz)){
            return cache.get(clazz);
        }else{
            ContainerComparacao c = leitor.lerMetadados(clazz);
            cache.put(clazz, c);
            return c;
        }
    }
}
```

Para utilizar o repositório, a classe `ComponenteComparacao` deve ser modificada conforme mostrado na próxima listagem. Observe que a instância de `RepositorioMetadados` é recuperada através do método `getInstance()` e, em seguida, a instância de `ContainerComparacao` é recuperada utilizando `getMetadados()`. Nesse caso, o leitor de metadados será invocado apenas na primeira vez que o container de uma determinada classe for solicitado, sendo que em invocações sucessivas a mesma instância, armazenada em cache, será retornada.

Classe que armazena os metadados de um método:

```
public class ComponenteComparacao {

    public <E> List<Diferenca> comparar(E velho, E novo){
        List<Diferenca> difs = new ArrayList<Diferenca>();
        Class<?> clazz = novo.getClass();
        ContainerComparacao cont = RepositorioMetadados.
                getInstance().getMetadados(clazz);
        //resto do método omitido
    }
}
```

A existência de um repositório de metadados não é útil apenas por questões de desempenho, pois uma outra consequência positiva é a existência de um local central para a recuperação dos metadados. Isso é muito útil, por exemplo, em cenários onde existem várias classes diferentes que precisarão utilizar essas informações. Esse repositório pode até mesmo ser utilizado por classes de fora do componente ou framework que podem precisar daqueles metadados para outros fins.

7.3 ESTENDENDO A LEITURA DE METADADOS

A separação da leitura dos metadados e de seu processamento permite que cada um desses processos possa evoluir de forma independente. No que tange à leitura, existem várias questões importantes de serem tratadas, que na maioria das vezes dizem respeito à forma com que os metadados são definidos e como serão consumidos pelo componente. Esta seção irá apresentar as boas práticas que podem ser utilizadas no componente para tornar esse processo extensível e mais flexível.

Leitores de metadados intercambiáveis

A seção *Definição de metadados* mostrou que existem diversas formas de definir os metadados de uma classe e que cada uma delas possui suas vantagens e desvantagens. Mesmo o uso de anotações, que é a forma de definição de metadados mais explorada nesse livro, possui suas limitações. Sua principal restrição é relacionada à sua principal vantagem, que é o fato de serem próximas aos elementos que anotam por serem adicionadas

diretamente no código-fonte. Esse fato impede, por exemplo, sua utilização em classes de bibliotecas de terceiros, às quais normalmente não se possui acesso para a modificação do código-fonte. Dessa forma, é interessante possibilitar que o componente possa dar suporte a diversas alternativas de definição de metadados, para que se possa utilizar a abordagem mais adequada dependendo da aplicação. Inclusive, é interessante até permitir que novos leitores de metadados possam ser criados e utilizados pelo componente.

O diagrama da figura seguinte mostra a estrutura de uma solução para permitir que diversos leitores de metadados possam ser utilizados pelo componente. Nessa solução, `LeitorMetadados` agora define uma interface que deve ser implementada pelos leitores concretos. Eles podem possuir implementações para obter os metadados de diferentes fontes, como, por exemplo, de anotações, documentos XML e bancos de dados. Dessa forma, dependendo da implementação utilizada, os metadados serão lidos considerando diferentes tipos de definição.

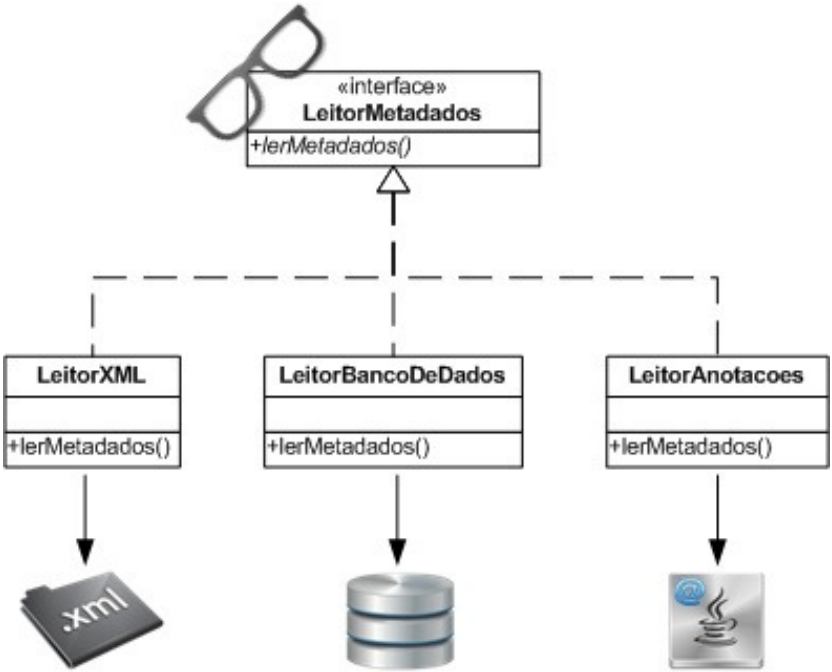

Figura 7.3: Provendo metadados de diversas fontes

Para que seja possível definir diferentes implementações a serem utilizadas pelo componente em diferentes cenários, é preciso prover uma estratégia de criação do leitor concreto que deverá ser efetivamente usado. Uma alternativa é a injeção de dependências, na qual o componente deve receber externamente a instância do leitor que deve utilizar. Ele pode receber essa instância a partir de seu construtor ou de um método setter. No caso de um diferente leitor precisar ser utilizado em cada chamada, ele pode inclusive ser passado como parâmetro. Uma outra abordagem seria o leitor ser recuperado através de uma fábrica, que seria responsável por retornar a implementação a ser usada pelo componente. A classe

para ser utilizada poderia, por exemplo, ser definida em um arquivo de configuração e instanciada por reflexão. A seção *Procurando por classes* mostra algumas ferramentas que podem ser usadas para buscar classes através de suas características, como interfaces que implementam ou anotações que possuem.

Para implementar a leitura dos metadados em uma fonte diferente no componente de comparação, o primeiro passo é criar uma abstração para representar a leitura de metadados. Sendo assim, foi criada a interface `LeitorComparacao`, que define apenas o método `lerMetadados()`, que recebe como parâmetro a classe e retorna o container de metadados. A classe que possuía esse nome foi renomeada como `LeitorComparacaoAnotacoes` e implementa essa interface que foi extraída.

A listagem a seguir mostra a classe `LeitorComparacaoXML` que realiza a leitura dos metadados a partir de um arquivo XML. Esse leitor irá buscar um arquivo que por convenção irá possuir o nome `<nome da classe>.comparacao`. O método `lerMetadados()` contém a lógica para localizar o arquivo e realizar o parsing do arquivo a partir de um handler SAX, que será implementado na própria classe a partir da extensão da classe `DefaultHandler`.

Leitor de metadados que obtém as informações de um arquivo XML:

```
public class LeitorComparacaoXML extends DefaultHandler
 implements LeitorComparacao{

    private ContainerComparacao container;
    private Class<?> clazz;

    @Override
```

```
    public ContainerComparacao lerMetadados(Class<?> clazz) {
        try {
            this.clazz = clazz;
            InputStream s =
             new FileInputStream(clazz.getName()+".comparacao");
            SAXParserFactory spf =
                SAXParserFactory.newInstance();
            SAXParser saxParser = spf.newSAXParser();
            XMLReader xmlReader = saxParser.getXMLReader();
            xmlReader.setContentHandler(this);
            xmlReader.parse(new InputSource(s));
            return container;
        } catch (Exception e) {
            throw new RuntimeException("Falha ao ler arquivo",
                                       e);
        }
    }

    //métodos do SAX omitidos
}
```

LEITURA DE XML COM SAX

SAX significa *Simple API for XML*, ou seja, API simples para XML. É uma forma de ler um documento XML baseado em eventos que são gerados a partir de elementos encontrados em sua leitura sequencial. Sendo assim, o inicio e o final da leitura de um documento é marcada com a chamada respectiva dos métodos `startDocument()` e `endDocument()`. De forma similar, cada vez que uma tag é aberta, o método `startElement()` é chamado, e toda vez que uma tag é fechada, o método `endElement()` é chamado. Dessa forma, para realizar o parse de um determinado documento XML, basta implementar esses métodos de forma a tratar os eventos que são de interesse. Por não carregar todo o documento XML em memória e ir realizando o parse à medida que vai sendo lido, essa abordagem é especialmente indicada para documentos muito grandes.

A listagem a seguir mostra os métodos da classe `LeitorComparacaoXML` que irão tratar os eventos do SAX. No método `startDocument()`, que é chamado no começo do processamento, é criada a instância de `ContainerComparacao` a ser utilizada para armazenar os metadados. O método `startElement()` irá procurar por elementos no XML chamados `<propriedade>`, os quais terão o atributo `nome` com o nome da propriedade e podem ter os atributos `tolerancia` e `profundo`. Observe que uma propriedade que deve ser ignorada simplesmente não deve ser adicionada no XML.

Métodos para o tratamentos dos eventos SAX dos metadados de comparação:

```
@Override
public void startDocument() throws SAXException {
    container = new ContainerComparacao();
}
@Override
public void startElement(String uri, String localName,
 String qName, Attributes attributes) throws SAXException {
    try {
        if(qName.equals("propriedade")){
            String nome = attributes.getValue("nome");
            Method m = clazz.getMethod("get"
                + nome.substring(0, 1).toUpperCase()
                + nome.substring(1));
            ComparacaoPropriedade cp =
                new ComparacaoPropriedade(m);
            if(attributes.getValue("tolerancia")!= null){
                cp.setTolerancia(Double.parseDouble(
                    attributes.getValue("tolerancia")));
            }
            if(attributes.getValue("profundo")!= null){
                cp.setProfundo(Boolean.parseBoolean(
                    attributes.getValue("profundo")));
            }
            container.addComparacaoPropriedade(cp);
        }
    } catch (Exception e) {
        throw new SAXException("Falha ao interpretar XML",e);
    }
}
```

As duas listagens a seguir mostram um exemplo de como seriam configurados os metadados através de XML para as classes `Pessoa` e `Endereco`. Eles estariam respectivamente nos arquivos `Pessoa.comparacao` e `Endereco.comparacao`. Observe que a tolerância e a comparação profunda foram configuradas exatamente como a configuração feita com anotações, devendo resultar no mesmo efeito em termos de lógica de comparação.

Definição dos metadados da classe Pessoa:

```
<?xml version="1.0" encoding="UTF-8"?>
<comparacao>
    <propriedade nome="nome"/>
    <propriedade nome="peso" tolerancia="0.1"/>
    <propriedade nome="endereco" profundo="true"/>
</comparacao>
```

Definição dos metadados da classe Endereco:

```
<?xml version="1.0" encoding="UTF-8"?>
<comparacao>
    <propriedade nome="rua"/>
    <propriedade nome="numero"/>
</comparacao>
```

Para completar o exemplo, basta prover a classe `RepositorioMetadados` com um método setter que permita configurar o leitor de metadados para ser utilizado. Dessa forma, pode-se configurar o leitor de XML nessa classe para que seja utilizado no lugar do leitor de anotações. Essa substituição foi feita na classe de teste mostrada anteriormente e os testes executaram com sucesso lendo os mesmos metadados, porém definidos de uma forma diferente.

Lendo metadados de mais de uma fonte

Algumas vezes, apenas poder ler os metadados de uma fonte ou de outra não é suficiente, pois é preciso colher os metadados de diversas fontes e combinar essas informações. A plataforma Java EE é o exemplo de uma API que utiliza muito esse tipo de recurso, permitindo que os metadados possam ser definidos na forma de anotações e em documentos XML. Enquanto as anotações acabam sendo mais fáceis de utilizar, a definição em XML permite que metadados possam ser adicionados e alterados em tempo de

deploy. Apesar de ver pouco isso sendo utilizado na prática, os EJBs foram concebidos para serem componentes reutilizáveis entre aplicações. Dessa forma, por exemplo, a configuração de todos os metadados em anotações pode definir questões muito específicas de uma aplicação e impedir seu reúso em outros contextos. Os metadados de controle de acesso seriam um exemplo de metadados que frequentemente seriam diferentes de uma aplicação para outra.

Uma outra funcionalidade interessante de um componente que consome metadados é a possibilidade de compor a leitura de metadados com vários leitores. Isso irá permitir que os metadados possam ir sendo complementados a partir da passagem por cada leitor. Isso irá facilitar que os metadados sejam definidos da forma mais conveniente em cada caso, podendo muitas vezes aproveitar informações já existentes na classe de outras formas.

A figura adiante mostra uma possível solução para a implementação de leitores compostos, utilizando o padrão de projeto Composite. Uma outra possibilidade de implementação seria utilizando um Chain of Responsibility. A classe `LeitorComposto` seria composta por outros leitores concretos e iria coordenar a leitura dos metadados por eles. O método de leitura de metadados deve, em vez de retornar um container de metadados, receber um como parâmetro para que possa complementar suas informações. Isso irá permitir que o mesmo container possa ser recebido e populado por todos os leitores.

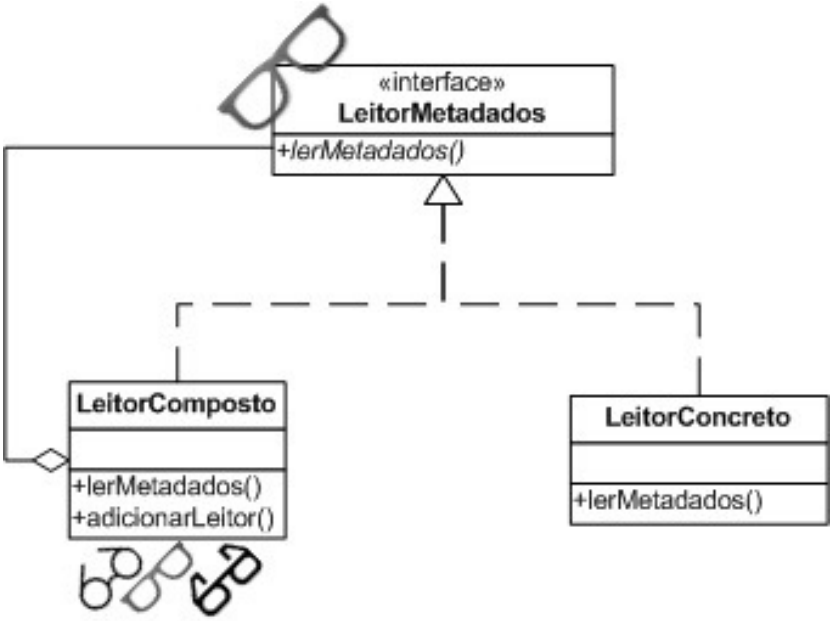

Figura 7.4: Compondo a leitura de metadados com diversos leitores

O primeiro passo para a implementação de uma leitura de metadados incremental no componente de comparação seria alterar a interface `LeitorMetadados` conforme a listagem a seguir. O método `lerMetadados()` agora retorna `void` e recebe como parâmetro a instância de `ContainerComparacao` .

Nova assinatura do método lerMetadados() da interface LeitorComparacao:

```
public interface LeitorComparacao {
    public void lerMetadados(Class<?> clazz,
        ContainerComparacao cont);
}
```

Obviamente somente a mudança na assinatura do método não

é suficiente e as implementações dos leitores de metadados também precisam se adaptar. Eles não devem mais criar um novo container de metadados para ser retornado e sim utilizar o que foi passado como parâmetro. No caso da classe `LeitorComparacaoXML`, por exemplo, o método `startDocument()` que inicializa a classe `ContainerComparacao` pode ser removido. Uma outra mudança que precisa ser feita é na classe que invoca os leitores, no caso `RepositorioMetadados`, pois ela precisa criar o container antes da invocação do método.

Quando se utiliza esse tipo de estratégia, deve-se levar em consideração que cada fonte de metadados não é única. Em outras palavras, deve-se levar em consideração que uma parte dos metadados pode já existir ou que pode ser configurada pelo próximo leitor. Por exemplo, a instância de `ComparacaoPropriedade` que era sempre criada no leitor, agora deve ser criada somente quando ela ainda não existir no container, conforme a listagem a seguir.

Criando um novo ComparacaoPropriedade somente se ainda não existir:

```
ComparacaoPropriedade cp = null;
if(container.getComparacaoPropriedade(nome) != null){
    cp = container.getComparacaoPropriedade(nome);
}else{
    cp = new ComparacaoPropriedade(m);
    container.addComparacaoPropriedade(cp);
}
```

Adicionalmente deve-se considerar a utilização de configurações negativas, que antes não eram necessárias. No exemplo do componente de comparação será feita primeira a leitura por anotação e, em seguida, a leitura por XML. Quando

havia somente um leitor, bastava não incluir a propriedade no XML para ela ser ignorada, porém isso não vai dar certo neste caso porque ela já foi incluída no container pelo leitor anterior. Agora, precisaria haver uma configuração explícita da propriedade dizendo que ela deve ser removida do container.

Outra questão que deve ser definida é como lidar com conflitos entre os metadados. No caso do componente de comparação, optou-se por dar prioridade aos metadados definidos no XML, de forma que eles irão sobrepor os metadados definidos em anotações. Porém, isso pode não ser adequado sempre. Por exemplo, é possível definir que um novo leitor pode definir novos metadados, mas nunca contradizer uma outra configuração. Sendo assim, é importante definir qual a estratégia para lidar com conflitos e documentar isso bem para que fique claro aos usuários do componente como isso irá funcionar.

A seguir está a listagem que apresenta a classe `LeitorMetadadosComposto`. Essa classe implementa a interface `LeitorComparacao` e é composta por uma lista de leitores, permitindo que eles sejam adicionados através do método `adicionarLeitor()`. Dessa forma, o método `lerMetadados()` itera por esses leitores, invocando-os em sequência.

Leitor de metadados composto:

```
public class LeitorMetadadosComposto implements
 LeitorComparacao {

    private List<LeitorComparacao> leitores = new ArrayList<>();

    @Override
    public void lerMetadados(Class<?> clazz,
     ContainerComparacao cont) {
        for(LeitorComparacao leitor : leitores){
```

```
            leitor.lerMetadados(clazz, cont);
        }
    }
    public void adicionarLeitor(LeitorComparacao leitor){
        leitores.add(leitor);
    }
}
```

Por fim, a próxima listagem mostra como o leitor composto seria configurado para ler primeiro as anotações e em seguida verificar se existe algum metadado adicional no XML. Inicialmente, a instância é criada e os dois leitores são adicionados. Depois, o leitor composto é adicionado na classe `RepositorioMetadados` para ser utilizada por ele. Observe que, com essa solução, para quem utiliza o leitor é transparente o fato de que múltiplos leitores estão sendo utilizados.

Criando um novo ComparacaoPropriedade somente se ainda não existir:

```
LeitorMetadadosComposto composto =
    new LeitorMetadadosComposto();
composto.adicionarLeitor(new LeitorComparacaoAnotacoes());
composto.adicionarLeitor(new LeitorComparacaoXML());

RepositorioMetadados.getInstance().setLeitor(composto);
```

Aproveitando metadados de outros frameworks

Ao utilizar diversos frameworks e ferramentas em uma mesma aplicação, muitas vezes dá a impressão de que estamos repetindo a mesma configuração em diversos pontos do código. Imagine, por exemplo, uma situação bem comum em que uma determinada informação de texto possua um tamanho máximo. Esse tamanho é configurado no componente de interface gráfica para impedir o usuário de entrar com mais caracteres do que o permitido, no

código que valida as informações recebidas e até nos metadados da tabela para armazenar esse texto.

Com esse exemplo, é possível perceber como em um software uma mesma informação pode acabar ficando duplicada por todo o código-fonte. Com anotações e metadados em geral não é diferente, pois muitas vezes as informações de que precisamos já estão definidas nas anotações de outros frameworks. Em várias situações, os metadados podem não ser exatamente os mesmos, porém pode ser que a informação de que precisamos possa ser de alguma forma inferida.

No caso do componente de comparação, se pensarmos na API de persistência JPA, tem várias questões que podemos levar em consideração. Por exemplo, se uma propriedade é uma entidade persistente, então isso significa que se trata de um objeto composto no qual faz sentido fazer uma comparação profunda. Uma propriedade transiente também seria uma excelente candidata a ser ignorada no momento da transição.

Dessa forma, é possível adicionar na cadeia de leitura de metadados, um leitor capaz de recuperar os metadados de outros frameworks e adicionar no container de metadados as informações relevantes para seu domínio. Talvez um primeiro impulso seja ler as anotações diretamente da classe, porém isso pode não funcionar para frameworks com estratégicas mais elaboradas de leitura de metadados. No caso da JPA, por exemplo, pode não haver uma anotação na classe, mas a informação estar em um arquivo XML de configuração. Nesse caso, uma alternativa é buscar as informações diretamente no repositório de metadados, que no caso da JPA seria a classe `SessionFactory` de onde é possível

recuperar o container de metadados representado pela classe `ClassMetadata`.

A listagem a seguir apresenta a classe `LeitorMetadadosAdaptadorJPA`, a qual acessa o repositório de metadados da JPA e adapta a informação de quais propriedades são entidades para o contexto do componente de comparação. Observe que a primeira ação feita no método `lerMetadados()` é a recuperação da instância de `ClassMetadata` para a classe da qual se quer obter os metadados. Em seguida, é feita uma iteração nas sua propriedades e caso se trate de uma entidade, a característica `profundo` é configurada para `true` no container de metadados.

Buscando informações para comparação a partir da JPA:

```
public class LeitorMetadadosAdaptadorJPA implements
 LeitorComparacao {
    private SessionFactory sessionFactory;

    public LeitorMetadadosJPA(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    public void lerMetadados(Class<?> clazz,
     ContainerComparacao cont) {
        ClassMetadata metadata =
            sessionFactory.getClassMetadata(c);
        if (metadata != null) {
            for (String prop : metadata.getPropertyNames()) {
                if (metadata.getPropertyType(prop)
                    .isEntityType()) {
                    if(cont.getComparacaoPropriedade(prop)
                       != null){
                        ComparacaoPropriedade cp =
                            container.getComparacaoPropriedade(
                                prop);
                        cp.setProfundo(true);
                    }
```

```
                }
            }
        }
    }
}
```

Uma observação importante de ser feita é que dificilmente um adaptador de metadados poderia ser adicionado em um componente ou framework que não suporta a adição de múltiplos leitores. Isso porque dificilmente os metadados de outros frameworks serão suficientes, porém eles podem ser muito úteis para complementar os metadados nativos. Sendo assim, essa prática faz muito mais sentido em um componente com múltiplos leitores.

7.4 TORNANDO OS METADADOS EXTENSÍVEIS

Existem alguns domínios em que é muito difícil prever todas as possibilidades de configurações. Um exemplo que pode ser citado é o domínio de validação de instâncias. É impossível prever todas as possibilidades de formas de validação, porque existem diversas validações que são específicas de um determinado domínio. A validação de CPF e de outros números que possuem dígitos verificadores é um exemplo que é bastante específico. O componente de comparação é um outro exemplo desse tipo, pois pode haver também diferentes critérios de comparação também específicos do domínio. Duas coordenadas geográficas, por exemplo, poderiam ter uma distância mínima para serem consideradas diferentes.

Nesses domínios, não adianta tentar criar várias anotações e

prever todas as possibilidades, pois frequentemente uma aplicação irá precisar de algo que o componente não dá suporte. Nesses casos, é preciso permitir a extensão do esquema de metadados para que a aplicação possa adicionar novas classes ao componente para que seus requisitos sejam atendidos. Esta seção irá mostrar como essa prática pode ser implementada dentro de um componente baseado em metadados, iniciando com a extensão da leitura e, em seguida, com a extensão do comportamento.

Estendendo a leitura de metadados

O primeiro passo para que a extensão de metadados seja possível é estender a leitura. A ideia é permitir que novos elementos de metadados possam ser introduzidos e que, através de uma extensão no mecanismo de leitura, eles possam ser compreendidos pelo componente. Normalmente se utiliza uma estratégia similar à apresentada na seção *Estendendo o esquema de anotações*, criando uma anotação para configurar classes que sabem interpretar cada anotação. Vamos desenvolver essa extensão no exemplo do componente de comparação para anotações, porém é possível fazer algo similar para os metadados em XML. Basta possibilitar no documento a configuração de classes para processar novos elementos ou atributos.

A próxima listagem apresenta a interface `LeitorAnotacao` destinada a realizar a leitura de uma anotação de comparação. Esta interface recebe a anotação e a respectiva instância de `ComparacaoPropriedade`, que se refere à propriedade em que a anotação foi encontrada. A ideia é que cada anotação possua uma implementação dessa interface associada, que saiba interpretar seus dados e configurar as informações no container de

metadados.

Interface do leitor de uma anotação:

```
public interface LeitorAnotacao {
    public void lerMetadados(ComparacaoPropriedade cp,
        Annotation an);
}
```

A próxima listagem apresenta a anotação `@ConfiguracaoLeitor`, que deve ser adicionada nas novas anotações. Ela recebe no parâmetro `value` uma classe que implemente a interface `LeitorAnotacao`. Justamente esta classe é que será utilizada pelo componente para interpretar a própria anotação.

Anotação para configuração do respectivo leitor de metadados:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface ConfiguracaoLeitor {
    Class<? extends LeitorAnotacao> value();
}
```

A seguir, é apresentada a nova implementação do método `lerMetadados` da classe `LeitorComparacaoAnotacoes`. Observe que, fora a anotação `@NaoComparar`, não existe referência explícita a nenhuma outra anotação de comparação. Em cada propriedade, é feita uma busca em todas as anotações por aquelas que são anotadas com `@LConfiguracaoLeitor`. Quando uma é encontrada, a classe configurada na sua propriedade `value` é instanciada e utilizada para realizar sua leitura.

Nova implementação do leitor de anotações:

```
@Override
```

```
public void lerMetadados(Class<?> clazz,
 ContainerComparacao cont){
    for (Method m : clazz.getMethods()) {
        try {
            boolean isGetter = m.getName().startsWith("get");
            boolean semParam =
                (m.getParameterTypes().length == 0);
            boolean naoGetClass =
                !m.getName().equals("getClass");
            boolean paraComparar =
                !m.isAnnotationPresent(NaoComparar.class);
            if (isGetter && semParam && naoGetClass &&
                paraComparar) {
                ComparacaoPropriedade cp =
                    new ComparacaoPropriedade(m);
                for(Annotation an : m.getAnnotations()){
                    Class<? extends Annotation> anType =
                        an.annotationType();
                    if(anType.isAnnotationPresent(
                       ConfiguracaoLeitor.class)){
                        ConfiguracaoLeitor cl = anType.
                            getAnnotation(
                                ConfiguracaoLeitor.class);
                        LeitorAnotacao leitor =
                            cl.value().newInstance();
                        leitor.lerMetadados(cp, an);
                    }
                }
                cont.addComparacaoPropriedade(cp);
            }
        } catch (Exception e) {
            throw new RuntimeException(
                "Erro recuperando propriedade", e);
        }
    }
}
```

Para entender como essa solução seria utilizada na prática pelas anotações, as duas listagens a seguir mostram respectivamente o leitor desenvolvido para a anotação `@Tolerancia` seguido por sua definição. A classe `LeitorTolerancia` implementa a interface `LeitorAnotacao`, sendo que no método

`lerMetadados()` ela recebe a instância da anotação configurada na classe e popula o container de metadados com as informações da anotação. Na sua definição, é possível observar que `@ConfiguracaoLeitor` configura qual a classe que deve ser utilizada para a leitura.

Leitor da anotação de tolerância:

```
public class LeitorTolerancia implements LeitorAnotacao{
    @Override
    public void lerMetadados(ComparacaoPropriedade cp,
     Annotation anotacao) {
        Tolerancia tolerancia = (Tolerancia) anotacao;
        cp.setTolerancia(tolerancia.value());
    }
}
```

Anotação de tolerância com a configuração do leitor de metadados:

```
@ConfiguracaoLeitor(LeitorTolerancia.class)
@Retention(RetentionPolicy.RUNTIME)
public @interface Tolerancia {
    double value();
}
```

Associando lógica aos novos metadados

Apesar de a alteração realizada permitir a adição de novas anotações, ainda não é possível a adição de um novo tipo de lógica de comparação. Por exemplo, é possível adicionar uma anotação `@Peso` que possui uma tolerância predefinida, porém não é possível ter uma `@Substring` com um novo tipo de comparação. Esta seção irá mostrar como estruturar o componente para permitir a extensão também do seu comportamento.

Para que isso seja possível, o primeiro passo seria a criação de

uma interface para representar a lógica de comparação de uma propriedade. A próxima listagem apresenta a classe `LogicaComparacao` que possui o método `comparar()`, que recebe uma `String` com o nome da propriedade e o valor dos dois objetos comparados para fazer a comparação. Esse método retorna uma lista de diferenças como resultado, pois em caso de tipos mais complexos, como coleções por exemplo, mais de uma diferença pode ser encontrada entre os objetos.

Interface que define uma lógica de comparação específica:

```
public interface LogicaComparacao {
    public List<Diferenca> comparar(String prop, Object novo,
     Object velho);
}
```

O próximo passo é alterar o container de metadados para armazenar uma instância de `LogicaComparacao`. No caso, a classe alterada foi `ComparacaoPropriedade`, que representa os metadados de comparação de uma propriedade. Como mostra a listagem a seguir, foi adicionado o atributo `comparacao` que irá substituir atributos mais específicos de comparação. Por exemplo, informações como o valor da tolerância seriam armazenadas dentro da instância de `LogicaComparacao`. Outras informações sobre a comparação da propriedade, como se ela é profunda e o nome da propriedade, continuam na classe, apesar de serem omitidos nessa listagem.

Container de metadados composto com uma classe que define uma lógica de comparação:

```
public class ComparacaoPropriedade {

    private LogicaComparacao comparacao;
```

```
    public LogicaComparacao getComparacao() {
        return comparacao;
    }
    public void setComparacao(LogicaComparacao comparacao) {
        this.comparacao = comparacao;
    }
    //outras propriedades omitidas
}
```

A listagem a seguir mostra o trecho que precisaria ser modificado no método `comparar()` do componente de comparação. Na versão anterior, o primeiro condicional verificava se havia alguma tolerância configurada, e, em caso positivo, chamava um método da própria classe para fazer esse tipo de comparação. Nessa nova versão, o que é verificado é se alguma instância de `LogicaComparacao` foi configurada no container de metadados para aquela propriedade. Caso a instância não seja nula, a comparação é delegada para o seu método `comparar()` e a lista de diferenças retornadas para propriedade são adicionadas na lista do objeto.

Modificação no método comparar() do componente de comparação:

```
for (String prop : cont.getPropriedades()) {
    try {
        ComparacaoPropriedade cp =
            cont.getComparacaoPropriedade(prop);
        Method m = cp.getAcesso();
        Object valorVelho = m.invoke(velho);
        Object valorNovo = m.invoke(novo);
        if (cp.getComparacao() != null) {
            LogicaComparacao logica = cp.getComparacao();
            difs.addAll(logica.comparar(prop, valorNovo,
                        valorVelho));
        } else if (cp.isProfundo() && valorNovo != null &&
          valorVelho != null) {
            compararProfundo(difs, cp.getNome(), valorVelho,
            valorNovo);
```

```
        } else {
            compararNormal(difs, cp.getNome(), valorNovo,
                           valorVelho);
        }
    } catch (Exception e) {
        throw new RuntimeException(
                    "Erro recuperando propriedade", e);
    }
}
```

Para entender como isso funciona, esse sistema de metadados extensíveis será aplicado à anotação de tolerância. A próxima listagem mostra a classe `ComparacaoTolerancia`, que implementa a interface `LogicaComparacao` e possui a lógica de comparação de uma propriedade com tolerância. Observe que a informação da tolerância que deve ser utilizada é um atributo dessa classe.

Implementação da comparação de tolerância:

```
public class ComparacaoTolerancia implements LogicaComparacao {

    private double tolerancia;

    public ComparacaoTolerancia(double tolerancia) {
        this.tolerancia = tolerancia;
    }
    @Override
    public List<Diferenca> comparar(String prop, Object novo,
     Object velho) {
        List<Diferenca> difs = new ArrayList<>();
        double dif =
            Math.abs(((Double) novo) - ((Double) velho));
        if (dif > tolerancia) {
            difs.add(new Diferenca(prop, novo, velho));
        }
        return difs;
    }
    public double getTolerancia() {
        return tolerancia;
    }
```

```
}
```

Para que a classe de comparação correta seja colocada no container de metadados, é preciso modificar o respectivo leitor de metadados. A listagem a seguir mostra a classe `LeitorTolerancia`, que, como foi visto na seção anterior, é configurada na anotação `@Tolerancia`. Observe que, em vez de modificar o atributo `tolerancia`, agora ela cria uma instância da classe `ComparacaoTolerancia` e o coloca no container.

Modificação na implementação do leitor da anotação de tolerância:

```
public class LeitorTolerancia implements LeitorAnotacao{
    @Override
    public void lerMetadados(ComparacaoPropriedade cp,
     Annotation anotacao) {
        Tolerancia tolerancia = (Tolerancia) anotacao;
        cp.setComparacao(new ComparacaoTolerancia(
                                            tolerancia.value()));
    }
}
```

Dessa forma, é possível adicionar novas anotações de comparação sem a necessidade de alterar o código das classes existentes no framework. Para isso, basta criar a nova anotação, seu respectivo leitor, e a sua lógica de comparação. Assim, a anotação é ligada ao leitor através da anotação `@LeitorTolerancia` e ele deve criar e inserir a classe que implementa a lógica de comparação no container de metadados.

Estrutura geral para a extensão dos metadados

Depois de ver através do exemplo como os metadados podem ser estendidos, esta seção mostra uma solução mais geral para

aplicar essa solução a outros componentes. A figura a seguir apresenta a estrutura básica para a implementação da extensão. Ela mostra de uma forma mais geral quais classes devem estar presentes e como elas se relacionam.

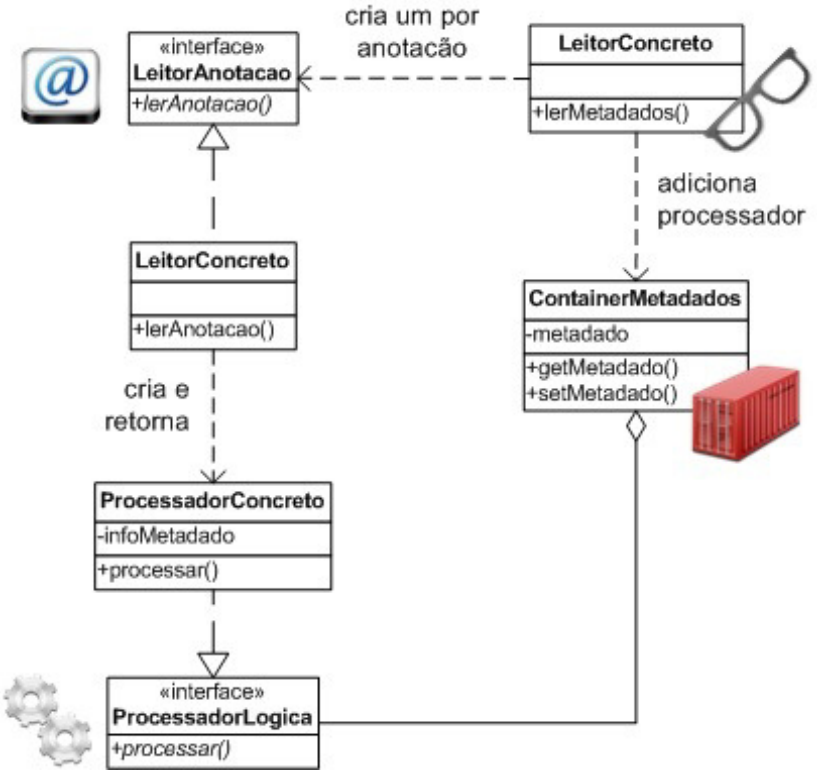

Figura 7.5: Estrutura geral para tornar os metadados extensíveis

Existem duas interfaces, representadas por `LeitorAnotacao` e `ProcessadorLogica`, que respectivamente representam abstrações para a leitura de uma anotação e para a execução de sua lógica no contexto do framework. Essas interfaces deverão ser implementadas cada vez que uma anotação for adicionada ao

framework. É importante ressaltar que o componente deve definir bem que informação deverá ser configurada pela anotação e que parte da lógica será delegada à classe que processa a sua lógica. No caso do componente de comparação foi a comparação entre duas propriedades, mas em outros contextos, por exemplo, poderia ser uma regra de segurança no domínio de controle de acesso ou uma condição para o acionamento do componente.

O leitor da anotação pode ser configurado utilizando a técnica mostrada na seção *Estendendo o esquema de anotações*, em que uma anotação configura a classe de leitura de uma outra anotação. Dessa forma, o leitor de metadados da classe deve procurar por anotações marcadas dessa forma, criando os respectivos leitores específicos de cada uma. A partir deles, os processadores são obtidos e adicionados ao container de metadados, que é retornado para a classe responsável pela lógica principal do componente. Este, por sua vez, deverá delegar uma parte bem definida de sua execução aos processadores que estão dentro do container de metadados.

O exemplo e a estrutura básica foram mostrados considerando que são utilizadas anotações para a definição de metadados, porém essa mesma estrutura também funciona para casos em que eles são definidos de uma outra forma. Por exemplo, no caso de documentos XML poderíamos ter leitores capazes de fazer a leitura de cada elemento. Dessa forma, ou no próprio XML ou em outro arquivo poderiam ser configuradas as classes que devem realizar a leitura de cada tipo de elemento.

Infelizmente, não é em todos os casos que essa técnica de extensão de metadados pode ser utilizada. Para isso, o componente

deve possuir um ponto de processamento independente para onde a execução poderia ser delegada a outra classe. Por exemplo, no caso do componente de comparação, a comparação de cada componente seria delegada a uma classe. No exemplo em que regras de segurança são configuradas por anotações, cada anotação poderia representar uma regra cuja execução seria delegada a uma classe. Infelizmente, nem todos os casos se encaixam nessa categoria.

Posso ter uma classe só para leitura e processamento da lógica?

Na estrutura que foi apresentada, existem duas classes que precisam ser criadas para agregar uma nova anotação no componente: uma para leitura e outra para o processamento da lógica. Em alguns componentes, essas duas responsabilidades são colocadas em uma mesma classe. Dessa forma, a leitura dos metadados serve apenas como forma de extrair as informações pertinentes dos metadados, colocando-as na própria instância que realiza a leitura. Esse método não precisa retornar nada ou mesmo adicionar algo no container, a própria lógica de criação desse leitor específico pode ser utilizada para isso.

Apesar de ser possível, existem algumas consequências negativas de se juntar essas responsabilidades em apenas uma classe. Uma delas é o acoplamento da anotação com a sua lógica de processamento. Caso, por exemplo, deseje-se fazer com que aquela configuração seja extraída de alguma outra forma, como de um documento XML, de metadados de outro framework ou mesmo de convenções de código, isso não será possível.

Outro problema é que isso amarra um relacionamento de um-para-um entre leitor e lógica de processamento. Algumas vezes é possível criar um processador mais geral que pode ser utilizado para várias anotações. Além disso, também é possível criar uma lógica de leitura genérica que seja possível de ser utilizada para várias anotações, mesmo gerando processadores diferentes como resultado. Ao colocar as duas responsabilidades na mesma classe, essa reutilização é dificultada.

O exemplo de um framework que possibilita a extensão de metadados e que utiliza uma classe só para leitura e processamento é o Hibernate Validator. Novas anotações de validação podem ser adicionadas a partir da criação de uma classe que as lê e possuem a respectiva lógica que validam os dados de uma propriedade. Porém, a única forma de adicionar metadados é a partir de anotações, não sendo possível sua adição a partir de outros meios. Isso porque a lógica que executa durante a validação está atrelada a leitura das anotações.

7.5 CAMADAS DE PROCESSAMENTO DE METADADOS

"Um adulto é uma criança com algumas camadas em cima." - Woody Harrelson

Em alguns casos, o componente que utiliza anotações possui diversas responsabilidades. O framework Hibernate, por exemplo, faz a validação das instâncias, chama métodos de callback (de pré-persistência e pós-recuperação) e realiza o acesso à base de dados. Por mais que seu domínio seja apenas um, as diversas tarefas que precisam ser executadas podem se acumular na classe central do

framework. Isso, além de tornar essa classe de difícil manutenção, também torna essa lógica principal pouco flexível, dificultando a adição de novas responsabilidades.

Uma forma de adicionar mais flexibilidade na lógica de negócios de um componente de processamento de metadados é a criação de camadas de processamento. Dessa maneira, parte da funcionalidade do componente é delegada para essas camadas. A partir dessa estrutura, é possível adicionar, remover e trocar a ordem de execução das camadas, tornando mais fácil a adaptação do seu comportamento.

Para apresentar essa prática, as próximas subseções irão apresentar como inserir camadas no componente de comparação. Inicialmente será mostrado como o componente irá receber o suporte a adição camadas e, em seguida, quais as camadas que serão criadas para atender os requisitos do componente. Por fim, essa solução será generalizada e serão feitas algumas considerações sobre a sua implementação em outros contextos.

Criação da estrutura

O primeiro passo para a inserção de camadas é definir que parte da execução que faria sentido ser delegada para as camadas. No caso do componente de comparação, é possível observar diversos condicionais no momento de comparar cada propriedade. Cada condicional é um passo na comparação e pode ser considerado uma camada na execução desse processo.

Para criar as camadas, é necessário definir uma interface para abstrair suas responsabilidades no contexto do componente. A listagem a seguir apresenta a classe `CamadaComparacao` , que

define o método `comparar()`, que faz uma comparação de valores de uma propriedade. Esse método recebe a lista de diferenças onde deve inserir as diferenças encontradas. O valor booleano retornado irá indicar se aquela camada foi necessária para a realização da comparação.

Interface para definição de uma camada de processamento:

```
public interface CamadaComparacao {

    public boolean comparar(List<Diferenca> difs,
            ComparacaoPropriedade cp,
            Object newValue, Object oldValue);
}
```

A seguir, a listagem mostra a classe `ComponenteComparacao` com a adição do suporte a adição de camadas. Observe que o atributo `camadas` armazena uma lista de camadas para serem invocadas na comparação. A classe agora disponibiliza um construtor default, que adiciona as camadas que fazem o que já era feito antes, cuja implementação será mostrada a seguir. Também é disponibilizado um construtor que recebe uma lista de camadas que pode ser especificada pela classe cliente.

Adição do suporte a camadas no componente de comparação:

```
public class ComponenteComparacao {

    private List<CamadaComparacao> camadas = new ArrayList<>();

    public void addCamada(CamadaComparacao camada){
        camadas.add(camada);
    }

    public ComponenteComparacao(){
        addCamada(new ComparacaoComNulo());
        addCamada(new ComparacaoProfunda(this));
```

```
        addCamada(new ComparacaoEspecifica());
        addCamada(new ComparacaoEquals());
    }

    public ComponenteComparacao(List<CamadaComparacao> camadas){
        this.camadas.addAll(camadas);
    }

    //método de comparação omitido
}
```

Por fim, a próxima listagem mostra o método de comparação refatorado para a utilização das camadas. Observe que os condicionais que verificavam o tipo de comparação foram removidos e deram lugar a uma iteração que invoca o método `comparar()` nas camadas. Os métodos auxiliares também foram removidos dessa classe, sendo a base para a implementação das camadas. As camadas vão sendo percorridas até que uma delas retorne `true`, indicando que a comparação já foi realizada. Uma forma alternativa de realizar essa implementação seria a utilização do padrão Chain of Responsibility, delegando a própria camada de processamento a responsabilidade de invocar a próxima camada se necessário.

Alteração do método de comparação para invocação das camadas:

```
public <E> List<Diferenca> comparar(E velho, E novo){
    List<Diferenca> difs = new ArrayList<Diferenca>();
    Class<?> clazz = novo.getClass();
    ContainerComparacao cont = RepositorioMetadados.
            getInstance().getMetadados(clazz);
    for (String prop : cont.getPropriedades()) {
        try {
            ComparacaoPropriedade cp =
                cont.getComparacaoPropriedade(prop);
            Method m = cp.getAcesso();
            Object valorVelho = m.invoke(velho);
```

```
            Object valorNovo = m.invoke(novo);
            boolean comparado = false;
            for(int i=0; i<camadas.size() && !comparado ; i++){
                comparado = camadas.get(i).comparar(difs, cp,
                                        valorNovo, valorVelho);
            }
        } catch (Exception e) {
            throw new RuntimeException(
                            "Erro recuperando propriedade", e);
        }
    }
    return difs;
}
```

Implementação das camadas

Essa seção irá mostrar a implementação de quatro camadas que implementam as funcionalidades de comparação que o componente já possuía. A primeira camada realiza a comparação quando um dos valores é nulo e está implementada na classe `ComparacaoComNulo` na listagem a seguir. O código verifica se um dos valores é nulo e adiciona a diferença caso um seja e o outro não. Se ambos forem nulos, nenhuma diferença será registrada e será retornado `true` indicando que a comparação foi realizada. Já se nenhum dos valores for nulo, é retornado `false` indicando que essa camada não foi capaz de realizar a comparação.

Camada de comparação com valores nulos:

```
public class ComparacaoComNulo implements CamadaComparacao {

    public boolean comparar(List<Diferenca> difs,
     ComparacaoPropriedade cp, Object newValue, Object oldValue)
     {
        if (newValue == null) {
            if (oldValue != null) {
                difs.add(new Diferenca(cp.getNome(), newValue,
                        oldValue));
            }
```

```
            return true;
        } else if(oldValue == null){
            difs.add(new Diferenca(cp.getNome(), newValue,
                    oldValue));
            return true;
        }
        return false;
    }
}
```

A próxima camada apresentada é a que realiza a comparação profunda, em que é feita a comparação das propriedades dos valores. Uma questão importante é que essa camada precisa ter acesso ao próprio componente de comparação para poder invocá-lo de forma recursiva. Dessa forma, observe que o componente é recebido no construtor da classe e armazenado em um atributo. Veja também na nova implementação da classe `ComponenteComparacao` que, no construtor em que as camadas são criadas, a classe `ComparacaoProfunda` recebe `this` como parâmetro em seu construtor.

Implementação da camada com comparação profunda:

```
public class ComparacaoProfunda implements CamadaComparacao {

    private ComponenteComparacao componente;

    public ComparacaoProfunda(ComponenteComparacao componente) {
        this.componente = componente;
    }
    public boolean comparar(List<Diferenca> difs,
     ComparacaoPropriedade cp, Object newValue, Object oldValue)
     {
        if(cp.isProfundo()){
        List<Diferenca> difsProp =
            componente.comparar(newValue, oldValue);
            for (Diferenca d : difsProp) {
                d.setPropriedade(cp.getNome() + "." +
                                 d.getPropriedade());
                difs.add(d);
```

```
            }
            return true;
        }
        return false;
    }
}
```

As próximas duas listagens mostram respectivamente as classes `ComparacaoEspecifica`, que faz a comparação quando o container de metadados possui uma instância de `LogicaComparacao`, e `ComparacaoEquals`, que, caso nenhuma outra camada tenha concluído a comparação, utiliza o método `equals()` para verificar se os valores são iguais. Observe que a última camada sempre irá retornar `true`, significando que sempre será possível fazer a comparação dessa forma.

Comparação com anotações específicas:

```
public class ComparacaoEspecifica implements CamadaComparacao{

    public boolean comparar(List<Diferenca> difs,
     ComparacaoPropriedade cp, Object newValue, Object oldValue)
     {
        if (cp.getComparacao() != null) {
            LogicaComparacao logica = cp.getComparacao();
            difs.addAll(logica.comparar(cp.getNome(), newValue,
                        oldValue));
            return true;
        }
        return false;
    }
}
```

Modificação na implementação do leitor da anotação de tolerância:

```
public class ComparacaoEquals implements CamadaComparacao {

    public boolean comparar(List<Diferenca> difs,
     ComparacaoPropriedade cp, Object newValue, Object oldValue)
```

```
    {
        if (!newValue.equals(oldValue)) {
            difs.add(new Diferenca(cp.getNome(), newValue,
                                   oldValue));
        }
        return true;
    }
}
```

Estrutura geral de camadas

A maior dificuldade na introdução dessa prática em um componente é descobrir como seu comportamento pode ser particionado e em que ponto isso é viável de ser realizado. No caso do componente de comparação, foi na comparação entre propriedades e não na comparação completa da classe. É preciso definir exatamente em que ponto o processamento pode ser delegado para as camadas, o que nem sempre é trivial, ou mesmo possível em alguns casos.

A figura seguinte apresenta a estrutura que pode ser utilizada para as camadas de processamento. O componente principal será composto por aquelas para as quais ele deve delegar parte de seu processamento. Cada camada irá acessar o container de metadados para acessar as informações relevantes para seu processamento. O container de metadados pode ser passado camada a camada pelo componente principal, ou pode ser acessado diretamente pelas camadas no repositório, como mostrado no diagrama.

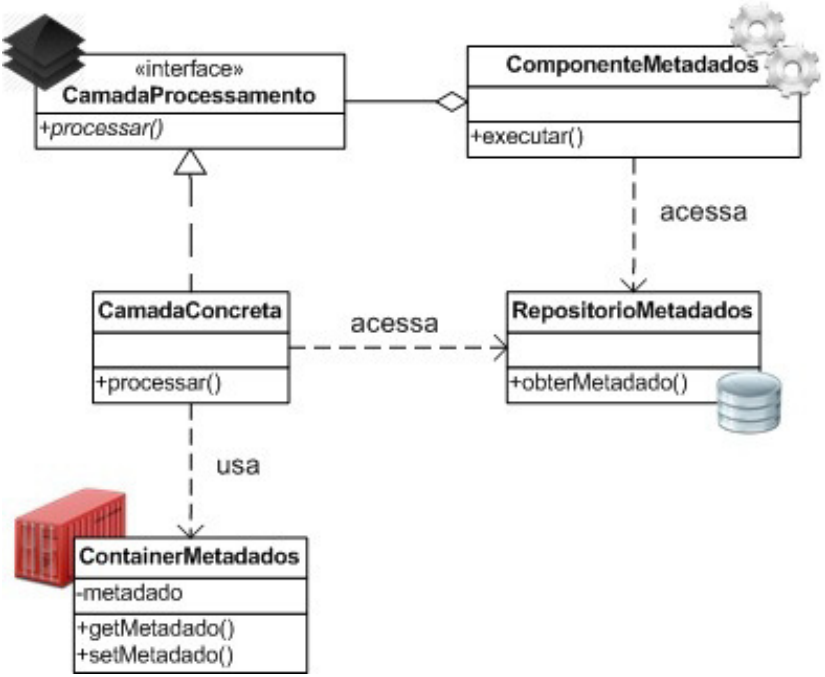

Figura 7.6: Inserindo camadas de processamento de metadados

A utilização do repositório não é obrigatória para implementação dessa prática. Como pode ser visto, o próprio componente de comparação não utilizou esse recurso, sendo um dos motivos o fato dele precisar somente da parte do metadado relativa a uma propriedade. Com o uso de camadas, o processamento dos metadados acaba ficando um pouco disperso e existe essa preocupação de passar o container para todas as camadas. Nesse caso, o uso do repositório eliminaria essa preocupação e deixaria os metadados acessíveis de qualquer ponto do componente.

7.6 CONSIDERAÇÕES FINAIS

Neste capítulo, foram apresentadas práticas para a estruturação de um componente que consome metadados. Essas práticas incluíram a separação entre a leitura e processamento, técnicas para tornar a leitura de metadados mais flexível, uma estrutura para permitir a extensão de metadados e, por fim, a criação de camadas de processamento. Essas práticas foram extraídas de diversos frameworks utilizados largamente na indústria, dando confiança de que realmente irão funcionar na prática.

Foi utilizado um *running example* no decorrer do capítulo para ilustrar a aplicação de cada prática apresentada. Observando a primeira versão do componente de comparação apresentada e o resultado final após ele ser refatorado diversas vezes, é possível perceber que houve um imenso ganho de flexibilidade. Essa flexibilidade permite que os mecanismos de leitura e processamento de metadados sejam estendidos e alterados em diversos pontos.

Por fim, é importante ressaltar que o fato de essa prática estar aqui documentada não significa que ela deva ser utilizada sempre em todos os frameworks que utilizem metadados. Antes de adicionar qualquer tipo de flexibilidade, que acaba tornando a estrutura como um todo mais complexa, avalie o porquê de a prática estar sendo aplicada e se realmente vale a pena.

Tópicos complementares

A primeira parte do livro mostrou os principais conceitos e as APIs básicas de reflexão, e a segunda parte apresentou boas práticas para serem seguidas na utilização desses conceitos. Se você está se perguntando se isso que viu até agora não é o suficiente, eu posso responder que isso é um bom começo, mas é o que vem agora que vai diferenciá-lo dos outros desenvolvedores. Se muitos acham que usar reflexão em Java é um tipo de "magia", é nesta parte do livro vamos ver o que existe de mais "sombrio"...

Isso não significa que o conteúdo que será apresentado é extremamente complicado e que apenas alguns vão conseguir entender. A grande dificuldade do que vem nesta última parte do livro tem a ver com os prerrequisitos necessários para a compreender, que são justamente o que você já aprendeu até agora. Então relaxe, que nos próximos capítulos você vai conhecer aqueles "pulos do gato" que poucos sabem que são possíveis!

No capítulo *Ferramentas: indo além na reflexão* são abordadas ferramentas livres desenvolvidas por terceiros que ajudam na utilização da reflexão e que, muitas vezes, permitem a criação de funcionalidades muito complicadas de serem desenvolvidas apenas com a API disponível na JDK. Em seguida, o capítulo *Manipulação de bytecode* aborda a técnica da manipulação de bytecode, na qual o desenvolvedor tem a possibilidade de criar novas classes e modificar classes existentes. As novidades em relação à API de reflexão para o Java 8 são abordadas no capítulo *Reflexão no Java 8*, no qual se fala sobre o acesso a nomes de parâmetros, anotações

repetidas e anotações de tipo. Por fim, o livro finaliza com o capítulo *Truques da API de reflexão*, que apresenta algumas questões menos conhecidas que se podem fazer com a API de reflexão, que irá complementar e consolidar o conhecimento do leitor sobre o assunto.

Capítulo 8

FERRAMENTAS: INDO ALÉM NA REFLEXÃO

"Se a única ferramenta que você tem é um martelo, você tende a ver todo problema como um prego." - Abraham Maslow

Uma das grandes vantagens de se trabalhar com a linguagem Java não tem nada a ver com a tecnologia, mas com a comunidade. Existe uma infinidade de frameworks e ferramentas livres e/ou de código aberto de excelente qualidade que podem ser utilizadas pelos desenvolvedores. Isso não poderia ser diferente dentro do contexto da reflexão. Também existem diversas bibliotecas e ferramentas que podem ser utilizadas para facilitar o trabalho com reflexão.

Alguns capítulos anteriores acabaram falando com mais detalhes de algumas ferramentas dentro de cada contexto. No capítulo *Proxy dinâmico*, foi mostrado como o CGLib pode ser utilizado para a criação de proxies dinâmicos. Este capítulo irá falar um pouco mais das funcionalidades de reflexão dessa biblioteca. Outro capítulo que falou sobre várias outras ferramentas foi o *Testando classes que usam Reflexão*, que mostrou ferramentas mais gerais de teste como o JUnit e o JMock, mas que também falou do ClassMock que é específico para testes de classes

que trabalham com reflexão e anotações.

Realmente seria impossível abordar todos os projetos livres relacionadas com reflexão nesse livro, porém selecionei três que considero muito interessantes de estarem na caixa de ferramentas para quem trabalha com reflexão. Serão mostradas ferramentas que simplificam ou otimizam tarefas corriqueiras, e outros que implementam funcionalidades que seriam bem difíceis de serem implementadas diretamente.

8.1 MANIPULANDO BEANS COM BEANUTILS

Com reflexão, é possível manipular qualquer tipo de classe, mas pelos próprios exemplos do livro pode-se perceber que em grande parte das vezes as classes utilizam reflexão para acessar JavaBeans (que por simplicidade chamaremos apenas de bean). Por exemplo, acessar uma propriedade através de seu método getter ou alterar seu valor através do seu método setter são tarefas recorrentes nesse tipo de componente. Apesar de não serem tarefas complexas, as coisas podem se complicar um pouco quando precisamos acessar propriedades aninhadas ou converter valores de acordo com o tipo das propriedades.

A Apache Software Foundation é uma comunidade de desenvolvedores que desenvolve diversos projetos livres, principalmente para a comunidade Java. O projeto **Apache Commons** reúne diversos componentes que contêm aspectos gerais que podem ser reutilizados em diversas aplicações e em diversos projetos. O **BeanUtils** é um desses componentes que encapsulam a utilização da API de reflexão, criando classes mais fáceis de serem utilizadas para funcionalidades comuns de

JavaBeans. Esta seção irá mostrar as principais funcionalidades dessa biblioteca e como ela pode auxiliá-lo a interagir com esse tipo de classe. Nos exemplos foi utilizada a versão 1.9.1 dessa biblioteca.

Acesso a propriedades de beans

A primeira coisa que se precisa ao acessar um bean é ter acesso as suas propriedades através dos métodos de acesso. A grande vantagem do BeanUtils é que ele provê uma forma fácil de acessar não só as propriedades simples, mas também propriedades aninhadas, indexadas em listas ou arrays, ou em mapas. A listagem a seguir apresenta o bean `Pessoa`, que será utilizado para o exemplo. Observe que essa classe possui propriedades do tipo array, do tipo `Map` e do tipo `Endereco`, que também é um bean e tem suas próprias propriedades.

JavaBean utilizado no exemplo:

```
public class Pessoa {

    private String nome;
    private int idade;
    private Endereco endereco = new Endereco();
    private String[] interesses = new String[10];
    private Map<String, Integer> notas = new HashMap<>();

    //métodos getters e setters omitidos
}
```

A próxima listagem apresenta um exemplo que mostra como é possível inserir e recuperar valores em diferentes propriedades de um bean. Ressalto que os métodos utilizados no exemplo são da classe `PropertyUtils` e importados utilizando um `static import`. O método básico de acesso a propriedades é o

`getProperty()`, que recebe como parâmetro o bean e o nome da propriedade. De forma complementar, o método `setProperty()` configura o valor de uma propriedade. Ele recebe os mesmos parâmetros e mais um adicional que é o valor que deve ser adicionado na propriedade.

Exemplo de referência a propriedades do JavaBean:

```
import static org.apache.commons.beanutils.PropertyUtils.*;

public class AcessoPropriedades {

    public static void main(String[] args) throws Exception {
        Pessoa p = new Pessoa();
        setProperty(p, "nome", "Fulano");
        setProperty(p, "endereco.rua", "Rua 21 de Abril");
        setProperty(p, "interesses[0]", "quadrinhos");
        setIndexedProperty(p, "interesses", 1, "RPG");
        setProperty(p, "notas(português)", 10);
        setMappedProperty(p, "notas", "matemática", 10);

        String nome = (String) getProperty(p, "nome");
        System.out.println("Nome: " + nome);

        String rua = (String) getProperty(p, "endereco.rua");
        System.out.println("Rua: " + rua);

        String i1 = (String) getProperty(p, "interesses[0]");
        String i2 =
            (String) getIndexedProperty(p, "interesses", 1);
        System.out.println("Interesses: "+i1+"; "+i2);

        Integer n1 =
            (Integer) getProperty(p, "notas(português)");
        System.out.println("Nota português = "+n1);
        Integer n2 =
            (Integer) getMappedProperty(p,"notas","matemática");
        System.out.println("Nota matemática = "+n2);
    }
}
```

Observe, pelo exemplo, que os métodos `getProperty()` e

`setProperty()` aceitam na `String` que define o nome da propriedade definições mais complexas. Por exemplo, pode-se navegar em propriedades aninhadas utilizando um ponto, como em `endereco.rua` . Adicionalmente, é possível também acessar propriedades indexadas, como listas e arrays, e propriedades em mapas, utilizando respectivamente colchetes, como em `interesses[0]` , e parênteses, como em `notas(português)` . Apesar de não ser demonstrado no exemplo, é possível utilizar esses recursos de forma combinada, como `produtos[1].info(descricao).voltagem` .

Além dos métodos gerais, existem também métodos mais específicos para o acesso de propriedades. Alguns que não são mostrados no exemplo são `getSimpleProperty()` e `setSimpleProperty()` para propriedades simples, e `getNestedProperty()` e `setNestedProperty()` para propriedades aninhadas. Para propriedades indexadas, como arrays e listas, existem os métodos `getIndexedProperty()` e `setIndexedProperty()` que recebem o nome simples da propriedade e um parâmetro adicional com o índice. De forma similar, os métodos `getMappedProperty()` e `setMappedProperty()` para propriedades mapeadas também recebem um parâmetro adicional com a chave do mapa.

Vale ressaltar que a classe `PropertyUtils` ainda possui alguns métodos interessantes de serem utilizados. Os métodos `isReadable()` e `isWriteable()` verificam respectivamente se uma dada propriedade possui método getter para recuperação e setter para atribuição de valor. O método `describe()` recebe o bean como parâmetro e retorna um mapa com suas propriedades, de forma similar ao que foi desenvolvido na seção *O primeiro*

contato com a API Reflection, mas sem suporte a anotações. Outro método interessante é o `copyProperties()`, que copia todas as propriedades que possuem o mesmo nome de um objeto para outro, mesmo que a classe não seja a mesma.

Lidando com conversão de valores

Uma questão que precisa ser levada em consideração quando estamos configurando valores de propriedades é que o dado que está sendo inserido tenha o tipo da propriedade. Isso acaba sendo um problema, principalmente quando os dados estão sendo lidos de uma fonte que os fornece como `String`, como a leitura de um arquivo texto ou parâmetros recebidos em uma aplicação web. Se passarmos uma `String` como valor de uma propriedade que não possui esse tipo nos métodos da classe `PropertyUtils`, será lançada uma exceção dizendo que o tipo não é compatível.

Felizmente, o BeanUtils fornece uma outra classe chamada `BeanUtils`, que possui métodos similares, porém, que recebem e retornam sempre uma `String` como valor da propriedade. A listagem a seguir mostra um exemplo que utiliza o `BeanUtils`. Observe que a propriedade `idade` recebe o valor como uma `String` mesmo sendo do tipo `Integer`. É importante ressaltar que, nesse caso, se o mesmo método de `PropertyUtils` fosse utilizado, um erro indicando a incompatibilidade de tipos seria lançado. Em seguida, a propriedade é recuperada utilizando o `BeanUtils` e o `PropertyUtils`, e o valor retornado será respectivamente dos tipos `String` e `Integer`.

Utilizando o BeanUtils para a conversão de valores:

```
Pessoa p = new Pessoa();
```

```
BeanUtils.setProperty(p, "idade", "23");

System.out.println(BeanUtils.getProperty(p, "idade")
                   .getClass());
System.out.println(PropertyUtils.getProperty(p, "idade")
                   .getClass());
```

O BeanUtils provê conversores de `String` para todos os tipos primitivos e suas respectivas classes wrappers. Porém, é possível criar conversores para outras classes, para que possam ser utilizados pelos métodos da classe `BeanUtils`. Essa classe precisa implementar a interface `Converter` e o método `convert()`. A classe apresentada na próxima listagem mostra o exemplo de um conversor para a classe `Endereco`, que recebe uma `String` com o nome da rua e o número separados por uma vírgula.

JavaBean utilizado no exemplo:

```
public class EnderecoConverter implements Converter {

    public Object convert(Class c, Object v) {
        String[] info = v.toString().split(",");
        Endereco e = new Endereco();
        e.setRua(info[0].trim());
        e.setNumero(info[1].trim());
        return e;
    }
}
```

Para que a conversão possa ser feita, é preciso registrar o novo conversor criado para a classe para a qual ele deve converter. Isso pode ser feito chamando o método `register()` da classe `ConvertUtils` passando como parâmetro a instância do conversor e a respectiva classe, como exemplificado na listagem a seguir. Observe que, depois, a propriedade pode ser configurada utilizando-se a `String` no formato adequado e, em seguida, a propriedade configurada pode ser recuperada normalmente.

JavaBean utilizado no exemplo:

```
ConvertUtils.register(new EnderecoConverter(), Endereco.class);

Pessoa p = new Pessoa();
BeanUtils.setProperty(p, "endereco", "Rua do Limoeiro, 25");

System.out.println(PropertyUtils.getProperty(p,"endereco.rua"));
System.out.println(PropertyUtils.getProperty(p,
                                             "endereco.numero"));
```

Trabalhando com coleções de beans

Além dessas funcionalidades para trabalhar com beans individuais, o BeanUtils também provê classes que permitem fazer operações em coleções de beans. Elas implementam a interface `Closure` do Commons Collections. A seguir estão apresentadas algumas das classes providas pelo BeanUtils e como elas podem ser utilizadas:

- `BeanPropertyValueChangeClosure` é uma classe utilizada para alterar uma propriedade em uma coleção de beans. Ela recebe como parâmetro em seu construtor o nome da propriedade e o valor a ser configurado. Ela deve ser passada como parâmetro, junto com a coleção desejada, para o método `forAllDo()` da classe `CollectionsUtils` para que a operação seja executada.
- `BeanPropertyValueEqualsPredicate` é uma classe utilizada para filtrar uma coleção, selecionando somente os objetos que possuírem uma propriedade igual a um valor. Ela recebe como parâmetro em seu construtor o nome da propriedade e o valor que serão utilizados na filtragem. Ela deve ser passada como

parâmetro, junto com a coleção desejada, para o método `filter()` da classe `CollectionsUtils` para que a filtragem seja realizada.

- `BeanToPropertyValueTransformer` é uma classe que cria uma nova coleção com uma das propriedades do bean. Por exemplo, de uma lista de objetos da classe `Pessoa` seria possível obter uma lista com o valor do atributo `email`. Ela recebe como parâmetro no construtor o nome da propriedade a partir da qual se deseja criar a nova coleção. Ela deve ser passada como parâmetro, junto com a coleção desejada, para o método `collect()` da classe `CollectionsUtils` para que a coleção com os valores da propriedade seja retornada.

A listagem a seguir exemplifica a utilização de cada uma das classes apresentadas a seguir. Se considerarmos a quantidade de possibilidades para filtragem e atualização de coleções, as classes fornecidas pelo BeanUtils são bem limitadas. Porém, a partir do modelo adotado por elas, é possível perceber que não é difícil desenvolver outras similares com comportamentos diferentes.

Lidando com coleções de beans:

```
// Configurando propriedade em vários beans
BeanPropertyValueChangeClosure closure =
    new BeanPropertyValueChangeClosure("processamento",
        Boolean.TRUE);
CollectionUtils.forAllDo(pessoas, closure);

// Filtrando lista por propriedade
BeanPropertyValueEqualsPredicate predicado =
    new BeanPropertyValueEqualsPredicate("processamento",
        Boolean.FALSE);
CollectionUtils.filter(pessoas, predicado);
```

```
// Transformando em coleção
BeanToPropertyValueTransformer transformador =
    new BeanToPropertyValueTransformer("nome");
Collection nomes =
    CollectionUtils.collect(pessoas, transformador);
```

Observando as funcionalidades do `BeanUtils`, é possível perceber que não são difíceis de serem implementadas com o que vimos no livro até agora. A grande utilidade dessa biblioteca é que essas são funcionalidades que utilizamos em diversos contextos e com frequência. Dessa forma, poupando o tempo de implementação dessas pequenas tarefas mais gerais, é possível se concentrar em questões mais importantes. Além disso, o BeanUtils já trata de diversas questões que são mais trabalhosas de implementar, como a conversão de parâmetros para `String` e o fato do "is" poder ser ser utilizado no lugar do "get" em propriedades booleanas.

8.2 INTERFACES FLUENTES PARA REFLEXÃO

A API Reflection é a alternativa provida pela JDK para se trabalhar com as metainformações das classes, porém, algumas pessoas discordam de que essa seja a melhor abordagem para a criação de uma API para esse fim. Sendo assim, existem diversos projetos de código aberto que encapsulam a API tradicional de reflexão e proveem uma outra alternativa. Essa alternativa utiliza o conceito de interface fluente (ver quadro), como sendo uma forma de utilizar as funcionalidades da reflexão fornecendo um código mais legível e mais simples. Exemplos de projetos que implementam esse tipo de interface é o FEST-Reflect (http://fest.easytesting.org/reflect/), o JOOR

(https://code.google.com/p/joor/) e o Mirror (http://projetos.vidageek.net/mirror/mirror/).

> **Interface fluente**
>
> O termo **interface fluente** foi cunhado por Martin Fowler e Erick Evans (FOWLER, 2005) como uma forma de descrever um estilo de construção de interfaces. A ideia é dar o nome dos métodos da classe de forma que o código pareça uma frase em linguagem natural. A listagem a seguir mostra o exemplo de uma interface tradicional, utilizando métodos começados com `get` e `set`, e a mesma classe utilizando uma interface fluente. Com uma interface fluente é como se você estivesse definindo uma nova linguagem dentro da linguagem de programação a partir dos nomes dos métodos, prática que também é chamada de DSL (Linguagem Específica de Domínio) interna (FOWLER, 2010).

Para demonstrar a utilização desse tipo de API, vamos utilizar o Mirror, que é um projeto que nasceu aqui no Brasil! A próxima listagem mostra como essa API seria utilizada para criar um método que imprime as propriedades de um objeto pelos seus métodos getters. Inicialmente, a classe `Mirror` é utilizada na recuperação dos métodos getters. Observe que, após a criação de sua instância, os métodos vão sendo chamados em sequência de forma que a cada passo se define uma parte do que se deseja. O método `getters()` foi utilizado para se obter todos os métodos getters da classe, mas o método `matching()` poderia ser chamado

recebendo uma instância de uma classe que poderia selecionar os métodos a partir de diferentes critérios. Observe, em seguida, que de forma similar é feita a invocação dos métodos.

Exemplo de utilização do Mirror:

```
public void imprimirPropriedades(Object obj) throws Exception{
    Class clazz = obj.getClass();
    List<Method> l =
        new Mirror().on(clazz).reflectAll().getters();
    for(Method m : l){
        Object valor = new Mirror().on(obj).invoke().method(m)
                        .withoutArgs();
        System.out.println(valor);
    }
}
```

A utilização de uma API desse tipo vai muito da preferência de cada um. Os conceitos são os mesmos que foram apresentados neste livro, porém são acessados de uma forma diferente. Existem também algumas funcionalidades interessantes que não estão disponíveis na API de reflexão, como, no caso do Mirror, a seleção de métodos a partir de variados critérios e a recuperação de métodos a partir apenas de seu nome (contando que não exista sobrecarga). Acho importante que os desenvolvedores saibam que essas alternativas existem e que são ferramentas que podem simplificar o seu trabalho e deixar o código mais legível.

8.3 PROCURANDO POR CLASSES

"Quem procura, acha." - ditado popular

Um dos problemas que temos quando utilizamos interfaces ou anotações para marcarmos classes de um determinado tipo é encontrar essas classes. Por exemplo, o framework web VRaptor

(CAVALCANTI, 2013) faz essa busca de classes na inicialização da aplicação buscando as que são anotadas com `@Resource`. Muitas vezes, não basta essa marcação, sendo ainda necessário configurar em arquivos externos, ou mesmo através de código, as classes cujos metadados serão lidos pelo framework. Apesar de não haver nenhuma API nativa que as busque no classpath a partir de suas características, existem algumas alternativas interessantes no mundo open source. Esta seção irá apresentar duas alternativas que podem ser utilizadas.

Buscando classes anotadas com Scannotation

O Scannotation é uma biblioteca de classes Java que cria uma base de informações de anotações a partir de um conjunto de arquivos com extensão `.class`. Ela cria uma base de informações, que fica registrada na classe `AnnotationDB`, sobre quais classes utilizam quais anotações e sobre quais anotações são utilizadas por uma determinada classe. Uma característica que torna interessante o uso do Scannotation é que isso é feito sem carregar cada uma delas. Por esse motivo, a base de informações utiliza mapas que representam as classes e anotações utilizando instâncias de `String` com seus nomes, e não instâncias de `Class`.

O primeiro passo para fazer a busca de anotações é declarar quais arquivos ou em qual parte do classpath deseja-se realizar a pesquisa. Para isso, deve-se utilizar a classe `ClasspathUrlFinder`, que possui diversos métodos que possibilitam obter as URLs para restringir a busca. De forma similar, a classe `WarUrlFinder` provê formas de realizar essa busca dentro do diretório `lib` de arquivos de extensão `.war`,

utilizados para o deploy de aplicações web.

A listagem a seguir mostra como o Scannotation pode ser utilizado para buscar classes e anotações. Nesse exemplo, como desejávamos fazer a busca em todo classpath, o método `findClassPaths()` foi utilizado para obter as URLs onde a busca deveria ser realizada. Em seguida, é criada uma instância de `AnnotationDB`, cujo método `scanArchives()` foi invocado passando as URLs obtidas como parâmetro. Essa chamada será responsável por popular a base de informações.

Procurando classes e anotações com Scannotation:

```
@Componente
public class ScannotationExemplo {

    public static void main(String[] args) throws IOException {
        URL[] urls = ClasspathUrlFinder.findClassPaths();
        AnnotationDB db = new AnnotationDB();
        db.scanArchives(urls);

        //Classes com a anotação @Componente
        Map<String, Set<String>> annotationIndex =
            db.getAnnotationIndex();
        Set<String> componente =
            annotationIndex.get("Componente");
        System.out.println(componente);

        //Anotações da classe ScannotationExemplo
        Map<String,Set<String>> classIndex = db.getClassIndex();
        Set<String> scannotationExemplo =
            classIndex.get("ScannotationExemplo");
        System.out.println(scannotationExemplo);
    }
}
```

Depois que isso for feito, o método `getAnnotationIndex()` pode ser utilizado para obter um mapa que possui as anotações como chave e um `Set` com as classes que a possuem como valor.

De forma inversa, o método `getClassIndex()` retorna um mapa cujas chaves são as classes e os valores `Set` com suas respectivas anotações. Observe que os mapas utilizam `String` para representar as classes, justamente porque elas não são carregadas para que a busca possa ser feita. Caso isso seja necessário, é preciso fazer a chamada `Class.forName()` para obter a respectiva instância de `Class`.

Vale ressaltar que, na busca realizada por essa biblioteca, não são consideradas somente anotações utilizadas nas classes, mas também em métodos, atributos e outros membros da classe. Porém, apesar de esse ser o comportamento default, é possível configurar na classe `AnnotationDB` em que elementos se deseja que a busca seja realizada. Por exemplo, para habilitar e desabilitar a busca em atributos, o método `setScanFieldAnnotations()` pode ser invocado recebendo um valor booleano como parâmetro. Existem métodos similares para anotações de classe, método e parâmetro. Também é possível restringir os pacotes onde a busca será realizada, através do método `setIgnoredPackages()`, para o qual um array com os pacotes a serem ignorados pode ser passado.

Uma outra funcionalidade interessante dessa biblioteca é a possibilidade de estender essa busca às anotações das interfaces implementadas pela classe e às anotações que anotam as anotações da classe, como mostrado na prática de mapeamento de anotações da seção *Mapeamento de anotações*. Os métodos `crossReferenceImplementedInterfaces()` e `crossReferenceMetaAnnotations()` da classe `AnnotationDB` podem ser chamados depois da chamada de `scanArchives()` para que os mapas considerem essas referências indiretas.

Extensible Component Scanner

Uma outra alternativa para a busca de classes com anotações é um projeto chamado **Extensible Component Scanner**. Esse projeto possui uma API mais amigável que o Scannotation, lidando diretamente com instâncias de `Class` em vez de `String`. Porém, diferentemente do componente apresentado na seção anterior, cuja busca é restrita através dos arquivos, aqui a busca é restrita através do pacote em que a classe se encontra.

A classe principal para a busca de componentes é a `ComponentScanner`. Ela possui o método `getClasses()` que é utilizado para a busca de classes e recebe como parâmetro uma instância do tipo `ComponentQuery`, que normalmente é implementada como uma classe anônima. A próxima listagem mostra a estrutura básica para a utilização do `ComponentScanner`. Observe que a classe abstrata `ComponentQuery` é estendida no próprio código, onde o método `query()` deve ser implementado para a execução da busca.

Estrutura básica de uso do ComponentScanner:

```
ComponentScanner scanner = new ComponentScanner();
Set<Class<?>> classes = scanner.getClasses(new ComponentQuery()
{
    protected void query() {
        //criação da query
    }
});
```

Na execução desse método, a busca será feita nas classes do classloader default, que pode ser recuperado pela chamada `Thread.currentThread().getContextClassLoader()`. Porém, esse método é sobrecarregado para possibilitar que o classloader

onde se deseja fazer a busca seja especificado como segundo parâmetro. Isso permite que a busca seja feita em outros locais, dependendo da implementação do classloader passado como parâmetro.

A cláusula básica da consulta é `select().from("pacotebase")`, onde é passado o pacote base onde a busca de classes será realizada. Na busca realizada, subpacotes do pacote passado como parâmetro também serão incluídos. Sendo assim, se existem classes em pacotes como `pacotebase.sub1` e `pacotebase.sub2`, elas serão incluídas na busca. Não é aceito como parâmetro uma `String` vazia.

Para a consulta realizada, existem duas possibilidades para o retorno do resultado. A primeira delas é retornar o resultado da query como o retorno do método `getClasses()`. Para isso, utiliza-se o método `returning()` depois da chamada de `from()`. Ele recebe como parâmetro uma condição de filtragem das classes. Algumas possibilidades são as chamadas `allAnnotatedWith()`, para buscar classes com anotações, `allExtending()`, para buscar classes com uma determinada superclasse, e `allImplementing()`, para buscar classes que implementam um determinado conjunto de interfaces. A listagem a seguir mostra um código que busca todas as classes do pacote `scannerexample` que possuem a anotação `@Componente`.

Busca de classes que possuem uma anotação:

```
@Componente
public class ComponentScannerExample {

    public static void main(String[] args) throws Exception {
        ComponentScanner scanner = new ComponentScanner();
        Set<Class<?>> classes =
```

```
            scanner.getClasses(new ComponentQuery(){
            protected void query() {
                select().from("scannerexample").
                    returning(allAnnotatedWith(
                                Componente.class));
            }
        });
        System.out.println(classes);
    }
}
```

Diferentemente do Scannotation, a classe `ComponenteScanner` só busca anotações que estão configurando diretamente sua declaração. Porém, é possível restringir a busca de acordo com algum atributo da anotação, utilizando o método `withArgument()`. Também é possível combinar condições utilizando o método `allBeing()`, o qual pode combinar as condições utilizando os métodos `and()` ou `or()`.

Uma outra possibilidade para a obtenção dos resultados é copiá-los para uma coleção em vez de retorná-lo. Isso é particularmente útil quando se deseja utilizar uma mesma consulta para obter dois tipos diferentes de classes. A listagem a seguir mostra como seria o código nesse caso. Em vez do método `returning()` para essa abordagem é chamado o método `andStore()`. Os métodos de filtragem são similares, porém em vez de `allAlgumaCoisa()` utiliza-se método com `thoseAlgumaCoisa()`. Por exemplo, em vez de `allAnnotatedWith()` utiliza-se `thoseAnnotatedWith()`.

Copiando resultado da busca para coleções:

```
public static void main(String[] args) throws Exception {
    final Set<Class<? extends Serializable>> serializaveis =
        new HashSet<>();
    final Set<Class<?>> encontradas = new HashSet<>();
```

```
    ComponentScanner scanner = new ComponentScanner();
    scanner.getClasses(new ComponentQuery() {
        protected void query() {
            select().from("scannerexample").andStore(
            thoseImplementing(Serializable.class)
                                        .into(serializaveis),
            thoseAnnotatedWith(Componente.class)
                                        .into(encontradas));
        }
    });
    System.out.println(serializaveis);
    System.out.println(encontradas);
}
```

Depois da filtragem, é preciso invocar o método `into()` passando a coleção em que os resultados serão armazenados. Elas podem ser do tipo `Set<Class<?>>` ou `Set<Class<? extends Tipo>>` , onde o `Tipo` é a restrição utilizada na busca para interface ou superclasse. Observe que as coleções são variáveis do tipo `final` , devido ao fato de serem utilizadas dentro da classe anônima que estende `ComponentQuery` .

IDENTIFICANDO CLASSES A SEREM UTILIZADAS A PARTIR DE ANOTAÇÕES

Em aplicações onde flexibilidade é um requisito, uma necessidade é permitir que uma determinada implementação possa ser substituída por outra em diferentes aplicações. Dessa forma, é possível adaptar e estender o comportamento de um componente ou framework. Para que isso seja possível, é preciso provê-lo de um mecanismo o qual ele possa dinamicamente identificar e carregar a classe para ser utilizada. Já vimos anteriormente que com reflexão é possível configurar essas classes em arquivos e instanciá-las em tempo de execução.

Esse mecanismo de busca a partir de anotações é extremamente interessante nesse cenário, pois elimina a necessidade de um arquivo externo de configurações. Através dessas soluções que foram apresentadas, é possível identificar as classes que podem ser utilizadas como um certo componente da aplicação, apenas a partir do fato de implementarem uma interface ou possuírem uma anotação. Isso provê uma forma simples de estender o comportamento de um componente e incluir uma nova implementação em seus pontos de extensão.

8.4 INSERINDO E SUBSTITUINDO ANOTAÇÕES COM ASPECTJ

O **AspectJ** (LADDAD, 2009) é uma linguagem baseada em Java que permite a utilização da orientação a aspectos. A partir desse paradigma de programação, é possível modularizar interesses considerados transversais aos da aplicação. Como forma de implementar essa separação de interesses, o AspectJ possui mecanismos capazes de interceptar a execução de métodos e de inserir novos membros em classes existentes. Porém, esta seção não irá falar sobre a orientação a aspectos, que por si só merece um livro completo, mas sobre o mecanismo que o AspectJ provê para a inserção e substituição de anotações.

No capítulo *Práticas no uso de anotações* foram apresentadas diversas práticas para o uso de anotações, mas para que elas possam ser aplicadas os frameworks devem implementá-las no momento da recuperação das anotações. Caso isso não seja suportado por eles, a princípio não seria possível utilizar configurações default e mapeamento de anotações, por exemplo. O AspectJ fornece uma alternativa que faz um pré-processamento nas classes possibilitando a inserção de anotações em tempo de compilação ou em tempo de carregamento.

Criando e executando projetos com AspectJ no Eclipse

Para trabalhar com o AspectJ, recomenda-se utilizar o plugin AJDT para Eclipse, que possui um suporte muito bom para o uso dessa ferramenta. Dessa forma, na hora de criar um projeto, em vez de criar um projeto Java comum, deve ser criado um projeto do tipo "AspectJ Project". Se o projeto já existe, uma outra opção é clicar nele com o botão direito nele e escolher "Configure > Convert to AspectJ Project". Para executar o projeto com os aspectos, basta escolher "Run as > AspectJ/Java Application". Todas as bibliotecas e configurações necessárias já estarão prontas.

Definindo configurações default para aplicação

Os aspectos proveem formas de inserir novos membros em classes existentes, como métodos e atributos. A partir dessa funcionalidade, também é possível a inserção de anotações. Com isso, podem-se criar configurações default de acordo com convenções mais específicas de uma determinada aplicação. A declaração para a inserção da anotação deve ser feita no seguinte formato: `declare @<tipo-elemento>: <padrão-elemento>: @<Annotação>([])propriedades);` . O tipo de alvo pode ser `type` , `constructor` , `field` ou `method` , e o `<padrão-elemento>` deve definir a convenção de código a ser seguida para a anotação ser inserida. O último parâmetro da declaração é a anotação que se quer inserir nos elementos que obedecerem o padrão.

Na Listagem a seguir, é apresentado um exemplo de aspecto que insere anotações de persistência (JPA) em classes. Nesse aspecto, a anotação `@Entity` é inserida em todas as classes do pacote `br.com.casadocodigo.entidade` ou de um de seus subpacotes. A anotação `@Id` é inserida em todos os atributos dessas classes chamados `id` e a anotação `@Temporal()` com o atributo `value` igual a `TemporalType.DATE` é inserida nos atributos do tipo `Date` de classes dentro do pacote `br.com.casadocodigo.entidade`. Note que diferentes critérios, como padrão de nome e tipo, podem ser utilizados para a inserção da anotação.

Definindo convenções para a inserção de anotações com AspectJ:

```
public aspect AnotacoesEntidade {

    declare @type: br.com.casadocodigo.entidade..* : @Entity;

    declare @field: * br.com.casadocodigo.entidade..*.id : @Id;

    declare @field: Date br.com.casadocodigo.entidade..*.* :
        @Temporal(TemporalType.DATE);
}
```

Por mais que as anotações sejam inseridas pelo aspecto de forma transparente, o plugin AJDT coloca um aviso ao desenvolvedor nos pontos onde anotações estão sendo inseridas. Isso ajuda os desenvolvedores a lidarem com a indireção criada pelo uso de convenções de código, pois indica quando um determinado elemento se encaixa em uma regra. A figura a seguir apresenta uma classe com os indicadores do AJDT, que indicam onde as anotações serão inseridas.

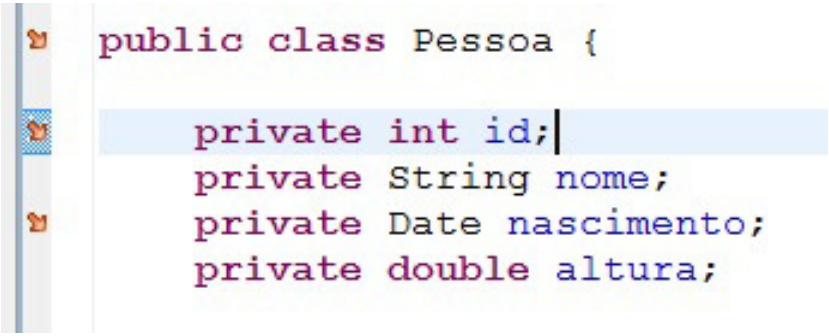

Figura 8.1: Plugin AJDT indicando os pontos onde as regras são ativadas

Mapeando anotações de frameworks existentes

A seção *Mapeamento de anotações* apresentou o conceito de mapeamento de anotações e anotações de domínio, mostrando como um componente poderia buscar por anotações que estão mapeadas para anotações entendias por eles. Uma aplicação disso é a criação de anotações de domínio, que são associadas ao domínio da aplicação e não ao domínio dos frameworks utilizados por elas. Esta seção irá mostrar como o AspectJ pode ser utilizado para fazer o mapeamento de anotações quando isso não é suportado pelo componente que irá o consumir.

Na próxima listagem, é apresentado um exemplo de criação do mapeamento de anotações usando o AspectJ. No exemplo, a anotação `@Administrativo` é uma anotação de domínio que indica que métodos de negócio são relativos à administração do sistema. Essa anotação é então mapeada para duas anotações do EJB 3 para configurar que o método anotado precisa possuir transações e só pode ser acessado por usuários com o papel "admin".

Mapeando anotações com AspectJ:

```
public aspect AnotacoesDominio {

    declare @method:
        @Administrativo * *.*(..): @RolesAllowed(“admin”);

    declare @method: @Administrativo * *.*(..):
            @TransactionAttribute(
                TransactionAttributeType.REQUIRED)
}
```

O QUE MAIS O ASPECTJ FAZ?

O AspectJ é uma ferramenta para um propósito bem mais geral do que o que foi mostrado aqui nesta seção. Os aspectos são tipos de componente que podem ser utilizados para modularizar requisitos transversais ao sistema, ou seja, aqueles que cortam a estrutura de classes do sistema. O exemplo clássico da orientação a aspectos seria o logging, visto que é uma funcionalidade que precisa ser inserida em diversas classes. Um aspecto pode interceptar a execução de um método e inserir membros em classes existentes. Qualquer funcionalidade que se conseguiria implementar com um proxy dinâmico também se consegue facilmente fazer utilizando um aspecto. A vantagem dos aspectos é que, diferente do proxy que precisa ser explicitamente adicionado na classe em algum ponto do código, os aspectos fazem isso através dos pontos de junção, que no exemplo dado são os "padrões" que definem os elementos de código que serão afetados por ele. Infelizmente, falar de aspectos está fora do escopo desse livro, mas se você se interessou, sinceramente acho que vale a pena dar uma olhada...

8.5 CONSIDERAÇÕES FINAIS

Uma das grandes vantagens no ecossistema da linguagem Java é a riqueza de ferramentas e projetos disponíveis para serem utilizados. Este capítulo arranhou a superfície dessa vastidão de projetos apresentando alguns que podem ser muito úteis ao se

trabalhar com reflexão e anotações. É importante ressaltar que não foi o objetivo explorar cada um dos projetos em profundidade ou explicar todas as suas funcionalidades. Como o que foi apresentado pode ser atualizado ou alterado nas próximas versões, sugere-se utilizar os códigos apresentados apenas como referência, recorrendo à sua documentação se necessário.

A primeira ferramenta apresentada foi o projeto BeanUtils, parte do Apache Commons. Esse projeto possui uma série de ferramentas para trabalhar com JavaBeans, o que acaba sendo uma tarefa recorrente para quem trabalha com reflexão. Em seguida, foram apresentadas duas ferramentas para busca de classes, o Scannotation e o Extensible Component Scanner. Ambos podem ser utilizados para buscar classes que possuam determinadas características para serem utilizadas pelos componentes, facilitando a extensão e a adição de novas classes. Por fim, foi mostrado o uso do AspectJ para a adição e mapeamento de anotações. Apesar de ele ser uma ferramenta bem mais ampla, essa funcionalidade permite a definição de configurações default e mapeamento de anotações quando isso não é nativamente suportado pelo componente.

Capítulo 9

MANIPULAÇÃO DE BYTECODE

"Geração de código, assim como beber álcool, é bom com moderação." - Alex Lowe

Toda classe Java para ser executada deve ser compilada para que o bytecode, que é armazenado no arquivo `.class` e posteriormente é carregado pela máquina virtual. Esse é um formato binário bem especificado, que possui informações como a estrutura das classes e a ordem de execução dos métodos. A manipulação de bytecode é um recurso extremamente poderoso que permite que a reflexão computacional seja utilizada na plataforma Java de forma mais poderosa, permitindo a criação de novas classes e modificação de classes existentes.

É através desse recurso que diversos frameworks implementam algumas funcionalidades que parecem mágica. Um exemplo que foi visto na seção *Proxy de classes com CGLib* é a biblioteca CGLib, que utiliza esse recurso para criação de proxies dinâmicos. Outro ferramenta que utiliza esse recurso é o AspectJ, cuja funcionalidade de interceptação de métodos foi mostrada na seção *Outras formas de interceptar métodos* e a de inserção de anotações na seção *Inserindo e substituindo anotações com AspectJ*. A manipulação de

bytecode possibilita também que o código seja instrumentado, isto é, que sejam inseridos pedaços de código que possibilitem a obtenção de métricas da aplicação em tempo de execução.

Existem várias ferramentas para a manipulação de bytecode, como o Javassist (http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/), que é um subprojeto do JBoss, e o BCEL (http://commons.apache.org/proper/commons-bcel/manual.html), que é parte do projeto Apache Commons. Neste capítulo será abordada apenas a ferramenta ASM (não é uma sigla), que possui como diferencial a sua facilidade de uso comparada com outras soluções e sua velocidade de processamento.

9.1 ENTENDENDO O BYTECODE

O bytecode é um formato binário utilizado pela máquina virtual para ler os programas em Java. Uma analogia que poderia ser feita é que o código de máquina está para um processador assim como o bytecode está para máquina virtual Java (Java Virtual Machine — JVM). Sendo assim, qualquer linguagem que compile para bytecode pode ser executada na JVM sem problemas.

Uma característica do bytecode é que ele se limita a menos de 256 tipos de instruções. Isso permite que cada instrução, chamada de **optcode**, seja armazenada em um byte. Dessa característica veio o nome "bytecode". Para vermos na prática a geração do bytecode, considere a classe apresentada na próxima listagem, que simplesmente possui um método `main()` e imprime um texto no console.

Exemplo de classe simples para visualização do bytecode:

```
public class VejaMeuBytecode {
    public static void main(String[] args) {
        System.out.println("Veja meu bytecode!");
    }
}
```

Quando o arquivo `.java` com o código é compilado, é gerado um arquivo `.class` com o bytecode. A próxima listagem apresenta o início do conteúdo do arquivo `.class` que será mostrado se ela for aberta em uma ferramenta simples de edição de texto. Antes que os leitores comecem a se preocupar, você não precisa entender esse código hexadecimal de cabeça. Uma característica curiosa é a sequência `cafe babe` no início do arquivo, que é utilizada para fazer a primeira verificação se ele é um arquivo `.class` válido.

Bytecode resultante da compilação da classe:

```
cafe babe 0000 0032 0022 0700 0201 000f
5665 6a61 4d65 7542 7974 6563 6f64 6507
0004 0100 106a 6176 612f 6c61 6e67 2f4f
626a 6563 7401 0006 3c69 6e69 743e 0100
0328 2956 0100 0443 6f64 650a 0003 0009
...
```

Existem algumas ferramentas que permitem visualizar o bytecode gerado de uma forma mais amigável. Um exemplo é o *class file disassembler* que vem na própria JDK e pode ser invocado a partir do comando `javap` . Dependendo dos parâmetros passados para ele, o bytecode pode ser visualizado com diferentes níveis de detalhamento.

Para os que fogem de linha de comando, uma opção é o Bytecode Outline plugin para o Eclipse, que é desenvolvido pela

própria equipe do ASM. Ele pode ser facilmente instalado pelo Eclipse Marketplace ou pelo update site http://download.forge.objectweb.org/eclipse-update/. Depois de sua instalação, a janela para visualização do bytecode da classe pode ser aberta pelo seguinte caminho: *Window -> Show View -> Other -> Java -> Bytecode*. A listagem a seguir mostra o bytecode da classe exemplos por ele.

Bytecode gerado para classe com uma visualização mais amigável:

```
public class VejaMeuBytecode {

  public <init>() : void
   L0
    LINENUMBER 2 L0
    ALOAD 0: this
    INVOKESPECIAL Object.<init>() : void
    RETURN
   L1
    LOCALVARIABLE this VejaMeuBytecode L0 L1 0
    MAXSTACK = 1
    MAXLOCALS = 1

  public static main(String[]) : void
   L0
    LINENUMBER 4 L0
    GETSTATIC System.out : PrintStream
    LDC "Veja meu bytecode!"
    INVOKEVIRTUAL PrintStream.println(String) : void
   L1
    LINENUMBER 5 L1
    RETURN
   L2
    LOCALVARIABLE args String[] L0 L2 0
    MAXSTACK = 2
    MAXLOCALS = 1
}
```

Observe que nesse código já existe uma grande diferença do

código Java, porém ele é muito mais amigável para humanos compreenderem. Observe que existem comandos como `ALOAD` e `RETURN`, que são palavras chamadas de **mnemônicos** que representam um opticode. Por exemplo, o opticode `B1` representa o comando `RETURN` para métodos que retornam `void`.

Se você está preocupado achando que vai precisar decorar todos esses códigos, fique tranquilo, pois você pode sempre estar gerando o bytecode para ver o que equivale ao comando que você deseja. Se você quer saber o que um comando específico faz, uma opção é observar o bytecode de uma classe sem o comando e comparar com uma classe similar com o comando desejado. Assim a diferença entre o bytecode das duas será o que é gerado na compilação daquele comando.

Para exemplificar esse processo, considere a classe apresentada na próxima listagem. Ela é bem parecida com a utilizada no exemplo desta seção, porém com a adição de uma variável que irá receber a `String` que será impressa no console. O plugin Bytecode Outline também pode ser utilizado nessa comparação, bastando clicar no arquivo .java ou .class e escolhendo "Compare with -> Other Class Bytecode". A figura adiante mostra a janela em que o plugin exibiria as diferenças.

Exemplo de classe modificada para comparação do bytecode:

```
public class VejaMeuBytecode2 {
    public static void main(String[] args) {
        String s = "Veja meu bytecode!";
        System.out.println(s);
    }
}
```

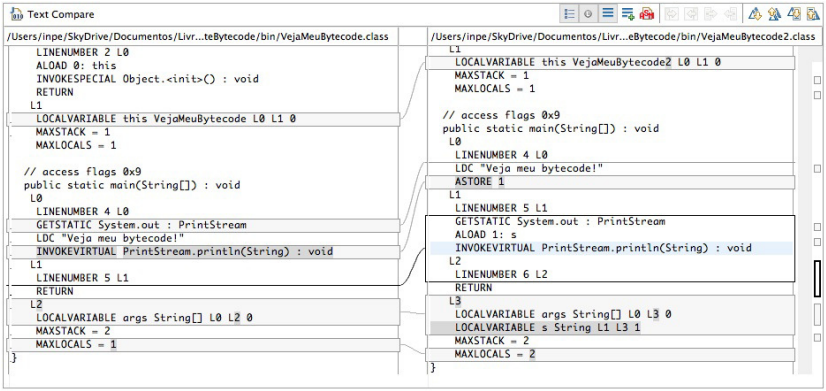

Figura 9.1: Comparação entre o bytecode de duas classes utilizando o Bytecode Outline

9.2 ENTENDENDO O ASM

O ASM é um framework que possibilita a leitura e escrita de arquivos compilados para bytecode em Java. Quem está acostumado com ferramentas de leitura e escrita de outros tipos de arquivos, como documentos XML, perceberá uma certa familiaridade. Ele possui duas APIs distintas para lidar com o bytecode: uma baseada em eventos e outra que organiza os objetos em árvore. Pode ser feito um paralelo com o SAX e o DOM para o processamento de XML. Cada API tem suas vantagens e desvantagens sendo que cada uma é adequada para um tipo de tarefa diferente. Devido à maior simplicidade da API baseada em eventos, somente ela será abordada neste capítulo.

API baseada no padrão Visitor

A API para gerar e transformar classes é baseada na classe abstrata `ClassVisitor` , apresentada na próxima listagem. Cada método que é apresentado nessa interface diz respeito a uma parte

do arquivo que armazena o bytecode de uma classe. Métodos que retornam void recebem como parâmetros informações sobre partes mais simples do bytecode. Seções mais complexas do bytecode, que normalmente são compostas por vários elementos, são representadas por métodos que retornam uma outra interface visitor.

Classe abstrata ClassVisitor:

```
public abstract class ClassVisitor {
  public ClassVisitor(int api);
  public ClassVisitor(int api, ClassVisitor cv);
  public void visit(int version, int access, String name,
      String signature, String superName, String[] interfaces);
  public void visitSource(String source, String debug);
  public void visitOuterClass(String owner, String name,
                              String desc);
  public AnnotationVisitor visitAnnotation(String desc,
                                           boolean visible);
  public void visitAttribute(Attribute attr);
  public void visitInnerClass(String name, String outerName,
      String innerName, int access);
  public FieldVisitor visitField(int access, String name,
      String desc, String signature, Object value);
  public MethodVisitor visitMethod(int access, String name,
      String desc, String signature, String[] exceptions);
  public void visitEnd();
}
```

Para exemplificar esse processo, o método `visit()` é uma chamada que representa o cabeçalho da classe. Pode-se observar que ele retorna `void` e recebe as informações necessárias para criar o cabeçalho da classe, como sua superclasse e suas interfaces. Um elemento da classe composto como um método é representado, por exemplo, pelo `visitMethod()`, que recebe como parâmetro as informações da assinatura do método. Nesse caso, é retornada uma instância de `MethodVisitor`, que possuirá

métodos para representar os elementos internos de um método.

A classe `ClassReader` é responsável por pegar um array de bytes com as informações de uma classe em bytecode e gerar os eventos em um `ClassVisitor`, e nas outras interfaces auxiliares. Já a `ClassWriter` estende `ClassVisitor` e, ao receber as chamadas em seus métodos, vai construindo a classe em formato binário. Sendo assim, podemos dizer que enquanto o `ClassReader` produz evento a partir de bytecode existente, o `ClassWriter` consome esses eventos gerando o bytecode correspondente à chamada dos métodos. O método `accept()` da classe `ClassReader` aceita a visita de uma classe que implemente `ClassVisitor`.

Lendo, escrevendo e transformando bytecode

Essa API normalmente é utilizada em três cenários distintos: usar o `ClassReader` para ler um arquivo em formato bytecode e obter informações sobre a classe; invocar métodos no `ClassWriter` para gerar o bytecode de uma nova classe; e ler uma classe `ClassReader` adaptando e repassar as chamadas recebidas para um `ClassWriter` para transformar uma classe existente.

A próxima listagem apresenta a classe `ContaElementos`, que ilustra o cenário da leitura do bytecode para a obtenção de informações sobre a classe. Ela estende a classe `ClassVisitor` e armazena seu nome, a quantidade de métodos e a quantidade de atributos. O método `getInfoClasse()` retorna um texto com essas informações. Observe que os métodos `visitField()` e `visitMethod()`, depois de incrementarem o contador, invocam

o mesmo método na superclasse e retornam o resultado de sua execução. Isso é importante pois um `ClassReader`, ao invocá-lo, vai pegar o seu retorno e invocar os métodos mais específicos.

Classe que conta a quantidade de métodos e atributos a partir do bytecode:

```
public class ContaElementos extends ClassVisitor {

  private String nomeClasse;
  private int qtdMetodos, qtdAtributos;

  public ContaElementos() {
    super(Opcodes.ASM4);
  }
  public void visit(int version, int access, String name,
      String signature, String superName, String[] interfaces) {
    nomeClasse = name;
  }
  public FieldVisitor visitField(int access, String name,
    String desc, String signature, Object value) {
    qtdAtributos++;
    return super.visitField(access,name,desc,signature,value);
  }
  public MethodVisitor visitMethod(int access, String name,
    String desc, String signature, String[] exceptions) {
    qtdMetodos++;
    return super.visitMethod(access, name, desc, signature,
                             exceptions);
  }
  public String getInfoClasse(){
    return nomeClasse+" possui "+ qtdMetodos +" métodos e " +
        qtdAtributos + " atributos";
  }
}
```

Para que a classe `ContaElementos` possa ser utilizada, é preciso passá-la para um `ClassReader` como feito na listagem a seguir. Inicialmente, é obtido um array de bytes a partir de um arquivo .class, o qual é passado como parâmetro para um

`ClassReader`. Em seguida, é criada uma instância de `ContaElementos`, que é passada no método `accept()` de `ClassReader`. Essa chamada fará com que seja lido o arquivo com o bytecode, sendo que os métodos correspondentes aos elementos lidos serão chamados na implementação de `ClassVisitor` passada como parâmetro.

Execução de uma classe que obtém informações a partir do bytecode:

```
public static void main(String[] args) throws IOException {
  byte[] data = Files.readAllBytes(Paths.get("Teste.class"));
  ClassReader cr = new ClassReader(data);
  ContaElementos ce = new ContaElementos();
  cr.accept(ce,0);
  System.out.println(ce.getInfoClasse());
}
```

Um outro cenário possível para a manipulação de bytecode é a criação de uma nova classe do zero. Em outras palavras, ela será criada baseada em informações da aplicação e não em uma classe já existente. Um exemplo é o framework ClassMock (FERNANDES; GUERRA; SILVEIRA, 2010), visto na seção *Gerando classes com ClassMock*, que cria classes em tempo de execução para serem utilizadas em testes de classes baseadas em reflexão. Nesse caso, os métodos invocados na instância de `ClassMock` é que irão determinar os elementos que estarão presentes na classe.

Para trabalhar dessa forma, somente a classe `ClassWriter` precisa ser utilizada. Ela estende a classe `ClassVisitor` e faz a geração do bytecode a partir dos métodos que são invocados nela. Dessa forma, para que ela seja gerada, os métodos correspondentes à geração daquele bytecode precisam ser invocados. Apesar de ser

necessário saber qual o bytecode para gerar um determinado código, mais à frente neste capítulo será mostrado como utilizar um utilitário que mostra as invocações necessárias para gerar uma classe existente. Dessa forma, não é necessário decorar todos os comandos do bytecode, sendo possível criar uma classe com o código que se deseja gerar e utilizá-la como base para ver quais métodos seriam invocados para a gerar.

A última abordagem apresentada nesta seção para trabalhar com manipulação de bytecode é a transformação de uma classe existente. O objetivo é pegar uma classe como base e realizar mudanças em cima de seu bytecode, sendo possível adicionar, remover e alterar elementos. Uma representação de como funciona a transformação está representado na figura a seguir.

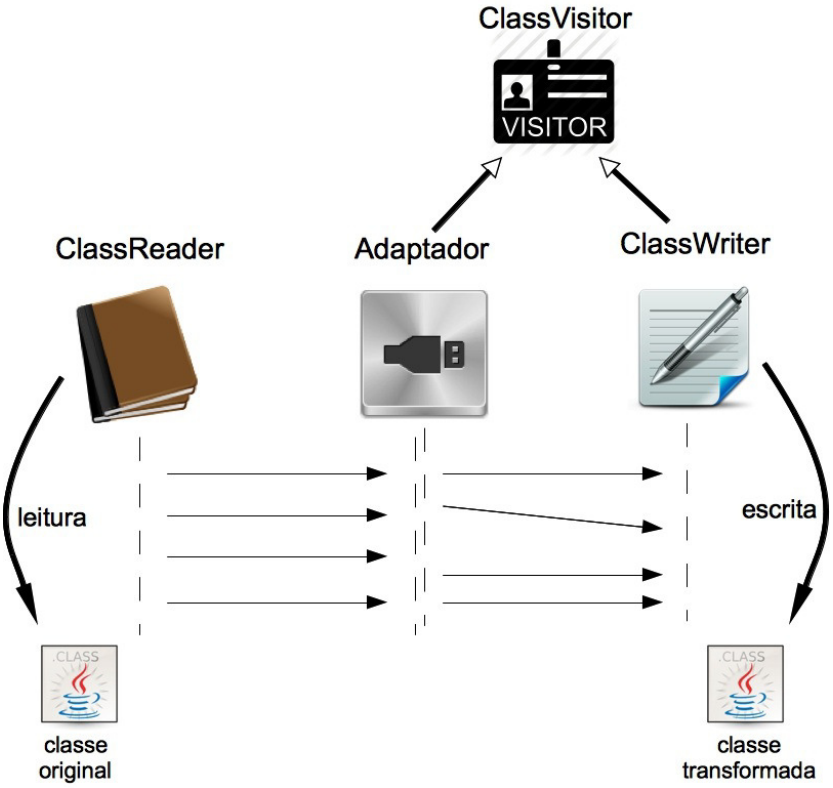

Figura 9.2: Funcionamento da transformação de bytecode

O primeiro passo é utilizar a classe `ClassReader` para ler o bytecode da que será utilizada como base. Em seguida, deve ser criado um adaptador que irá receber as chamadas provenientes da leitura do bytecode e repassar as chamadas a um `ClassWriter`, de acordo com as transformações que devem ser feitas. É importante ressaltar que, para o adaptador poder receber as chamadas de `ClassReader`, ele precisa estender a classe `ClassVisitor`. Dessa forma, see algum elemento precisa ser removido, ele não repassa a chamada ao `ClassWriter` e, se não

precisa ser modificado, repassa exatamente da forma como recebeu. Quando novos elementos precisam ser adicionados ou modificados, o adaptador deve realizar essas chamadas adicionais no `ClassWriter`.

A listagem a seguir apresenta a sequência de invocação para a modificação de uma classe existente. Observe que a classe `Adaptador` encapsula o `ClassWriter`, como se fosse um proxy, e é passada para o `ClassReader` no método `accept()`. Por fim, o bytecode da classe resultante pode ser extraído da instância de `ClassWriter`. Vale ressaltar que diversos adaptadores podem ser encadeados, de forma que cada um faça uma mudança diferente na classe. Os exemplos que serão apresentados mais à frente neste capítulo irão ilustrar a transformação e a geração de bytecode com mais detalhes.

Execução da transformação de uma classe:

```
ClassWriter cw = new ClassWriter();
Adaptador adaptador = new Adaptador(cw);
ClassReader cr = new ClassReader(originalClass);
cr.accept(adaptador,0);
byte[] transformada = cw.toByteArray();
```

Trabalhando com atributos e métodos

Como foi dito anteriormente, `ClassVisitor` possui métodos apenas para representar os elementos principais da classe. Para elementos compostos, como métodos, atributos e anotações, existem outras classes que são retornadas por aqueles de `ClassVisitor` que possuem as chamadas para representar seus elementos internos. Para exemplificar como isso funciona, esta seção irá estender o exemplo apresentado na seção anterior que

contava a quantidade de métodos e atributos contidos pela classe. Agora, ele irá também contar as anotações dos métodos e atributos, exemplificando o tratamento dos eventos internos de cada um.

A listagem a seguir exemplifica a criação de um `FieldVisitor` que conta as anotações de um atributo. Observe que ele recebe como parâmetro somente a instância da classe `ContaElementos`, onde ele chama o método `contaAnotacao()` para incrementar a contagem de anotações. Ele sobrescreve somente o método `visitAnnotation()`, no qual realiza a contagem e retorna o resultado da execução do mesmo método na superclasse.

Exemplo de um FieldVisitor para contar anotações:

```
public class ContaAnotacoesAtributo extends FieldVisitor {

  private ContaElementos ce;

  public ContaAnotacoesAtributo(ContaElementos ce) {
    super(Opcodes.ASM4);
    this.ce = ce;
  }
  public AnnotationVisitor visitAnnotation(String desc,
   boolean visible) {
    ce.contaAnotacao();
    return super.visitAnnotation(desc, visible);
  }
}
```

A próxima listagem apresenta a classe `ContaAnotacoesMetodo`, que é similar a apresentada anteriormente, porém estendendo `MethodVisitor` ao invés de `FieldVisitor`. Observe que novamente somente o método `visitAnnotation()` é sobrescrito.

Exemplo de um MethodVisitor para contar anotações:

```
public class ContaAnotacoesMetodo extends MethodVisitor{

  private ContaElementos ce;

  public ContaAnotacoesMetodo(MethodVisitor mv,
   ContaElementos ce) {
    super(Opcodes.ASM4);
    this.ce = ce;
  }
  public AnnotationVisitor visitAnnotation(String desc,
   boolean visible) {
    ce.contaAnotacao();
    return super.visitAnnotation(desc, visible);
  }
}
```

Por fim, o mais importante para que essas classes sejam invocadas é que sejam retornadas nas respectivas chamadas de `visitField()` e `visitMethod()` na implementação de `ClassVisitor`. Quando o ASM está lendo o bytecode de uma classe e encontra, por exemplo, um método, ele primeiro invoca o `visitMethod()` no `ClassVisitor` e, em seguida, invoca métodos relativos aos elementos do método na instância de `MethodVisitor` retornada por ele. Dessa forma, é importante que eles retornem as instâncias das classes adequadas, se não elas não serão utilizadas.

A próxima listagem mostra uma nova versão da classe `ContaElementos`, que inclui a contagem de anotações nos métodos e atributos. Observe que foi adicionado o atributo `qtdAnotacoes` para armazenar a quantidade de anotações, assim como um método público chamado `contaAnotacao()` para ser invocado pelas respectivas classes visitor. Observe que os métodos `visitField()` e `visitMethod()` retornam respectivamente

instâncias de `ContaAnotacoesAtributo` e `ContaAnotacoesMetodo`. Como foi dito, será nessas instâncias que os métodos mais específicos de cada um serão invocados. Adicionalmente, o método `getInfoClasse()` também retorna a quantidade de anotações.

Execução de uma classe que obtém informações a partir do bytecode:

```
public class ContaElementos extends ClassVisitor {

  private String nomeClasse;
  private int qtdMetodos, qtdAtributos, qtdAnotacoes;

  public ContaElementos() {
    super(Opcodes.ASM4);
  }
  public void visit(int version, int access, String name,
   String signature, String superName, String[] interfaces) {
    nomeClasse = name;
  }
  public FieldVisitor visitField(int access, String name,
   String desc, String signature, Object value) {
    qtdAtributos++;
    return new ContaAnotacoesAtributo(this);
  }
  public MethodVisitor visitMethod(int access, String name,
   String desc, String signature, String[] exceptions) {
    qtdMetodos++;
    return new ContaAnotacoesMetodo(this);
  }
  public void contaAnotacao(){
    qtdAnotacoes++;
  }
  public String getInfoClasse(){
    return nomeClasse+" possui "+ qtdMetodos +" métodos e " +
        qtdAtributos + " atributos, com "+
        qtdAnotacoes +" anotações";
  }
}
```

ASMificando uma classe

Uma grande dificuldade para quem está pensando em trabalhar com manipulação de bytecode é a necessidade de entender os optcodes e como um determinado código Java é compilado para eles. Felizmente, apesar de esse conhecimento ser muito útil para uma compreensão mais profunda do código com o qual se está trabalhando, essa é uma questão fácil de ser contornada. Existem utilitários e ferramentas que "ASMificam" o bytecode de uma classe existente. Em outras palavras, ele pega o bytecode de uma classe e cria o código necessário para o ASM criar aquele bytecode.

Há duas alternativas para executar esse utilitário do ASM, sendo através da linha de comando ou usando o plugin Bytecode Outline. A próxima listagem mostra o comando que precisaria ser executado para ASMificar o bytecode da classe armazenada no arquivo `Teste.class` . A figura adiante apresenta o mesmo processo sendo feito pelo Bytecode Outline. Observe que na parte direita superior da aba "Bytecode" existe um botão escrito ASM, que, se for selecionado, alterna na visualização pura do bytecode para a visualização do código ASM para gerá-lo. Uma vantagem na utilização do plugin é que selecionando uma linha do código da classe no editor, ele mostra qual seria a parte do código ASM para gerá-la.

Execução da ASMificação através da linha de comando:

```
java -cp asm.jar;asm-util.jar org.objectweb.asm.util
 .ASMifierClassVisitor Teste.class
```

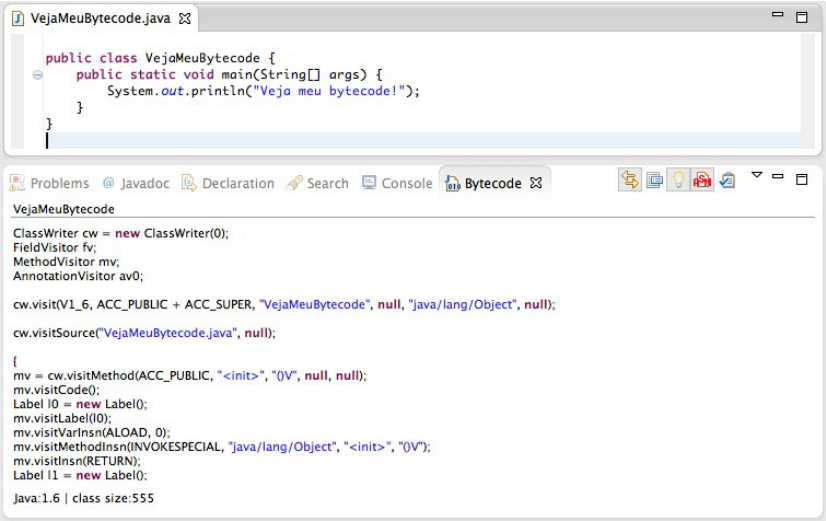

Figura 9.3: Funcionamento da transformação de bytecode

A listagem a seguir apresenta o resultado da ASMificação resultante da classe `VejaMeuBytecode` apresentada anteriormente neste capítulo. Observe o código e veja como funciona, por exemplo, a geração de um método. Inicialmente, é invocado `visitmethod()` em `ClassWriter` e depois são invocados métodos nele para os comandos que serão adicionados dentro do método. Observe que, mesmo não sendo necessário, essa ferramenta coloca a geração de cada elemento do código, como métodos e atributos, dentro de blocos de código para facilitar sua leitura.

Resultado a ASMificação da classe:

```
public class VejaMeuBytecodeDump implements Opcodes {

  public static byte[] dump() throws Exception {

    ClassWriter cw = new ClassWriter(0);
```

```
    FieldVisitor fv;
    MethodVisitor mv;
    AnnotationVisitor av0;

    cw.visit(V1_6, ACC_PUBLIC + ACC_SUPER, "VejaMeuBytecode",
     null, "java/lang/Object", null);
    cw.visitSource("VejaMeuBytecode.java", null);

    {
      mv = cw.visitMethod(ACC_PUBLIC + ACC_STATIC, "main",
          "(Ljava/lang/String;)V", null, null);
      mv.visitCode();
      Label l0 = new Label();
      mv.visitLabel(l0);
      mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out",
          "Ljava/io/PrintStream;");
      mv.visitLdcInsn("Veja meu bytecode!");
      mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream",
       "println", "(Ljava/lang/String;)V");
      Label l1 = new Label();
      mv.visitLabel(l1);
      mv.visitInsn(RETURN);
      Label l2 = new Label();
      mv.visitLabel(l2);
      mv.visitMaxs(2, 1);
      mv.visitEnd();
    }
    cw.visitEnd();
    return cw.toByteArray();
  }
}
```

A partir dessa ferramenta, é possível trabalhar com o ASM de uma forma bem simples. No caso do uso para geração de novas classes, é possível ASMificar classes para servirem de modelo para o código que deseja ser gerado. Se o desejado é a modificação, é possível criar versões da classe, antes e depois da modificação, e verificar como o ASM geraria o código que precisa ser inserido.

9.3 QUANDO O BYTECODE PODE SER

ALTERADO?

Existem diversas formas de utilização do ASM no contexto de uma aplicação ou framework. A forma mais simples de utilizá-lo é na transformação ou criação de classes antes da execução da aplicação. Neste caso, a aplicação que faz a transformação lê o arquivo que possui a definição da classe e, após a transformação, cria novos arquivos ou sobrescreve o arquivo original. Essa abordagem deve ser utilizada quando se desejar fazer um pré-processamento nos arquivos .class antes de eles serem carregados por uma aplicação. Esse procedimento normalmente é integrado à construção da aplicação, executando depois de sua compilação.

Uma outra abordagem de utilização do ASM é para a criação de classes em tempo de execução. A máquina virtual Java não permite a alteração de uma classe que já foi carregada por ela, então qualquer classe criada em tempo de execução precisaria ser uma nova classe. Quando se deseja modificar o comportamento de uma classe, uma opção é a criação de uma nova classe, que estende essa classe e implementa um proxy dinâmico, como é feito com a biblioteca CGLib e foi mostrado na seção *Proxy de classes com CGLib*.

Para a geração de novas classes em tempo de execução, é preciso criar um `ClassLoader` que torne público o método `defineClass()`. A listagem a seguir mostra como deve ser criado esse `ClassLoader`. Observe que essa classe não tem nenhuma lógica adicional a não ser a delegação do método `accept()` para a superclasse.

ClassLoader que permite o carregamento de uma classe definida em tempo de execução:

```
public class DynamicClassLoader extends ClassLoader {
  public Class defineClass(String name, byte[] b){
    return defineClass(name, b, 0, b.length);
  }
}
```

A próxima listagem mostra como a classe gerada dinamicamente seria carregada por esse `ClassLoader`. O primeiro passo seria criar uma instância do `ClassLoader` e, em seguida, o método `defineClass()` seria invocado passando o nome da classe e o respectivo array de bytes com o seu bytecode. Isso vai retornar uma instância de `Class`, da qual instâncias podem ser criadas e manipuladas utilizando reflexão. Como a classe nem existe em tempo de compilação, não seria possível acessá-la de outra forma, a não ser que ela implemente uma interface ou estenda uma classe conhecida.

Exemplo de como seria o carregamento de uma classe em tempo de execução:

```
ClassWriter cw = new ClassWriter();
//escrita do bytecode da classe

DynamicClassLoader cl = new DynamicClassLoader();
Class classe = cl.defineClass("NomeClasse", cw.toByteArray());
Object obj = classe.newInstance();
```

Outro momento em que poderia ser feita a manipulação de bytecode seria quando elas são carregadas pela máquina virtual. Dessa forma, o carregamento das classes seria interceptado, permitindo que a classe seja modificada. Para fazer isso, é preciso criar uma classe que implementa a interface `ClassFileTransformer` e adicioná-la na instância de `Instrumentation` durante a execução do método `premain()`. A listagem a seguir apresenta como a classe

`TransformadorClasses` seria adicionada para a modificação das classes. Vale ressaltar que vários transformadores podem ser definidos em cadeia no método `premain()`.

ClassLoader que permite o carregamento de uma classe definida em tempo de execução:

```
public class PreMain {
  public static void premain(String arg, Instrumentation i)
   throws ClassNotFoundException {
    i.addTransformer(new TransformadorClasses());
  }
}
```

Para que o método `premain()` seja executado, a classe que o contém deve estar dentro de um arquivo jar. Também é necessário que no arquivo `MANIFEST.MF`, que fica dentro do diretório `META-INF`, tenha a propriedade `Premain-Class` seguida do nome da classe que contém esse método. A listagem a seguir exemplifica como seria o conteúdo desse arquivo.

Exemplo de MANIFEST.MF para execução do premain():

```
Manifest-Version: 1.0
Premain-Class: pacote.PreMain
```

Finalmente, é preciso definir a classe `TransformadorClasses` que irá efetivamente fazer a modificação. A próxima listagem apresenta um modelo de como seria a estrutura dessa classe. Observe que o método `transform()` recebe como um dos parâmetros um array de bytes chamado `classFileBuffer`, que representa o bytecode da classe que está sendo carregada. Se esse método `transform()` retornar null, significa que aquela classe não será redefinida por aquele transformador. Caso contrário, se esse método retornar um novo array de bytes, esse será

considerado como o novo bytecode daquela classe.

Classe que realiza a transformação de classes em tempo de carregamento:

```
public class TransformadorClasses implements
 ClassFileTransformer {
  public byte[] transform(ClassLoader cl, String className,
    Class<?> classBeingRedefined, ProtectionDomain pd,
    byte[] classFileBuffer) throws IllegalClassFormatException {

    ClassReader cr = new ClassReader(classfileBuffer);
    ClassWriter cw =
      new ClassWriter(cr, ClassWriter.COMPUTE_FRAMES);
    ClassVisitor cv = new Adaptador(cw);

    return cw.toByteArray();
  }
}
```

Além de toda essa configuração, ainda é necessário incluir o seguinte parâmetro no momento de executar a aplicação: `-javaagent:arquivo.jar` . No lugar de "arquivo.jar" seria incluído o nome do arquivo que possui a classe `PreMain` , a classe que realiza a transformação e a configuração no manifest. Esse arquivo deveria estar no classpath da aplicação.

Cálculo de frames e mapas de variáveis

Observe que na última listagem o parâmetro `ClassWriter.COMPUTE_FRAMES` foi passado no construtor da classe `ClassWriter`. Existem diversos cálculos que devem ser feitos a respeito dos frames e mapa de variáveis locais de um método e, a não ser que você seja um usuário muito avançado, você vai querer que o ASM faça esse cálculo para você. A documentação diz que esse cálculo pode deixar a geração da classe mais lenta, porém para manter a simplicidade, esse parâmetro será utilizado para os exemplos que serão mostrados.

9.4 CRIANDO UM BEAN A PARTIR DE UM MAPA

Como primeiro exemplo de uso da manipulação de bytecode, será considerada a geração de novas classes. Para isso, como já foi feita no início do livro a geração de um mapa a partir das propriedades de um bean, esse exemplo irá fazer o oposto: a geração de um bean a partir das propriedades de um mapa. Para isso, precisaremos gerar uma nova classe com aquelas propriedades e, em seguida, populá-la com os valores.

Como foi visto neste capítulo, o primeiro passo para ter um modelo de como o bytecode precisa ser gerado é criar uma classe para servir de base. A listagem a seguir mostra a classe `Teste` que possui a estrutura básica de um bean com uma propriedade, que

inclui um atributo e seus métodos de acesso. A ideia é que deve ser gerada uma propriedade com essa estrutura para cada entrada do mapa.

Classe para ser utilizada como exemplo na geração do bytecode:

```
public class Teste {

  private Object prop;

  public Object getProp() {
    return prop;
  }

  public void setProp(Object prop) {
    this.prop = prop;
  }
}
```

Depois da criação para ser utilizada como modelo, o passo seguinte é a geração do seu código ASM correspondente. A listagem a seguir apresenta o código gerado a partir do plugin Bytecode Outline. Uma dica para que o código gerado seja apenas o essencial é não utilizar opções de compilação que adicionem informações no bytecode, principalmente as que são utilizadas para debug, como para adição da informação sobre linhas de código. Um fato interessante de ser notado é que, mesmo não sendo definido de forma explícita no código, o bytecode contém um construtor default, que é o método definido com nome `<init>` . Fora isso, é possível ver claramente a definição do atributo e dos dois métodos de acesso.

Classe modelo ASMificada:

```
public class TesteDump implements Opcodes {
```

```
public static byte[] dump() throws Exception {

  ClassWriter cw = new ClassWriter(0);
  FieldVisitor fv;
  MethodVisitor mv;
  AnnotationVisitor av0;

  cw.visit(V1_6, ACC_PUBLIC + ACC_SUPER, "Teste", null,
      "java/lang/Object", null);

  {
    fv = cw.visitField(ACC_PRIVATE, "prop",
          "Ljava/lang/Object;", null, null);
    fv.visitEnd();
  }
  {
    mv = cw.visitMethod(ACC_PUBLIC, "<init>", "()V", null,
                        null);
    mv.visitCode();
    mv.visitVarInsn(ALOAD, 0);
    mv.visitMethodInsn(INVOKESPECIAL, "java/lang/Object",
        "<init>", "()V");
    mv.visitInsn(RETURN);
    mv.visitMaxs(1, 1);
    mv.visitEnd();
  }
  {
    mv = cw.visitMethod(ACC_PUBLIC, "getProp",
         "()Ljava/lang/Object;", null, null);
    mv.visitCode();
    mv.visitVarInsn(ALOAD, 0);
    mv.visitFieldInsn(GETFIELD, "Teste", "prop",
                      "Ljava/lang/Object;");
    mv.visitInsn(ARETURN);
    mv.visitMaxs(1, 1);
    mv.visitEnd();
  }
  {
    mv = cw.visitMethod(ACC_PUBLIC, "setProp",
         "(Ljava/lang/Object;)V", null, null);
    mv.visitCode();
    mv.visitVarInsn(ALOAD, 0);
    mv.visitVarInsn(ALOAD, 1);
    mv.visitFieldInsn(PUTFIELD, "Teste", "prop",
                      "Ljava/lang/Object;");
```

```
      mv.visitInsn(RETURN);
      mv.visitMaxs(2, 2);
      mv.visitEnd();
    }
    cw.visitEnd();

    return cw.toByteArray();
  }
}
```

Para desenvolver a geração do bean, vamos utilizar uma abordagem *top-down*, ou seja, começar com a rotina principal e depois descer para as funções auxiliares. A listagem a seguir apresenta o método `gerar()`, que possui os passos para a geração do bean a partir do mapa. Inicialmente, é chamado o método `criarClasseEConstrutor()`, e para cada item do mapa são chamados métodos para geração do atributo e dos dois métodos de acesso. Após isso, é utilizada a classe `DynamicClassLoader` para carregar a classe gerada e, por meio de reflexão, instanciá-la e inserir as propriedades, com a biblioteca BeanUtils mostrada na seção *Manipulando beans com BeanUtils*.

Método principal para geração do bean:

```
public Object gerar(String nome, Map<String,Object> mapa) throws
 Exception{
  ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);

  criarClasseEConstrutor(nome, cw);

  for(String prop : mapa.keySet()){
    criarAtributoPrivado(prop, cw);
    criarGetter(nome, cw, prop);
    criarSetter(nome, cw, prop);
  }
  DynamicClassLoader cl = new DynamicClassLoader();
  Class classe = cl.defineClass(nome, cw.toByteArray());
  Object obj = classe.newInstance();
  for(String prop : mapa.keySet()){
```

```
    PropertyUtils.setProperty(obj, prop, mapa.get(prop));
  }
  return obj;
}
```

Começando a descer para os métodos auxiliares, a próxima listagem mostra o método `criarClasseECosntrutor()`. Observe que seu corpo é exatamente o código retirado da classe ASMificada para a definição do cabeçalho e para o construtor. A única diferença do código gerado é que o nome da classe foi substituído pelo parâmetro.

Método que gera o bytecode da classe e de seu construtor:

```
protected void criarClasseEConstrutor(String nome,
 ClassWriter cw) {
  cw.visit(V1_6, ACC_PUBLIC + ACC_SUPER, nome, null,
           "java/lang/Object", null);
  MethodVisitor mv =
    cw.visitMethod(ACC_PUBLIC, "<init>", "()V", null, null);
  mv.visitCode();
  mv.visitVarInsn(ALOAD, 0);
  mv.visitMethodInsn(INVOKESPECIAL, "java/lang/Object",
                     "<init>", "()V");
  mv.visitInsn(RETURN);
  mv.visitMaxs(1, 1);
  mv.visitEnd();
}
```

A listagem a seguir apresenta os outros métodos utilizados para a geração do bytecode referente a uma propriedade, incluindo a criação do atributo e dos métodos de acesso. Observe que o código obtido a partir da ASMificação da classe utilizada como modelo é alterado em alguns pontos para refletir as particularidades da classe sendo gerada. Nesse caso, o nome da classe e da propriedade foram substituídos pelos parâmetros passados aos métodos.

Métodos para criação de um atributo com seus respectivos

métodos de acesso:

```
protected void criarAtributoPrivado(String prop, ClassWriter cw)
{
  FieldVisitor fv = cw.visitField(ACC_PRIVATE, prop,
      "Ljava/lang/Object;", null, null);
}
protected void criarGetter(String nome, ClassWriter cw,
 String prop) {
  String propM = prop.substring(0, 1) + prop.substring(1);
  MethodVisitor mv = mv = cw.visitMethod(ACC_PUBLIC,
      "get"+propM , "()Ljava/lang/Object;", null, null);
  mv.visitCode();
  mv.visitVarInsn(ALOAD, 0);
  mv.visitFieldInsn(GETFIELD, nome, prop, "Ljava/lang/Object;");
  mv.visitInsn(ARETURN);
  mv.visitMaxs(1, 1);
  mv.visitEnd();
}
protected void criarSetter(String nome,
 ClassWriter cw, String prop) {
  String propM = prop.substring(0, 1) + prop.substring(1);
  MethodVisitor mv = cw.visitMethod(ACC_PUBLIC, "set"+propM,
    "(Ljava/lang/Object;)V", null, null);
  mv.visitCode();
  mv.visitVarInsn(ALOAD, 0);
  mv.visitVarInsn(ALOAD, 1);
  mv.visitFieldInsn(PUTFIELD, nome, prop, "Ljava/lang/Object;");
  mv.visitInsn(RETURN);
  mv.visitMaxs(2, 2);
  mv.visitEnd();
}
```

Por fim, próxima listagem demonstra o uso do código desenvolvido. Inicialmente, é criado um mapa com duas propriedades e, em seguida, invoca-se o método `gerar()` da classe `MapaParaBean` , passando `Pessoa` como parâmetro para ser o nome da classe. Em seguida, para verificar se a classe retornada realmente está no formato de bean e possui as propriedades que estavam no mapa, o método `getProperty()` da classe `BeanUtils` é utilizada para recuperar as propriedades, que

em seguida são impressas no console.

Exemplo de geração de um bean através de um mapa:

```
public static void main(String[] args) throws Exception{
  HashMap<String, Object> m = new HashMap<>();
  m.put("nome", "Eduardo");
  m.put("idade",34);

  MapaParaBean mpb = new MapaParaBean();
  Object obj = mpb.gerar("Pessoa", m);
  System.out.println(BeanUtils.getProperty(obj, "nome"));
  System.out.println(BeanUtils.getProperty(obj, "idade"));
}
```

A ideia desse exemplo foi ilustrar o uso do bytecode para a criação e carregamento de uma classe em tempo de execução. Nesse caso, o bean é criado com base em um mapa cujas informações realmente só podem ser acessadas em tempo de execução, impedindo que o manipulação do bytecode seja feita em um outro momento. Como a classe não existia antes, só é possível acessar seus métodos através de reflexão ou, nesse exemplo, bibliotecas como o Bean Utils. No caso de se saber de antemão algum método da classe, é possível definir de forma estática uma interface para ser implementada pela classe que será gerada, sendo assim, pode ser feito o cast para a interface e os métodos invocados normalmente.

9.5 DATA DA ÚLTIMA MODIFICAÇÃO DE UM OBJETO

Imagine que em um sistema seja importante saber o momento em que um objeto foi modificado pela última vez. A implementação dessa funcionalidade em uma classe com algumas

propriedades não é problema, porém pode ser complicado e trabalhoso adicionar isso em diversas classes de um sistema. Sendo assim, esta seção mostra como essa característica pode ser adicionada nas classes através da manipulação do bytecode, transformando-as no momento em que estiverem sendo carregadas.

Para permitir que as classes possam recuperar o momento da última atualização sem precisar recorrer à reflexão, a listagem a seguir apresenta a interface `UltimaAtualizacao`. Ela será inserida nas classes no momento da manipulação do seu bytecode, não havendo necessidade das classes que serão transformadas fazerem isso explicitamente no seu código.

Interface para a recuperação do momento da última atualização:

```
public interface UltimaAtualizacao {
  public long getUltimaModificacao();
}
```

Da mesma forma que foi feito antes, será criada uma classe para ser ASMificada e utilizada como modelo. Como dessa vez as classes serão modificadas e não criadas, precisamos ver qual a diferença entre uma classe antes e depois da modificação. Sendo assim, a classe `Teste` da seção anterior será utilizada como "antes" e a `TesteModificado` da próxima listagem será utilizada como base para o depois. Observe que as modificações formam: implementação da interface `UltimaAtualizacao`, adição do atributo `ultimaModificacao`, adição do método `getUltimaModificacao()` e atualização do atributo `ultimaModificacao` no método setter. A figura adiante mostra a comparação do bytecode dessas duas classes utilizando o plugin

Bytecode Outline.

Classe modificada para servir como modelo:

```
public class TesteModificado implements UltimaAtualizacao{

  private Object prop;
  private long ultimaModificacao;

  public Object getProp() {
    return prop;
  }
  public void setProp(Object prop) {
    this.prop = prop;
    ultimaModificacao = System.currentTimeMillis();
  }
  public long getUltimaModificacao() {
    return ultimaModificacao;
  }
}
```

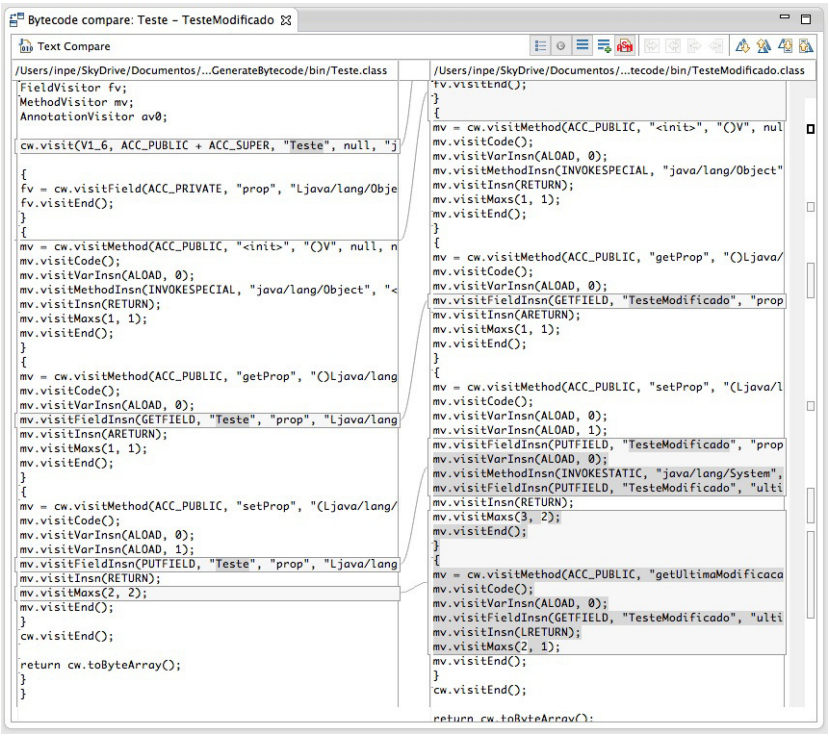

Figura 9.4: Comparação do bytecode ASMificado das classes

Para esse exemplo, primeiramente será mostrada a lógica para a manipulação do bytecode e, em seguida, como ela seria inserida no momento do carregamento da classe. A listagem a seguir apresenta classe `UltimaModificacaoAdaptadorClasse` que estende `ClassVisitor` para a modificação de uma classe. Observe que ela recebe um outro `ClassVisitor` em seu construtor, onde se espera que se passe como parâmetro um `ClassWriter` ou uma outra implementação de `ClassVisitor` para realizar outra alteração em cadeia. Essa instância recebida é passada no construtor da superclasse, a qual irá delegar para ela

todas as chamadas de métodos que não forem sobrescritos.

ClassVisitor para modificação da classe:

```
public class UltimaModificacaoAdaptadorClasse extends
 ClassVisitor {

  private String nomeClasse;

  public UltimaModificacaoAdaptadorClasse(ClassVisitor visitor){
    super(Opcodes.ASM4, visitor);
  }

  @Override
  public void visit(int version, int access, String name,
      String signature, String superName, String[] interfaces) {

    nomeClasse = name;

    String[] novasInterfaces =
      Arrays.copyOf(interfaces, interfaces.length+1);
    novasInterfaces[interfaces.length] = "UltimaAtualizacao";
    cv.visit(version, access, name, signature, superName,
             novasInterfaces);

    FieldVisitor fv = cv.visitField(ACC_PRIVATE,
                                    "ultimaModificacao",
                                    "J", null, null);
    fv.visitEnd();

    MethodVisitor mv = cv.visitMethod(ACC_PUBLIC,
                                      "getUltimaModificacao",
                                      "()J", null, null);
    mv.visitCode();
    mv.visitVarInsn(ALOAD, 0);
    mv.visitFieldInsn(GETFIELD, nomeClasse, "ultimaModificacao",
                      "J");
    mv.visitInsn(LRETURN);
    mv.visitMaxs(2, 1);
    mv.visitEnd();
  }

  @Override
  public MethodVisitor visitMethod(int access, String name,
```

```
    String desc, String signature, String[] exceptions) {
     MethodVisitor mv =
       cv.visitMethod(access, name, desc, signature, exceptions);
     if(name.startsWith("set")){
       mv = new UltimaModificacaoAdaptadorMetodo(mv, nomeClasse);
     }
     return mv;
   }
}
```

Quando a modificação de uma classe envolve a adição de novos elementos, deve-se escolher o momento em que serão inseridos e sobrepor um dos métodos de `ClassVisitor` para adicioná-los. No caso da adição do atributo para armazenar o momento da última atualização e seu respectivo método de acesso, decidiu-se que eles seriam adicionados logo após a declaração da classe. Dessa forma, o próprio método `visit()` foi utilizado para adicionar esses elementos.

O primeiro passo do método `visit()` é a adição da interface `UltimaAtualizacao` na classe. Dessa forma, cria-se um array com o mesmo conteúdo do parâmetro `interfaces`, porém com uma posição a mais, onde é adicionada a nova interface. Observe que é esse novo array que é passado para a chamada ao método `visit()` repassada. Observe que essas chamadas são feitas em uma variável chamada `cv`, a qual é declarada na superclasse e armazena o `ClassVisitor` passado em seu construtor. Em seguida, são adicionados o novo atributo e o seu respectivo método de acesso, cujo código se baseia no resultado obtido da ASMificação.

A modificação faltante é a atualização do valor do atributo `ultimaModificacao` no método setter, porém, para isso a transformação deve ser feita dentro do método. Dessa forma,

`visitMethod()` é sobreposto para que a instância de `MethodVisitor` possa ser encapsulada pela instância de `UltimaModificacaoAdaptadorMetodo` em métodos iniciados com "set". Essa classe responsável pela transformação do método está apresentada na listagem a seguir.

MethodVisitor que adiciona atualização do momento de acesso em métodos setters:

```
public class UltimaModificacaoAdaptadorMetodo extends
 MethodVisitor {

  private String nomeClasse;

  public UltimaModificacaoAdaptadorMetodo(MethodVisitor visitor,
   String nomeClasse) {
    super(Opcodes.ASM4, visitor);
    this.nomeClasse = nomeClasse;
  }

  @Override
  public void visitCode() {
    mv.visitCode();
    mv.visitVarInsn(ALOAD, 0);
    mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
                       "currentTimeMillis", "()J");
    mv.visitFieldInsn(PUTFIELD, nomeClasse, "ultimaModificacao",
                      "J");
  }
}
```

O encapsulamento do `MethodVisitor` passado como parâmetro é análogo ao encapsulamento do `ClassVisitor` , já explicado anteriormente. Nesse caso, a variável que armazena o `MethodVisitor` encapsulado se chamada `mv` . Ao observar o código ASMificado, é possível perceber que as chamadas relativas à atualização do atributo `ultimaModificacao` ocorrem logo após uma chamada do método `visitCode()` . Dessa forma, esse

método foi sobreposto para que essas chamadas fossem adicionadas.

Depois da implementação da lógica da manipulação de bytecode, o próximo passo é criar uma classe que invoque essa lógica no momento do carregamento. A próxima listagem mostra o código de `TransformadorUltimaModificacao`, que implementa `ClassFileTransformer` como apresentado anteriormente neste capítulo. A novidade nessa listagem é que é feito um filtro para as classes que devem ser modificadas. Nesse caso, são as classes que pertencerem ao pacote `org.casadocodigo`, porém outros critérios podem ser utilizados.

Classe que executa a transformação das classes:

```
public class TransformadorUltimaModificacao implements
 ClassFileTransformer {

  @Override
  public byte[] transform(ClassLoader loader, String className,
      Class<?> classBeingRedefined,
      ProtectionDomain protectionDomain,
      byte[] classfileBuffer) throws IllegalClassFormatException
      {

    if(!className.startsWith("org/casadocodigo"))
      return null;

    ClassReader cr = new ClassReader(classfileBuffer);
    ClassWriter cw =
      new ClassWriter(cr, ClassWriter.COMPUTE_FRAMES);
    ClassVisitor cv = new UltimaModificacaoAdaptadorClasse(cw);
    cr.accept(cv, 0);
    return cw.toByteArray();
  }
}
```

Filtrando classes para transformação

É sempre aconselhável utilizar algum critério para filtrar as classes cujo bytecode precisa ser manipulado, pois, se não, todas as classes carregadas pela máquina virtual, incluindo as da própria biblioteca da JDK irão passar por ali. Exemplos comuns incluem filtrar por algum padrão de nome, ou a presença de uma interface ou anotação. O método `transform()` recebe como parâmetro uma instância do tipo `Class` chamada `classBeingRedefined`, porém, deve-se tomar muito cuidado ao se utilizá-la pois muitas vezes o valor passado é nulo. Sendo assim, o aconselhável, no caso de um filtro que verifique a presença de uma interface ou anotação, é extrair isso do bytecode mesmo. Como foi mostrado, isso pode ser feito com o ASM a partir de um `ClassVisitor` que apenas recupera e armazena as informações que são de interesse.

O último passo é a criação de uma classe com o método `PreMain`, como mostrada na próxima listagem, e o empacotamento dessa classe em um arquivo jar. Observe que ela adiciona o transformador criado na instância de `Instrumentation` recebida como parâmetro pelo método `premain()`. É importante ressaltar que ela não pode estar no pacote default.

Classe que adiciona o transformador para ser executado no carregamento das classes:

```
package pacote;

public class PreMain {
  public static void premain(String arg, Instrumentation i)
   throws ClassNotFoundException {
      i.addTransformer(new TransformadorUltimaModificacao());
  }
}
```

Para esse exemplo, será criado um arquivo com esses arquivos chamado `transforma.jar`. O jar que precisaria ser criado conteria os seguintes arquivos:

- `PreMain` — classe que é executada para adição dos transformadores (essa classe não pode estar no pacote default).
- `TransformadorUltimaModificacao` — classe que é chamada no carregamento da classe permitindo que o bytecode seja transformado.
- `UltimaModificacaoAdaptadorClasse` — classe que contém a lógica de adição de bytecode na classe.
- `UltimaModificacaoAdaptadorMetodo` — classe que contém a lógica de adição de bytecode nos métodos setter.
- `META-INF/MANIFEST.MF` — arquivo que contém a indicação da classe com o método `premain()`, no seguinte formato `Premain-Class: pacote.PreMain`.

Por fim, para utilizar o componente criado, como foi dito anteriormente, é preciso colocar o arquivo, assim como as dependências do ASM, no classpath da aplicação. Adicionalmente, é preciso executar a máquina virtual com o parâmetro `-javaagent:transforma.jar`, para que o carregamento das classes

seja interceptado. A listagem a seguir apresenta um código que pode ser utilizado para testar se realmente a classe foi alterada. Vale ressaltar que, devido ao filtro criado, a classe `Teste` precisa estar no pacote `org.casadocodigo`.

Exemplo de utilização da classe transformada:

```
public static void main(String[] args){
    Teste t = new Teste();
    t.setProp("teste");
    UltimaAtualizacao ut = (UltimaAtualizacao)t;
    System.out.println(ut.getUltimaModificacao());
  }
```

Esse segundo exemplo ilustra a prática da manipulação de bytecode sendo utilizada em tempo de carregamento para alteração de classes. O uso dessa técnica nesse tipo de cenário é indicada para a inserção de características gerais nas classes. Isso também pode ser uma alternativa para casos em que é necessário alterar alguma classe cujo código-fonte não está disponível. Uma aplicação disso é para instrumentação de código, ou seja, adição de código extra para realizar medições na aplicação em tempo de execução.

9.6 CONSIDERAÇÕES FINAIS

Este capítulo apresentou uma técnica bem avançada que pode ser utilizada em aplicações Java para transformação e criação de novas classes. Enquanto em linguagens dinâmicas isso é algo trivial de ser feito, em linguagens estaticamente tipadas isso deve ser feito com bastante cuidado. Porém, é uma técnica que, se utilizada no cenário correto, permite realizar certas "mágicas" que podem gerar um grande diferencial para o projeto desenvolvido.

Apesar de existirem outras ferramentas que podem ser utilizadas para a manipulação de bytecode, esse capítulo focou apenas no ASM, devido à sua facilidade de uso e maior utilização. Foi visto, como se pode manipular o bytecode sem conhecer a fundo os comandos, através da ASMificação, e três diferentes momentos em que a manipulação pode acontecer: estaticamente durante a construção, para geração de novas classes em tempo de execução e para a transformação de classes no momento do carregamento. O capítulo finalizou apresentando dois exemplos completos, em dois cenários distintos, onde essa técnica é aplicável.

Capítulo 10

REFLEXÃO NO JAVA 8

"A coisa mais importante que alguém já fez por mim foi me expor a novidades." - Temple Grandin

O Java 8 foi lançado e com ele vieram várias novidades para reflexão e anotações. O grande carro-chefe nessa nova versão da linguagem foram as funções lambda, o que de certa forma ofuscou um pouco outras novas funcionalidades interessantes. Para os adeptos do uso da reflexão e anotações, o que acredito ser o caso do leitor que chegou a essa altura do livro, existem grandes novidades que podem facilitar bastante a vida do desenvolvedor, e até permitir novas aplicações para o uso de anotações.

O objetivo deste capítulo é apresentar as novas funcionalidades do Java 8 em relação a reflexão e anotações. Para começar serão apresentadas duas funcionalidades simples, mas que simplificam bastante algumas questões: a recuperação de nomes de parâmetros e as anotações múltiplas. Para esses casos, exemplos apresentados previamente no livro serão revisitados considerando a utilização dessas novas funcionalidades. Em seguida, serão vistas as anotações de tipo, que permitem que anotações sejam adicionadas em um qualquer declaração de tipo. Nesse caso, será apresentada a ferramenta Checker Framework e como ela pode ser utilizada para fazer verificações em tempo de compilação. Finalmente, como não

poderia faltar, serão mostradas as expressões lambda, e será discutido em que situações podem ser utilizadas no lugar de instâncias de `Method`.

10.1 ACESSANDO NOMES DE PARÂMETROS

Uma funcionalidade que desejei desde que comecei a criar componentes que utilizavam reflexão é a recuperação do nome de um parâmetro. Da mesma forma que podemos utilizar convenções de nomenclatura para relacionar uma classe ou um método com outras informações, isso nunca foi possível com os parâmetros, pois seu nome não estava disponível em tempo de execução. Como visto na seção *Anotações em parâmetros*, uma forma de contornar essa situação era a adição de anotações nos parâmetros que identificavam seus nomes. Essa prática é utilizada em APIs como o JAX-RS, que mapeia os parâmetros de um método para os de um serviço web.

No Java 8, agora é possível que os nomes dos parâmetros sejam armazenados no bytecode da classe para serem carregados e estarem disponíveis em tempo de execução. Foi introduzida uma nova classe chamada `Parameter`, que concentra os metadados de um parâmetro. A partir de uma instância dela é possível recuperar não somente o nome do parâmetro, mas também outras informações como anotações, tipo, modificadores, se é um parâmetro do tipo varargs etc. Isso certamente é bem melhor do que a abordagem anterior, na qual a classe `Method` disponibilizava métodos que retornavam essas informações como um array, sendo um elemento para cada parâmetro, como nos métodos `getParameterTypes()` e `getParameterAnnotations()`.

Para mostrar essa nova abordagem, vamos revisitar o exemplo da seção *Anotações em parâmetros*, no qual os parâmetros de um método são recuperados em um mapa a partir de seus nomes. Na versão inicial, eram utilizadas anotações para dar nomes aos parâmetros, visto que não se tinha acesso a essas informações diretamente. Nessa nova versão, apresentada na próxima listagem, os próprios nomes dos parâmetros são utilizados como base. Observe que inicialmente é utilizado o método `getParameters()` para obter um array da classe `Parameter`, de onde o nome é acessado através de `getName()`.

Utilizando o nome dos parâmetros para identificar o valor que será passado para ele:

```
public static Object invocarMetodo(Method m, Object obj,
                    Map<String,Object> info) throws Exception{
    Parameter[] params = m.getParameters();
    Object[] paramValues = new Object[params.length];

    for(int i=0; i<paramValues.length; i++){
        Parameter param = params[i];
        paramValues[i] = info.get(param.getName());
    }
    return m.invoke(obj, paramValues);
}
```

A próxima listagem exemplifica como esse método seria utilizado. O método `metodo()` define dois parâmetros, sendo um chamado `inteiro` e o outro chamado `texto`. O método `main()` define um mapa com diversos valores, os quais são passados como parâmetro para `invocarMetodo()`. De acordo com sua implementação, ele deve passar como parâmetro os valores do mapa com a chave igual ao nome do parâmetro.

Exemplo de utilização da implementação de

invocarmetodo():

```
public class ParametroMetodo {

    public static void main(String[] args) throws Exception {
        Map<String,Object> info = new HashMap<>();
        info.put("inteiro", 13);
        info.put("numero", 23);
        info.put("string", "OK");
        info.put("texto", "NOK");

        ParametroMetodo pm = new ParametroMetodo();
        Method m = pm.getClass().getMethod("metodo",
                    Integer.class, String.class);
        invocarMetodo(m, pm, info);
    }

    public void metodo(Integer inteiro, String texto){
        System.out.println("Parametro inteiro = "+inteiro);
        System.out.println("Parametro texto = "+texto);
    }
}
```

Ao executar o código, é uma surpresa ao observar que ele não funciona! O motivo disso é que os parâmetros não são armazenados no bytecode por default, e, nesse caso, o nome dos parâmetros retorna os valores `argo` e `arg1`, de acordo com sua posição. Para que isso seja feito, é necessário passar a diretiva `-parameter` para o `javac` na hora de compilar o código. No Eclipse, existe essa opção nos parâmetros de compilação, conforme mostra a figura adiante. A principal justificativa dada para isso é o aumento do tamanho do arquivo com o bytecode. Pessoalmente, acho que poderia ser o contrário, e isso poder ser desabilitado com um parâmetro. O não armazenamento dessas informações por definição certamente dificultará a aplicação de componentes que usam reflexão em classes de terceiros, o que não permite que se confie na presença dessa informação.

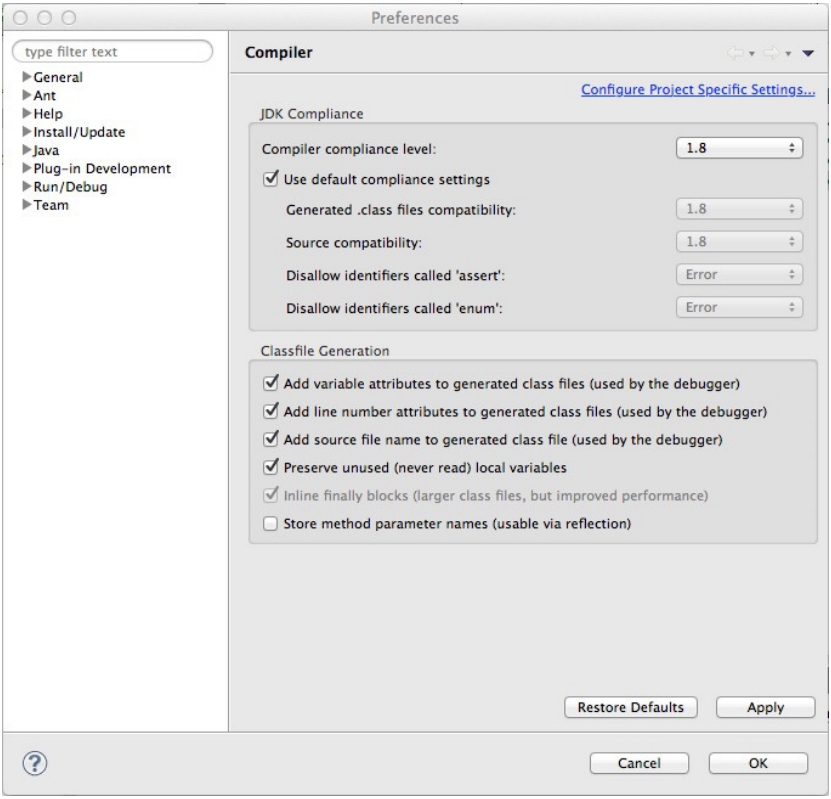

Figura 10.1: Janela com parâmetros de compilação do Eclipse para o Java 8

10.2 MÚLTIPLAS ANOTAÇÕES

Uma outra limitação na definição e recuperação de metadados é a impossibilidade de ter mais de uma anotação do mesmo tipo em um elemento. Existem diversos frameworks que precisam disso, o que faz desse tipo de definição de metadados uma necessidade. Um exemplo famoso é a definição de *named queries* na API JPA, visto que é preciso poder definir mais de uma consulta desse tipo em uma classe.

A solução adotada até então pelos frameworks era a utilização de uma anotação que possui um array de anotações do outro tipo, que foi chamada na seção *Mais de uma anotação do mesmo tipo* de anotação vetorial. Veremos agora é possível incluir mais de uma anotação em um mesmo elemento, porém a estrutura do bytecode para as anotações continua a mesma, visto que ainda é necessária a criação da anotação vetorial, apesar de ela não ser utilizada diretamente na classe.

Para exemplificar a implementação e utilização dessa funcionalidade, será desenvolvido um proxy dinâmico que adiciona valores default para os parâmetros caso o valor passado seja nulo. Para configurar esses valores, serão adicionadas anotações do tipo `@ValorDefault` no método, podendo ser mais de uma. Essas anotações referenciam o nome do parâmetro e o valor default. A próxima listagem apresenta a definição dessa anotação. Observe que foi adicionada a `@Repeatable` que referencia uma outra anotação chamada `@Valores`.

Anotação que poderá ser repetida:

```
@Repeatable(Valores.class)
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ValorDefault {
    String parametro();
    String valor();
}
```

A listagem a seguir apresenta a anotação `@Valores` que, como pode ser visto, nada mais é do que uma anotação vetorial como foi visto na seção *Mais de uma anotação do mesmo tipo*. Ela precisa ser criada, porém ela não será adicionada diretamente na classe. O que acontece é que, quando o código for compilado, essa anotação será

introduzida "na surdina" no bytecode da classe, gerando o mesmo código que seria gerado quando era necessário adicionar a anotação vetorial diretamente. Ela precisa possuir a mesma retenção e o mesmo alvo da anotação original.

Anotação vetorial que precisa ser criada, mas não adicionada na classe:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Valores {
    ValorDefault[] value();
}
```

A seguir, pode ser visto o código do proxy dinâmico que verifica se o parâmetro é nulo e, nesse caso, adiciona o valor default caso este esteja configurado. Inicialmente, as anotações do tipo `@ValorDefault` são recuperadas através do método `getAnnotationsByType()`, o qual é um método novo na API de reflexão. A partir desse método, é possível recuperar todas as anotações de um mesmo tipo configuradas em um elemento.

Proxy dinâmico que insere os valores default nos parâmetros de valor nulo:

```
public class ValorDefaultProxy implements InvocationHandler {

    public static Object criarProxy(Object obj){
        return Proxy.newProxyInstance
              (obj.getClass().getClassLoader(),
                obj.getClass().getInterfaces(),
                new  ValorDefaultProxy(obj));
    }

    private Object obj;
    public ValorDefaultProxy(Object obj) {
        this.obj = obj;
    }
```

```
    public Object invoke(Object proxy, Method method,
     Object[] args) throws Throwable {
        Map<String,String> valores = new HashMap<>();
        for(ValorDefault v : method.getAnnotationsByType(
         ValorDefault.class)){
            valores.put(v.parametro(), v.valor());
        }
        Parameter[] params = method.getParameters();
        for(int i=0; i<args.length; i++){
            if(args[i] == null){
                args[i] = valores.get(params[i].getName());
            }
        }
        return method.invoke(obj, args);
    }
}
```

Observe, em seguida, que a partir dos dados da anotação é criado um mapa, cuja chave é o nome do parâmetro e o valor é o valor default a ser utilizado. Em seguida, são recuperados os parâmetros, a partir da nova API mostrada na seção anterior. A partir dessa informação, é feita uma busca nos parâmetros do método, substituindo os valores nulos pelos configurados nas anotações. Vale ressaltar que a implementação como está irá funcionar apenas para parâmetros do tipo `String` .

A próxima listagem exemplifica a adição das anotações repetidas em um método. O método `ImpressaoInterface` define o método `imprimirDados()` e possui três anotações do tipo `@ValorDefault` . Vale ressaltar que essa sintaxe não era válida antes do Java 8.

Interface com a configuração de anotações repetidas em um método:

```
public interface ImpressaoInterface {

    @ValorDefault(parametro="nome", valor="Fulano")
```

```
    @ValorDefault(parametro="sobrenome", valor="da Silva")
    @ValorDefault(parametro="estadoCivil", valor="solteiro")
    public void imprimirDados(String nome, String sobrenome,
                              String estadoCivil);
}
```

Por fim, a próxima listagem exemplifica a utilização do proxy dinâmico criado. É definida a classe `ImpressaoInfo` que implementa a interface com as anotações. Vale ressaltar que a instância de `Method` recebida no método `invoke()` se refere à interface, e se desejarmos pegar essas informações do método da classe é preciso recuperá-la do objeto e buscar o método com as mesmas informações. Observe que o método `imprimirDados()` apenas imprime os parâmetros recebidos, e o método `main()` cria a instância dessa classe, encapsulando-a com o proxy dinâmico, e invoca seu método duas vezes com parâmetros nulos. Ao executar, devem ser impressos os parâmetros, sendo que os valores configurados nas anotações serão impressos no lugar dos valores nulos.

Exemplo de utilização do proxy dinâmico:

```
public class ImpressaoInfo implements ImpressaoInterface {

    @Override
    public void imprimirDados(String nome, String sobrenome,
            String estadoCivil) {
        System.out.println("Nome: "+nome);
        System.out.println("Sobrenome: "+sobrenome);
        System.out.println("Estado Civil: "+estadoCivil);
    }
    public static void main(String[] args){
        ImpressaoInterface i = (ImpressaoInterface)
             ValorDefaultProxy.criarProxy(new ImpressaoInfo());
        i.imprimirDados(null, "Peixoto", "Casado");
        i.imprimirDados("Pedro", null, null );
    }
}
```

Para os desenvolvedores adeptos de reflexão e que entendem a estrutura em que isso é armazenado no bytecode, é interessante entender o que acontece debaixo dos panos. Na verdade, quando o código é compilado, as anotações repetidas no mesmo elemento são substituídas pela anotação cuja classe está configurada em `@Repeatable`. A próxima listagem mostra o código equivalente ao que é gerado pelas anotações repetidas. Observe que é exatamente a solução com a anotação vetorial mostrada na seção *Mais de uma anotação do mesmo tipo*. É por isso que a anotação `@Valores` precisa existir, mesmo não sendo explicitamente configurada na classe.

Configuração de anotações equivalente a utilizada com anotações múltiplas:

```
public interface ImpressaoInterface {

    @Valores({
        @ValorDefault(parametro="nome", valor="Fulano"),
        @ValorDefault(parametro="sobrenome", valor="da Silva"),
        @ValorDefault(parametro="estadoCivil",
                      valor="solteiro")})
    public void imprimirDados(String nome, String sobrenome,
                              String estadoCivil);
}
```

Se você é curioso e quer tirar a prova de se realmente é isso que acontece, você pode realizar a chamada `isAnnotationPresent()` passando a anotação `@Valores` como parâmetro. O resultado será `true` pois, como foi dito, essa anotação é adicionada. Vendo dessa forma, parece que essa nova feature é mais um açúcar sintático, porém tem o seu valor. Na minha opinião, o principal ganho é relacionado à diminuição de verbosidade na configuração das anotações, visto que agora elas podem ser repetidas diretamente no elemento. Outro ganho, que na verdade poderia

ser substituído por uma função auxiliar, é a adição na API de reflexão de métodos para recuperar anotações desse tipo, no caso o `getAnnotationsByType()`.

10.3 ANOTAÇÕES DE TIPO

A grande novidade em termos de anotações no Java 8 são as anotações de tipo. No capítulo *Metadados e anotações* foram vistos os tipos de elementos que podem ser anotados. Em resumo, esses elementos são declarações de tipo (classes, interfaces, enums), métodos, construtores, atributos, variáveis locais, pacotes e anotações. Para cada um desses tipos de elemento, existe um valor do enum `ElementType` que deve ser configurado na anotação `@Target`.

No Java 8, dois novos valores foram adicionados no enum `ElementType`, significando que existem dois novos locais onde anotações podem ser adicionadas. Esta seção irá mostrar como criar essas anotações e exemplificar a aplicação delas para verificações estáticas e para estender o sistema de tipo da linguagem.

Criando anotações de tipo

As anotações de tipo podem ser adicionadas em qualquer ponto do código onde um tipo é utilizado. Diferentemente das definidas com `ElementType%.TYPE`, que podem ser utilizadas nas declarações de tipo, o valor `ElementType.TYPE_USE` permite anotações em que esse tipo é utilizado. Isso inclui declarações de variáveis, construtores, herança e, até mesmo, casting de variáveis. Já o tipo de elemento declarado com

`ElementType.TYPE_PARAMETER` deve ser utilizado para permitir a adição da anotação dentro dos parâmetros de tipo de um tipo genérico.

A listagem a seguir exemplifica como uma anotação de tipo pode ser criada. Como foi dito, é preciso definir como `@Target` os valores `TYPE_USE`, `TYPE_PARAMETER` ou ambos. As anotações com essas configurações podem ser adicionadas nos tipos.

Definição de uma anotação de tipo:

```
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface AnotacaoTipo {
}
```

A próxima listagem exemplifica o uso de anotações de tipo em usos relacionados a variáveis. Nessa classe, o atributo `var` recebe a anotação no seu tipo `String`. Observe que quem está recebendo a anotação não é o atributo, mas o tipo do atributo. Dentro do método `main()` são exemplificados mais dois usos desse tipo de anotação, sendo um na invocação do construtor, para realizar o cast e com operador `instanceof`. Vale ressaltar, que apesar de as anotações poderem ser adicionadas no `instanceof`, elas não serão consideradas para verificar se o valor é uma instância do tipo anotado.

Anotações de tipo em tipos de variáveis e casts:

```
public class ExemploClasse {

    public static @AnotacaoTipo String var;

    public static void main(String[] args){
        Object valor = new @AnotacaoTipo String("valor");
        var = (@AnotacaoTipo String) valor;
        boolean eDoTipo = valor instanceof @AnotacaoTipo String;
```

```
    }
}
```

A listagem a seguir exemplifica a utilização de anotações de tipo em declarações. Na declaração da classe, podem ser adicionadas anotações nas interfaces implementadas e na classe estendida. Já em uma declaração de método, as anotações podem ser adicionadas ao tipo do retorno, dos parâmetros e das exceções lançadas.

Anotações de tipo em declarações de métodos e classes:

```
public class ExemploDeclaracoes
                extends @AnotacaoTipo Thread
                implements @AnotacaoTipo Runnable{

    public @AnotacaoTipo String exec(
            @AnotacaoTipo String s)
            throws @AnotacaoTipo Exception{
        return "valor";
    }
}
```

Finalmente, a próxima listagem apresenta o exemplo de uso das anotações de tipo dentro de parâmetros genéricos. Vale ressaltar que esse uso das anotações de tipo é que exige o `@Target` com o valor `TYPE_PARAMETER`. Para os que já achavam a sintaxe dos tipos genéricos verbosa, tenho certeza de que a adição de anotações nos parâmetros de tipo vai dar arrepio em vários desenvolvedores...

Anotações de tipo em parâmetros de tipos genéricos:

```
public class ExemploGenerics {
    List<@AnotacaoTipo String> lista;
    Map<@AnotacaoTipo Integer, String> mapa;
}
```

Consumindo anotações de tipo

Na verdade, a adição de anotações de tipo no Java 8 não foi motivada pelo seu uso com a API de reflexão. Sua principal motivação foi fornecer metadados para que fosse possível a realização de processamento estático em tempo de compilação. O exemplo mais popular é o da anotação `@NonNull` que, se adicionada a um tipo, permite a validação de se existe a possibilidade da variável com o tipo receber o valor nulo. Com esse tipo de anotação, é possível ter sistemas de tipos plugáveis, que podem ser utilizados em aplicações para forçar o uso de determinadas regras e fazer com que erros que ocorrem em tempo de execução possam ser pegos durante a compilação.

Como sabemos, as anotações apenas adicionam informações e, sem algum componente para fazer essa verificação, nenhuma regra será imposta ao código que utilizar essas anotações. Infelizmente, apesar de o Java 8 adicionar esse tipo de anotação na linguagem, a JDK 1.8 não traz nenhum componente ou processador que as utiliza. O projeto Checker Framework, cujos participantes foram bastante ativos na JSR que definiu esse novo tipo de anotação, possui diversos conjuntos anotações com seus respectivos processadores, que fazem uso das anotações de tipo. Esse tipo de utilização de metadados em tempo de compilação não está no escopo desse livro.

Checker Framework

O Checker Framework (http://types.cs.washington.edu/checker-framework/) é um projeto que tem sido uma grande referência no uso das anotações de tipo. Ele possui a implementação de diversos processadores com respectivas anotações com diferentes aplicações. Alguns exemplos são verificações para valores nulos, evitando as famosas `NullPointerException` , para objetos não-modificáveis, SQL injection, erros de concorrência, entre outros. Além dos processadores que podem ser plugados facilmente ao compilador Java, o projeto ainda disponibiliza um plugin para o Eclipse que simplifica sua introdução em projetos reais. Além dos processadores próprios, o projeto ainda provê uma estrutura para a criação de processadores personalizados. Outra facilidade é a definição de novos sistemas de tipos plugáveis, apenas com a definição de anotações, sem criação de código adicional.

Apesar da grande aplicação desse tipo de anotação não ser o consumo através de reflexão, em alguns casos é possível recuperá-las caso possuam visibilidade em RUNTIME. Esses casos são aqueles em que a anotação de tipo é utilizada dentro de alguma declaração, como a de uma classe, um método ou um atributo. A próxima listagem mostra como a anotação de tipo `@AnotacaoTipo` poderia ser recuperada respectivamente de seu uso nas classes `ExemploClasse` e `ExemploDeclaracoes` mostradas na seção anterior.

Recuperando anotações de tipo com reflexão:

```
Class<?> c1 = ExemploClasse.class;
System.out.println(c1.getField("var").
        getAnnotatedType().getAnnotations()[0]);

Class<?> c2 = ExemploDeclaracoes.class;
System.out.println(c2.getAnnotatedInterfaces()[0].
        getAnnotations()[0]);
System.out.println(c2.getAnnotatedSuperclass().
        getAnnotations()[0]);

Method m = c2.getMethod("exec", String.class);
System.out.println(m.getAnnotatedReturnType().
        getAnnotations()[0]);
System.out.println(m.getAnnotatedExceptionTypes()[0].
        getAnnotations()[0]);
System.out.println(m.getParameters()[0].getAnnotatedType().
        getAnnotations()[0]);
```

Foram adicionados na API Reflection diversos métodos no formato `getAnnotated???()`, como pôde ser visto no exemplo de código apresentado. Esses métodos retornam uma instância da classe `AnnotatedType`, que possui métodos para a recuperação das anotações de tipo, como `getAnnotations()`, `getAnnotationByType()`, `isAnnotationPresent()`, e todos providos pela interface `AnnotatedElement`.

Exemplo de uso de anotação de tipo

Hoje, é difícil dizer com certeza a aplicabilidade das anotações de tipo em componentes que utilizam reflexão, pois como o Java 8 acabou de ser lançado, ainda não existem muitos casos reais nos quais nos basearmos. Porém, uma aplicação que enxergo para esse tipo de anotação seria a configuração de um metadado na associação de dois elementos de código. Por exemplo, na API do Java EE existe uma anotação `@ApplicationException` que pode

ser adicionada em uma classe de exceção. Dessa forma, será sempre o mesmo metadado para qualquer uso dessa classe. Caso a anotação fosse adicionada na declaração do método que a lançasse, além de possibilitar que ela fosse tratada diferente em diferentes métodos, isso também possibilitaria configurar o efeito em classes já existentes.

Para exemplificar essa utilização, será desenvolvido um proxy dinâmico que deve loggar (que para simplificar iremos apenas imprimir no console) as exceções marcadas no método através de uma anotação de tipo. Para deixar o exemplo mais interessante, esse proxy dinâmico irá atuar somente nas interfaces marcadas na classe com a anotação de tipo `@Logged`, apresentada na próxima listagem. Vale ressaltar que essa anotação deverá ser adicionada antes do nome da interface na declaração `implements`.

Anotação de tipo que configura a interface que deverá ser interceptada pelo proxy dinâmico:

```
@Target(ElementType.TYPE_USE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Logged {
}
```

A listagem a seguir mostra o método `criarProxy()` que inclui no proxy dinâmico somente as interfaces das quais ele implementa que possuem a anotação `@Logged`. Observe que, através do método `getAnnotatedInterfaces()`, as interfaces implementadas pela classe são retornadas em instâncias de `AnnotatedType`. A partir dessa instância, é possível verificar se anotação está presente e, em caso positivo, adicioná-la em uma lista. Em seguida, essa lista é transformada em um array do tipo `Class` e só então o proxy dinâmico é criado.

Incluindo apenas as interfaces marcadas no proxy:

```
public static Object criarProxy(Object obj){
    List<Class<?>> marcadas = new ArrayList<>();
    for(AnnotatedType t : obj.getClass()
     .getAnnotatedInterfaces()){
        if(t.isAnnotationPresent(Logged.class)){
            marcadas.add((Class<?>)t.getType());
        }
    }
    Class<?>[] marcadasArray = new Class<?>[marcadas.size()];
    marcadasArray =
        Arrays.copyOf(marcadas.toArray(), marcadas.size(),
            marcadasArray.getClass());
    return Proxy.newProxyInstance (
        obj.getClass().getClassLoader(), marcadasArray,
        new ExceptionLogger(obj));
}
```

Na próxima listagem é apresentada a anotação de tipo `@DoLog` que deve ser adicionada somente nas exceções de tipo que devem ser logadas pelo proxy. Elas devem ser adicionadas na cláusula `throws`, logo antes do nome da exceção que está anotando.

Anotação de tipo para marcar as exceções que devem ser logadas:

```
@Target(ElementType.TYPE_USE)
@Retention(RetentionPolicy.RUNTIME)
public @interface DoLog {
}
```

A listagem a seguir mostra o código do proxy dinâmico `ExceptionLogger`. Observe que a primeira coisa que ele faz é recuperar o método da própria classe, pois o método recebido no `invoke()` possui as declarações da interface. Isso irá permitir que as anotações sejam adicionadas na própria classe que a implementa, e não na interface que define o método.

Proxy dinâmico que imprime no console as exceções anotadas:

```
public class ExceptionLogger implements InvocationHandler {

    private Object obj;

    public ExceptionLogger(Object obj) {
        this.obj = obj;
    }
    public Object invoke(Object proxy, Method m, Object[] args)
            throws Throwable {
        m = obj.getClass().getMethod(m.getName(),
            m.getParameterTypes());
        try {
            return m.invoke(obj, args);
        } catch (InvocationTargetException e) {
            Throwable erro = e.getTargetException();
            AnnotatedType[] excps =
                m.getAnnotatedExceptionTypes();
            for(AnnotatedType excp : excps){
                Class<?> excpType = (Class<?>) excp.getType();
                if(excpType.isInstance(erro) &&
                    excp.isAnnotationPresent(DoLog.class)){
                    System.out.println("LOGGER:"
                                       + erro.getMessage());
                }
            }
            throw erro;
        }
    }
}
```

Em seguida, caso a invocação do método lance uma exceção do tipo `InvocationTargetException`, a qual significa que o próprio método lançou a exceção, ela é recuperada por um bloco `catch`. A exceção original ocorrida é recuperada utilizando o método `getTargetException()`. Como próximo passo, as exceções declaradas pelo método são recuperadas utilizando `getAnnotatedExceptionTypes()`. Então, itera-se sobre essas

exceções buscando uma que possua a anotação `@DoLog` e da qual a exceção lançada seja instância. Se encontrada, a mensagem de erro é impressa no console.

A listagem a seguir exemplifica como uma classe seria anotada para ser interceptada pelo proxy desenvolvido. Observe que a classe implementa duas interfaces, porém somente a `Execucao` possui a anotação `@Logged`, e somente os métodos dela serão interceptados pelo proxy. Adicionalmente, o método `exec()`, dessa interface, também declara duas exceções, mas apenas a `SecurityException` possui a anotação `@DoLog`. Sendo assim, ela será impressa no console caso seja lançada, o que não ocorrerá com a `IOException` que não possui a anotação.

Exemplo de classe anotada para o proxy dinâmico ExceptionLogger:

```
public class Teste implements @Logged Execucao, OutraInterface{

    @Override
    public void exec(String s) throws @DoLog SecurityException,
    IOException {
        //implementação
    }
}
```

Para os desenvolvedores que gostariam de explorar mais a fundo as possibilidades das anotações de tipo com reflexão, sugiro tentar se aprofundar nesse exemplo adicionando outras características. Além de integrar com uma API de logging de verdade, a anotação `@DoLog` poderia conter informações relacionadas à severidade do erro e filtros baseados na mensagem e/ou em subclasses para decidir se seu registro é necessário ou não. Adicionalmente, a anotação `@Logger` poderia definir

configurações default que poderiam ser sobrescritas pela anotação na declaração do método.

Mas essas informações não poderiam estar em anotações no método ou na classe?

Sim! É exatamente assim que é feito hoje sem as anotações de tipo. A diferença é que, se eu adicionar uma informação na classe sobre uma interface que ela implementa, eu preciso referenciar essa interface. De forma similar, é possível adicionar uma anotação no método referenciando uma de suas exceções. Tudo é uma questão de onde e como os metadados são definidos, pois é possível também definir tudo em um documento XML e referenciar todos os elementos. A vantagem das anotações de tipo é que elas permitem adicionar os metadados diretamente aos elementos que referenciam, tornando essa ligação mais direta e evitando erros no referenciamento.

10.4 REFLEXÃO E EXPRESSÕES LAMBDA

Ao ficar sabendo que o Java 8 viria com expressões lambda e referências estáticas a métodos logo fiquei interessado em saber como a API de reflexão lidaria com isso. Ao estudar um pouco mais a fundo como essa nova funcionalidade funciona, fiquei decepcionado ao ver que, na verdade, para a API de reflexão nada mudou. Minha esperança de que seria possível passar referências estáticas de métodos em atributos de anotações também foi por

água abaixo.

Esta seção irá abordar de forma rápida o funcionamento das expressões lambda no Java 8, apresentando a nova sintaxe e formas de utilização desse tipo de expressão. Em seguida, faremos alguns experimentos em relação a como a API de reflexão enxerga a passagem de lambdas e referências de método como parâmetro.

Uma rápida introdução aos lambdas

Em Java, como não é possível passar métodos como parâmetro, quando desejamos passar um comportamento, definimos uma interface e passamos uma implementação dela. Um exemplo é a interface `Runnable` da API de concorrência, na qual uma instância na verdade define um procedimento a ser executado em paralelo por uma `Thread`. Outro exemplo é a interface `Comparator`, que serve para definir um critério de comparação que pode ser utilizado, por exemplo, na ordenação de uma coleção. Nesses dois casos, precisa-se apenas de um método, mas como ele não existe sozinho, ele é colocado dentro de uma interface.

Considere o método `forEach()` de uma coleção, que recebe como parâmetro a instância de uma implementação da interface `Consumer`. Esse método chama o método `accept()` dessa classe para cada um de seus elementos. Uma forma de fazer isso para definir a lógica executada no próprio método é utilizando classes anônimas. Utilizando esse recurso, usa-se o operador `new` com o nome da interface e, em seguida, define-se a implementação de seus métodos.

Para exemplificar o uso de classes anônimas, a listagem a seguir mostra o método `forEach()` sendo utilizado duas vezes em uma

lista de instâncias da classe `Produto`. Inicialmente é chamado o método `aumentaPreco()` em cada um dos produtos e em seguida cada um é impresso no console.

Criando classes anônimas com implementação de interfaces:

```
List<Produto> lista = new ArrayList<>();
lista.add(new Produto("arroz", 8.00));
lista.add(new Produto("fejão", 5.00));
lista.add(new Produto("batata", 7.00));

lista.forEach(new Consumer<Produto>() {
    public void accept(Produto p) {
        p.aumentaPreco();
    }
});

lista.forEach(new Consumer<Produto>() {
    public void accept(Produto p) {
        System.out.println(p);
    }
});
```

No Java 8, foi introduzido o conceito de interface funcional. Essas são as interfaces criadas para encapsular uma única operação, ou seja, que possuem apenas um método. Quando um método espera a instância de uma interface desse tipo, é possível definir uma expressão lambda que estabelece o comportamento que seria definido pela implementação daquela interface. A sintaxe é a definição dos parâmetros do método, seguidos de uma `->` e da lógica a ser executada. O tipo do parâmetro pode ser inferido de acordo com a interface.

A listagem a seguir apresenta como ficaria as duas invocações do método `forEach()` utilizando expressões lambda. A expressão `p -> p.aumentaPreco()`, ao ser passada no lugar de um `Consumer`, representa que seu único método, no caso o

`accept()`, vai receber o parâmetro `p`, que no caso é um `Produto`, e vai executar `p.aumentaPreco()`. Caso mais de um comando precise ser definido, pode-se criar um bloco de código entre chaves. Uma estrutura similar é utilizada para o comando que imprime cada um dos elementos.

Utilizando expressões lambda no lugar da interface Consumer:

```
lista.forEach(p -> p.aumentaPreco());
lista.forEach(p -> System.out.println(p));
```

Uma outra opção para passar no `forEach()` é uma referência de método. Um método pode ser referenciado através do nome da classe, seguido de duas vezes dois pontos, seguido do nome do método, neste formato: `Classe<::metodo`. No caso, pode ser um método para ser invocado no objeto que está sendo passado como parâmetro ou um método de um objeto é passado como parâmetro. A seguir, o mesmo código é apresentado utilizando referências de método. Observe que `System.out` é um objeto, do qual está sendo passada uma referência para seu método `println()`.

Utilizando referências de método no lugar da interface Consumer:

```
lista.forEach(Produto::aumentaPreco);
lista.forEach(System.out::println);
```

Vale afirmar que esta seção apenas arranhou as funcionalidades da linguagem Java em relação a funções lambda. O objetivo foi apenas fazer uma introdução ao assunto, mostrando o que são as expressões lambda e as referências de método. Para uma apresentação mais completa sobre o tema, sugere-se a leitura do

livro *Java 8 Prático: Lambdas, Streams e os novos recursos da linguagem* (TURINI, Rodrigo; SILVEIRA, 2014).

Lendo os lambdas com reflexão

Ao procurar saber mais sobre como a API de reflexão entendia as expressões lambda, não encontrei muita documentação. Então, como todo bom desenvolvedor curioso, criei um método que colhe informações sobre o que foi recebido como parâmetro. A listagem abaixo mostra o método estático `quemSouEu()` que recebe um `Consumer` como parâmetro. Observe que ele irá imprimir no console o nome da classe, sua superclasse, suas interfaces, seus métodos e seus atributos.

Método que identifica características da classe recebida como parâmetro:

```
public static void quemSouEu(Consumer<Produto> cons) throws
 Exception{
    Class<?> c = cons.getClass();
    System.out.println("NOME:"+c.getName());
    System.out.println("SUPERCLASSE:"
                       + c.getSuperclass().getName());
    for(Class<?> i : c.getInterfaces()){
        System.out.println("INTERFACE:"+i.getName());
    }
    for(Method m : c.getDeclaredMethods()){
        System.out.println("METODO:"+m.getName());
    }
    for(Field f : c.getDeclaredFields()){
        System.out.println("FIELD:"+f.getType().getName() +
                " "+f.getName());
    }
}
```

O primeiro experimento foi realizar a chamada `quemSouEu(p -> p.aumentaPreco())`, e o resultado impresso no console está

apresentado na próxima listagem. Observe que a classe foi criada pelo compilador em tempo de execução e implementa a interface `Consumer`, possuindo apenas o método `accept()`.

Resultado da chamada quemSouEu(p -> p.aumentaPreco()):

```
NOME:ExemploCollection$$Lambda$1/713338599
SUPERCLASSE:java.lang.Object
INTERFACE:java.util.function.Consumer
METODO:accept
```

O segundo experimento foi chamar o `quemSouEu()` em uma referência de método, no caso `quemSouEu(System.out<::println)`. Aqui, a principal mudança é a existência de um atributo, chamado de `arg$1`, que armazena uma instância de `PrintStream`, que é a classe do objeto passado na referência do objeto, o `System.out`. A partir dessa informação podemos inferir que essa instância será utilizada no método `accept()` para a execução da lógica.

Utilizando referências de método no lugar da interface Consumer:

```
NOME:ExemploCollection$$Lambda$1/821270929
SUPERCLASSE:java.lang.Object
INTERFACE:java.util.function.Consumer
METODO:accept
METODO:get$Lambda
FIELD:java.io.PrintStream arg$1
```

Com isso, vemos que as expressões lambda não adicionam nada de novo em termos da estrutura da API de reflexão. Tanto a expressão lambda quanto a referência de método serão transformados pelo compilador em uma instância da interface funcional relacionada. Por mais que esta seção tenha concluído que nada mudou, é importante saber isso para que se possa

desenvolver algoritmos que utilizam reflexão e que envolvem classes que usam essa nova prática.

Referências de método versus classe Method

É possível na linguagem Java obtermos uma instância de `Class` utilizando `Classe.class` . Porém, não é possível obter uma instância de `Method` utilizando `Classe<::metodo` . É importante lembrar que a referência de método retorna uma instância de uma interface funcional, que será gerada pelo compilador. Essa referência (infelizmente) nada tem a ver com a representação dos métodos na API de reflexão.

10.5 CONSIDERAÇÕES FINAIS

Para os desenvolvedores que utilizam reflexão, é sempre importante procurar saber o que está por trás das mudanças realizadas na linguagem. Muitas vezes elas são apenas sintáticas, não mudando a representação dos elementos da linguagem em tempo de execução. Esse, por exemplo, foi o caso das expressões lambda e das referências de método, que são simplesmente formas simplificadas de utilizar coisas que já existiam.

Este capítulo também explorou outras mudanças na linguagem como a adição do acesso ao nome dos parâmetros, a possibilidade do uso de anotações repetidas em um elemento e as anotações de tipo. Apesar de alguns exemplos práticos terem sido apresentados com essas novas funcionalidades do Java 8, somente o tempo e a

criatividade dos desenvolvedores irá dizer como elas serão utilizadas na prática.

Capítulo 11

TRUQUES DA API DE REFLEXÃO

"A alegria encontra o seu complemento no sucesso." - Immanuel de Roma (Mahberot)

Quando nos propomos a escrever um livro técnico, sabemos a mensagem que queremos passar e as coisas que achamos importante de serem ensinadas. A partir disso, o desafio está em tentar organizar esse conhecimento, agrupando assuntos em capítulos e criando uma sequência de aprendizado que faça sentido. Infelizmente alguns assuntos importantes e interessantes acabam não se encaixando em nenhum lugar, e são justamente esses assuntos que foram agrupados aqui neste capítulo.

Os tópicos que foram colocados aqui são questões complementares à utilização da API de reflexão em Java. São algumas cartas na manga dos desenvolvedores que podem ser utilizadas para fazer coisas bem interessantes. Cada seção pode ser lida de forma independente e trás uma dica complementar no uso da reflexão. Espero que apreciem!

11.1 QUEM ME CHAMOU?

"Onde está você? Me telefona. Me Chama! Me Chama! Me Chama...." - Lobão, Me chama

Uma coisa interessante que é possível fazer em Java é recuperar a pilha de execução do método que está sendo invocado. Em outras palavras, é possível saber que classe chamou o método corrente. Inicialmente, isso era utilizado apenas por exceções para registrar o seu *stack trace*, porém os desenvolvedores começaram a criar exceções somente para recuperar essa informação e obter os métodos que o chamaram. Percebendo esse uso, a partir da JDK 1.5 a classe `Thread` passou a disponibilizar o método `getStackTrace()` que retorna sua pilha de métodos corrente.

A listagem a seguir exemplifica a recuperação da pilha de execução. O método `main()` chama o método `metodoA()`, que por sua vez chama o `metodoB()`. Este último recupera a `Thread` de execução corrente utilizando o método estático `getCurrentThread()` e recupera dele o *stack trace*. Em seguida, através de um loop, é feita a iteração entre os elementos do *stack trace*, imprimindo-os no console.

Exemplo de recuperação do Stack Trace:

```
public class QuemChamou {

    public static void main(String[] args) {
        metodoA();
    }
    public static void metodoA(){
        metodoB();
    }
    public static void metodoB(){
        StackTraceElement[] stes =
            Thread.currentThread().getStackTrace();
        for(StackTraceElement ste : stes){
            System.out.println(
```

```
                ste.getClassName()+ "->" +
                ste.getMethodName()+"()");
        }
    }
}
```

Infelizmente, as informações presentes na classe `StackTraceElement` são muito poucas. No exemplo apresentado, são recuperados o nome da classe com `getClassName()` e o nome do método com `getMethodName()`. Utilizando a chamada `Class.forName()` é possível recuperar a respectiva instância de `Class`, porém, como somente o nome do método está disponível, se houver sobrecarga de métodos, não é possível saber exatamente de que método veio a chamada. As outras informações disponíveis para serem recuperadas em `StackTraceElement` é o arquivo de onde veio a chamada e a respectiva linha de código.

A próxima listagem mostra o que é impresso no console com a execução do código da listagem anterior. Observe que o primeiro método a ser impresso é o próprio `getStackTrace()` e o segundo é o `metodoB()`, que fez a chamada de `getStackTrace()`. Dessa forma, o método que o chamou poderá ser encontrado apenas no terceiro elemento do array retornado.

Saída do programa que imprime o Stack Trace:

```
java.lang.Thread->getStackTrace
quemchamou.QuemChamou->metodoB
quemchamou.QuemChamou->metodoA
quemchamou.QuemChamou->main
```

A informação da classe que fez a chamada pode ser utilizada para que o componente possa recuperar metadados do *stack trace*, adaptando seu comportamento de acordo com as classes que fizeram a invocação. Isso pode ser útil para que essas classes não

precisem ficar passando parâmetros de configuração do componente em diversas chamadas, bastando a adição de uma anotação.

A listagem a seguir mostra como uma anotação pode ser recuperada de uma classe presente no *stack trace*. Como não é possível obter o método com precisão, caso essa estratégia seja utilizada, é melhor que as anotações sejam colocadas nas classes. Observe que a busca de classes começa no quarto elemento do array, sendo o primeiro o método `getStackTrace()`, o segundo, esse próprio método e o terceiro, o método que fez a chamada. Em seguida, a classe é obtida a partir do seu nome e é verificada se a anotação desejada está presente. Caso não esteja, é feita a procura na classe anterior do *stack trace*. Isso é feito até a anotação ser encontrada ou até que se chegue ao fim da pilha de execução.

Busca de anotações em classes do StackTrace:

```
public static Annotation buscaAnotacaoClasseQueChamou(
        Class<? extends Annotation> anot) throws Exception {
    StackTraceElement[] stes =
        Thread.currentThread().getStackTrace();
    for(int i=3; i<stes.length; i++){
        StackTraceElement ste = stes[i];
        Class<?> clazz = Class.forName(ste.getClassName());
        Annotation an = clazz.getAnnotation(anot);
        if(an != null)
            return an;
    }
    return null;
}
```

11.2 RECUPERANDO TIPOS GENÉRICOS COM REFLEXÃO

Os tipos genéricos em Java são validados em tempo de compilação e a informação sobre o tipo genérico de um objeto não é mantida em tempo de execução. Dessa forma, ao chamar o método `getClass()` em um determinado objeto, não é possível saber o tipo genérico associada a essa classe. Porém, em declarações de métodos, atributos ou de classe que utilizam tipos genéricos, é sim possível saber qual o tipo genérico declarado. Essa informação pode ser um metadado muito útil para execução de algumas funcionalidades.

Como exemplo, será criado um algoritmo que recupere todas as entidades relacionadas com uma determinada classe, considerando uma entidade como toda classe com a anotação `@Entity`. Nesse caso, seria necessário varrer os atributos dela e procurar pelos tipos que possuem essa anotação. Dentro desse contexto, considere, por exemplo, a classe `Pessoa` apresentada na listagem a seguir. Nela, além da classe `Endereco`, que é o tipo de um atributo, o atributo `telefones` possui uma entidade declarada como um tipo genérico.

Classe que possui tipos relacionados declarados em tipos genéricos:

```
@Entity
public class Pessoa {

    private String nome;
    private int idade;
    private Endereco endereco;
    private List<Telefone> telefones;
    private Set<String> cargos;

    //getters e setters omitidos
}
```

Na API de reflexão, a classe `Field` possui um método chamado `getGenericType()`, que não retorna os parâmetros de tipo, mas uma instância de `ParameterizedType`, apesar de o retorno ser do tipo `Type` e ser necessário fazer o cast. Essa classe possui um método chamado `getActualTypeArguments()`, que retorna um array com os parâmetros genéricos do tipo.

A listagem a seguir mostra o método `getEntidadesRelacionadas()`, que recebe uma classe e retorna uma lista com as entidades relacionadas a ela. Primeiramente, esse método verifica se a classe do atributo possui a anotação `@Entity` e, em caso positivo, a adiciona à lista que será retornada. Em caso negativo, é verificado se o tipo implementa a interface `Collection`, recuperando nesse caso o tipo genérico, para o qual também é verificado se possui a anotação `@Entity`.

Método que retorna as entidades relacionadas, procurando nos tipos genéricos:

```
public static List<Class> getEntidadesRelacionadas(Class c){
    List<Class> list = new ArrayList<>();
    for(Field f : c.getDeclaredFields()){
        Class tipo = f.getType();
        if(tipo.isAnnotationPresent(Entity.class)){
            list.add(tipo);
        }else if(Collection.class.isAssignableFrom(tipo)){
             ParameterizedType tipoGenerico =
                 (ParameterizedType) f.getGenericType();
             Class parametroGenerico =
                (Class) tipoGenerico.getActualTypeArguments()[0];
             if(parametroGenerico
                .isAnnotationPresent(Entity.class)){
                 list.add(parametroGenerico);
             }
        }
    }
    return list;
```

```
}
```

A recuperação do parâmetro genérico passado para superclasse quando ela é estendida pela classe é um outro cenário onde isso pode ser muito útil. Imagine o exemplo de uma abstração para a representação de uma classe que abstrai as operações de persistência, ou seja, implementa o padrão de projeto DAO. Muitas vezes, é preciso saber qual é a classe que será persistida para implementação dos métodos, principalmente se a implementação for JPA. Dessa forma, a superclasse pode disponibilizar um método para a recuperação do parâmetro genérico configurado na subclasse.

A próxima listagem apresenta um exemplo em que essa estratégia é implementada. A classe `DAO` é abstrata e define o parâmetro genérico `E`. Como normalmente as subclasses a implementam para a persistência de uma classe específica, esse tipo genérico é configurado na declaração da herança. Sendo assim, o método `getTipoGenerico()` é disponibilizado para ser utilizado nas subclasses recuperando o tipo genérico configurado por elas para a superclasse `DAO`. A partir da classe do objeto, é feita uma busca nas superclasses até que `DAO` seja encontrada. Então, o método `getGenericSuperclass()` é utilizado para recuperar uma instância de `ParameterizedType`, a qual é utilizada na recuperação dos parâmetros genéricos.

Exemplo de uma superclasse com tipo genérico que fornece método para recuperação de seu parâmetro genérico:

```
public abstract class DAO<E> {

    public abstract void salvar(E obj);
    public abstract E recuperar(int id);
    public abstract List<E> listarTodos();
```

```
    public Class<?> getTipoGenerico(){
        Class<?> c = this.getClass();
        while(!c.getSuperclass().equals(DAO.class)){
            c = c.getSuperclass();
        }
        ParameterizedType tipoGenerico =
            (ParameterizedType) c.getGenericSuperclass();
        return (Class) tipoGenerico.getActualTypeArguments()[0];
    }
}
```

Sempre que for possível ter uma declaração de tipo que pode receber um tipo genérico, haverá um método com o nome `getGenericSomething()` que irá possibilitar recuperar a instância respectiva de `ParameterizedType`. A classe `Method`, por exemplo, possui os métodos: `getGenericExceptionTypes()` para a recuperação do tipo genérico das exceções; `getGenericParameterTypes()` para a recuperação do tipo genérico dos parâmetros; e `getGenericReturnType()` para a recuperação do tipo genérico do retorno.

11.3 ACESSANDO MEMBROS PRIVADOS

Quando abordamos o acesso a atributos e a invocação de métodos, foi dito que, se tentarmos acessar um membro privado de uma classe, será lançada a exceção `IllegalAccessException`. Porém, existe uma forma de permitir esse acesso aos membros privados de um objeto através da reflexão, e, na verdade, é algo muito simples. Tanto na classe `Method` quanto na classe `Field` existe um método chamado `setAccessible()` que, se for invocado com o valor `true`, habilita o acesso a eles, mesmo que sejam privados.

A listagem a seguir mostra um exemplo de como um atributo privado pode ser acessado através de reflexão. É importante ressaltar que, para fazer sentido o exemplo, a classe `AcessoAtributoPrivado` com o atributo `var` está em um arquivo separado da classe `Principal`. Outro ponto importante é que o acesso ao atributo deve ser feito utilizando `getDeclaredField()` e não o `getField()`, que não recupera membros privados da classe. O método `setAccessible()` é o responsável por desbloquear o acesso do membro privado.

Exemplo de acesso a atributo privado:

```
//classe com atributo privado
public class AcessoAtributoPrivado {
    private int var = 23;
}

//classe que acessa atributo privado com reflexão
public class Principal {
    public static void main(String[] args) throws Exception {
        AcessoAtributoPrivado obj = new AcessoAtributoPrivado();
        Field f =
            AcessoAtributoPrivado.class.getDeclaredField("var");
        f.setAccessible(true);
        System.out.println(f.getInt(obj));
    }
}
```

A listagem a seguir mostra o mesmo procedimento feito para a invocação de métodos. Da mesma forma, o método `setAccessible()` é utilizado para desbloquear o acesso à invocação do método.

Exemplo de acesso a método privado:

```
//classe com método privado
public class AcessoMetodoPrivado {
    private String executar(){
```

```
        return "OK";
    }
}

//classe que acessa método privado com reflexão
public class Principal {
    public static void main(String[] args) throws Exception {
        AcessoMetodoPrivado obj = new AcessoMetodoPrivado();
        Method m =
        AcessoMetodoPrivado.class.getDeclaredMethod("executar");
        m.setAccessible(true);
        System.out.println(m.invoke(obj));
    }
}
```

Uma informação importante é que o método `setAccessible()` irá desbloquear o acesso a métodos e atributos privados somente via reflexão. O acesso direto aos métodos e atributos privados continuará bloqueado fora da própria classe. É importante ressaltar também que o uso do `setAccessible()` pode ser bloqueado em aplicações que possuam um `SecurityManager`. Por exemplo, se você tentar fazer isso em uma Applet, será lançada uma exceção da classe `java.security.AccessControlException`. Sendo assim, um componente que utiliza esse recurso pode não funcionar em aplicações que executam em ambientes mais controlados.

O fato de esse recurso tornar possível o acesso a membros privados da classe, não quer dizer que isso deva ser feito com frequência. Por exemplo, se a classe disponibiliza um método de acesso público a um atributo, é muito melhor utilizar reflexão para acessá-los do que tentar acessar o atributo diretamente quebrando o encapsulamento da classe. Sendo assim, procure utilizar isso apenas em casos bem específicos, para testar uma classe legada, por exemplo.

11.4 API METHOD HANDLE

Apesar de a linguagem Java e sua máquina virtual já terem evoluído muito desde sua criação, o bytecode executado por ela se manteve durante muito tempo o mesmo. Foi somente no Java 7 que uma nova instrução foi adicionada na máquina virtual. Essa instrução, `invokedynamic`, teve como principal objetivo prover um melhor suporte da máquina virtual a linguagens dinâmicas. O maior problema é que, mesmo que fossem feitas diversas otimizações, o custo da invocação de métodos para linguagens dinâmicas compiladas para a máquina virtual ainda era bem grande. Essa nova instrução permite a invocação de um método sem a obrigatoriedade de conhecer os tipos envolvidos.

Para que os programas em Java também pudessem fazer uso dessa nova instrução, foi criada uma nova API localizada no pacote `java.lang.invoke`. A partir dela é possível criar referências para métodos, permitindo sua invocação de forma mais eficiente. Diferentemente da API Reflection, onde as assinaturas dos métodos são fixas, nessa API as assinaturas dos métodos são polimórficas, possibilitando uma maior flexibilidade. Esta seção irá mostrar como utilizar essa API e as principais diferenças dela para a API tradicional de reflexão.

Utilização do Method Handle

A listagem a seguir mostra um exemplo da sequência básica de utilização da API Method Handle para a invocação de um método. O primeiro passo é a obtenção do tipo do método com a classe `MethodType`. O método construtor `methodType()` recebe como primeiro parâmetro a classe do retorno do método e como

argumentos seguintes, as classes dos parâmetros. Em seguida, deve ser obtida uma instância da classe `Lookup` que é responsável por fazer uma busca nos métodos.

Exemplo básico de utilização do MethodHandle:

```
public class ExemploMH {

    public static void main(String[] args) throws Throwable {
        MethodType methodType = MethodType.methodType(
                String.class, String.class, int.class);
        MethodHandles.Lookup lookup = MethodHandles.lookup();
        MethodHandle methodHandle = lookup.findVirtual(
                ExemploMH.class, "repetir", methodType);

        ExemploMH ex = new ExemploMH();
        System.out.println(methodHandle.invoke(ex, "teste", 5));
    }

    public String repetir(String s, int qtd){
        String resultado = "";
        for(int i=0; i<qtd; i++)
            resultado += s;
        return resultado;
    }
}
```

A classe `Lookup` possui alguns métodos que podem ser utilizados para a localização do método desejado. No exemplo apresentado, é utilizado o método `findVirtual()`, que deve ser empregado para a busca de métodos de instância. Um outro exemplo é o `findStatic()`, que deve ser utilizado para métodos estáticos. Eles irão retornar uma instância de `MethodHandle`, que será utilizada para a invocação. O método `invoke` recebe como parâmetro o objeto onde o método será invocado e seus parâmetros.

Se por acaso já existe uma referência ao método por reflexão

através de uma instância da classe `Method`, também é possível obter o `MethodHandle` a partir dele. A classe `Lookup` possui métodos que permitem "traduzir" as instâncias da API Reflection. A listagem a seguir, por exemplo, mostra o uso do método `unreflect()` para obter o `MethodHandle` a partir de um `Method`.

Transformando de Method para MethodHandle:

```
public class ExemploUnreflect {

    public static void main(String[] args) throws Throwable {
        Method m = ExemploUnreflect.class.getMethod(
                "soma", int.class, int.class);
        MethodHandle methodHandle =
                MethodHandles.lookup().unreflect(m);

        ExemploUnreflect ex = new ExemploUnreflect();
        System.out.println(methodHandle.invoke(ex, 10, 5));
    }
    public int soma(int a, int b){
        return a + b;
    }
}
```

Apesar de o `MethodHandle` representar um método, ele também pode ser utilizado para o acesso a atributos. Os métodos `findGetter()` e `findSetter()` da classe `Lookup` podem ser utilizados para criar um `MethodHandle`, que representa um método de acesso a um atributo, mesmo que aquele atributo não possua um método de acesso na classe. Esses dois métodos recebem como parâmetro a classe do atributo, o seu nome e o seu tipo. A listagem a seguir exemplifica o uso desse recurso criando instâncias de `MethodHandler` para ler e escrever em um atributo.

Acessando atributos com MethodHandle:

```
public class ExemploField {

    private String text = "teste";

    public static void main(String[] args) throws Throwable {
        MethodHandles.Lookup lookup = MethodHandles.lookup();
        MethodHandle getter = lookup.findGetter(
                ExemploField.class, "text", String.class);
        MethodHandle setter = lookup.findSetter(
                ExemploField.class, "text", String.class);

        ExemploField ex = new ExemploField();
        setter.invoke(ex, "outro");
        System.out.println(getter.invoke(ex));
    }
}
```

Uma característica interessante dessa nova API é a possibilidade de modificar uma instância de `MethodHandle`, encadeando outras instâncias dessa classe. Dessa forma, é possível modificar um `MethodHandle`, por exemplo, transformando parâmetros ou adicionando um tratador de erros. Imagine, por exemplo, que se espere como parâmetro um `MethodHandler` que receba como parâmetro uma `String`. Nesse caso, instâncias de `MethodHandler` poderiam ser criadas a partir de métodos com outras assinaturas, adaptando-as à que é esperada pelo método, seja transformando o valor ou fixando algum outro parâmetro. Esse tipo de flexibilidade não existe na API tradicional de reflexão com a classe `Method`. A listagem a seguir mostra um exemplo em que instâncias de `MethodHandler` são encadeadas para fixar o objeto onde o método é invocado e um de seus parâmetros.

Encadeando MethodHandle:

```
public class ExemploCadeia {

    public static void main(String[] args) throws Throwable {
        MethodType methodType = MethodType.methodType(
```

```
                String.class, String.class, int.class);
        MethodHandles.Lookup lookup = MethodHandles.lookup();
        MethodHandle methodHandle = lookup.findVirtual(
                ExemploMH.class, "repetir", methodType);
        methodHandle = MethodHandles.insertArguments(
                methodHandle, 2, 5);
        methodHandle = MethodHandles.insertArguments(
                methodHandle, 0, new ExemploMH());

        System.out.println(methodHandle.invoke("teste"));
    }
    public String repetir(String s, int qtd){
        String resultado = "";
        for(int i=0; i<qtd; i++)
            resultado += s;
        return resultado;
    }
}
```

A classe `MethodHandles`, observe que com "s" no final, possui diversos métodos que podem ser utilizados para modificar um `MethodHandle` existente, retornando um novo `MethodHandle` modificado, que irá encapsular a invocação do original. Na listagem, foi utilizado o método `insertArguments()` para fixar os parâmetros do método `invoke()`. Observe que a primeira chamada a esse método fixa o segundo parâmetro como 5 e a segunda fixa o objeto onde ele será invocado.

Controle de acesso aos métodos

Um outro diferencial do `MethodHandle` é em relação ao controle de acesso. Enquanto com a API Reflection é verificado a cada vez se o local de invocação pode ter acesso ao método ou atributo, com a API do Method Handle essa verificação é feita apenas uma vez no local da sua criação. Sendo assim, é possível ter acesso a membros privados da classe, sem nenhuma restrição do `SecurityManager`, contando que eles sejam criados dentro da

classe.

Para exemplificar essa diferença, considere a classe `ClasseAcessada` apresentada na próxima listagem. Ela possui um atributo privado chamado `atributo` e um método que cria um `MethodHandler` do tipo getter para o acesso a esse atributo.

Classe com atributo privado que provê um MethodHandle para seu acesso:

```
public class ClasseAcessada {

    private int atributo = 13;

    public MethodHandle atributoGetter() throws Throwable{
        MethodHandles.Lookup lookup = MethodHandles.lookup();
        MethodHandle getter = lookup.findGetter(
                ClasseAcessada.class, "atributo", int.class);
        return getter;
    }
}
```

A listagem a seguir tenta realizar o acesso ao atributo privado de duas formas distintas, sendo a primeira delas através de um `MethodHandle` criado na própria classe, e a segunda, a partir da instância de `MethodHandle` recuperada da classe onde está o atributo. Como resultado da execução desse código, a primeira tentativa que cria o `MethodHandle` fora da classe onde está o atributo irá falhar lançando uma exceção. Já a segunda tentativa que recupera o `MethodHandle` criado dentro da classe onde está o atributo será feita normalmente. Observe nas listagens que a criação das instâncias é idêntica, mudando somente o local onde é realizada.

Verificando o controle de acesso com MethodHandle:

```
public class ControleAcesso {
```

```
    public static void main(String[] args) {
        ClasseAcessada ca = new ClasseAcessada();

        System.out.println(
            "-> Tentando acesso direto ao atributo");
        try {
            MethodHandles.Lookup lookup =
                MethodHandles.lookup();
            MethodHandle getter = lookup.findGetter(
                ClasseAcessada.class, "atributo", int.class);
            System.out.println(
                "Obtido valor :"+getter.invoke(ca));
        } catch (Throwable t) {
            System.out.println("Não foi possível acessar "
                + " o atributo diretamente");
        }

        System.out.println("-> Tentando acessar MethodHandler "
                           + "criado na classe");
        try {
            MethodHandle getter = ca.atributoGetter();
            System.out.println("Obtido valor :"
                               + getter.invoke(ca));
        } catch (Throwable t) {
            System.out.println("Não foi possível acessar o "
                + "atributo diretamente");
        }
    }
}
```

Comparação de desempenho

A grande promessa da API Method Handle é que, devido ao fato de a verificação do controle de acesso ser feita apenas no momento da criação e do uso do bytecode `invokedynamic`, a invocação do método seria mais eficiente do que o uso da API de reflexão. Em alguns locais, inclusive, diz-se que o tempo de invocação pode-se aproximar de uma chamada direta em alguns casos. Como eu só acredito em código rodando, decidi verificar na

prática se o desempenho é realmente assim tão bom.

Na seção *Usar reflexão tem um preço?* foi mostrada uma comparação de desempenho entre a invocação de um método feito da forma normal, a invocação sendo feita com reflexão e a invocação com reflexão, mas fazendo cache da instância de `Method`. Nesta seção será incluída uma nova abordagem de invocação de método na comparação de desempenho utilizando o `MethodHandle`. A listagem a seguir mostra a implementação da interface `InvocadorMetodo`, que utiliza o `MethodHandle` para a invocação do método. Observe que o método `invokeExact()`, que segundo a documentação possui um melhor desempenho, foi utilizado.

Implementação de InvocadorMetodo que utiliza o MethodHandle:

```
public class InvocadorMethodHandler implements InvocadorMetodo {

    public void invocarMetodo(int vezes) {
        try {
            MethodHandles.Lookup lookup =
                MethodHandles.lookup();
            MethodType methodType =
                MethodType.methodType(void.class);
            MethodHandle methodHandle = lookup.findVirtual(
                ClasseTeste.class, "metodoVazio", methodType);

            ClasseTeste ct = new ClasseTeste();
            for(int i=0; i<vezes; i++){
                methodHandle.invokeExact(ct);
            }
        } catch (Throwable e) {
            throw new RuntimeException(
                "Não consegui invocar o método", e);
        }
    }
}
```

A listagem a seguir mostra o resultado de uma execução do teste de desempenho, incluindo a classe que utiliza reflexão fazendo cache do método e da que utiliza o `MethodHandle`, lembrando que é passado o valor 100000 para que o método `invocadorMetodo()` repita a operação. Observe que nessas condições a API Method Handle chega perto de cumprir sua promessa, executando em um tempo próximo da invocação normal do método.

Resultado de uma execução da comparação de desempenho:

```
A classe InvocadorNormal demorou 3293000 nano segundos
A classe InvocadorReflexaoCache demorou 15583000 nano segundos
4.732159125417552 vezes mais que o normal
A classe InvocadorMethodHandler demorou 4878000 nano segundos
1.4813240206498632 vezes mais que o normal
```

Antes de fechar essa questão do desempenho, preciso dizer que, ao executar esse mesmo código em outras máquinas, em alguns casos o tempo de execução com o `MethodHandle` era um pouco superior ao da invocação com reflexão. Outra questão interessante é que, aumentando o número de repetições do teste de desempenho, chega um ponto em que a API de reflexão começa a ter um desempenho muito superior ao do `MethodHandle`, talvez devido a alguma otimização mais agressiva da máquina virtual. Ao procurar outros testes de desempenho feitos por outros desenvolvedores e publicados em blogs, pude perceber que vários também tiveram esses resultados.

Sendo assim, antes de trocar todas as invocações de método do seu componente por uma invocação com `MethodHandle`, procure realizar testes de desempenho e verificar no seu caso qual seria o mais adequado. Essa API é muito interessante e possui diversos

recursos que irão ajudar muito na criação de componentes que utilizam reflexão. Basta torcer que esse ganho de desempenho seja mais estável e melhor em versões futuras da máquina virtual.

11.5 CONSIDERAÇÕES FINAIS

Ao se utilizar reflexão, existem aqueles recursos tradicionais que a grande maioria dos desenvolvedores que trabalharam com isso sabem, e outros mais escondidos que estão apenas na caixa de ferramentas de alguns. Esses recursos de conhecimento mais restrito muitas vezes podem ser úteis na busca de alguma metainformação que, por desconhecimento, precisa ser explicitamente passada como parâmetro. Por exemplo, já vi classes que implementam uma interface com um tipo genérico e que exigem que a instância de `Class` da classe do tipo genérico seja passada como parâmetro. A frase anterior soou um pouco repetitiva nas palavras, mas é exatamente assim que o código também irá ficar. Por que fazer o desenvolvedor passar como parâmetro algo que pode ser recuperado de outra forma?

Da mesma maneira, outros recursos mostrados neste capítulo também podem ser utilizados para evitar repetições e prover uma API mais amigável e com menos repetições. Por exemplo, uma classe que acessa o banco de dados, através do acesso à sua pilha de execução, poderia facilmente saber de qual *servlet* o acesso foi originado. Adicionalmente, você pode implementar o padrão Memento, que guarda o estado interno de um objeto, utilizando o acesso aos seus membros privados. Pode ser que agora você ainda não consiga enxergar uma aplicação para esses recursos, porém quando enxergar, tenha certeza de que farão uma grande diferença!

Capítulo 12

PALAVRAS FINAIS

"Não é o fim que é interessante, mas os meios para lá chegar." - Georger Braque

Este capítulo termina este livro com algumas palavras que tentam abordar de forma concisa as técnicas que foram apresentadas em toda a obra. Não irei falar sobre as técnicas em si, mas sobre como elas podem ser aplicadas no dia a dia de um desenvolvedor e fazer uma enorme diferença na produtividade da equipe.

O Java é uma linguagem estaticamente tipada, a qual possui diversas amarras que são utilizadas para garantir a segurança do código. É devido a isso que é gerado um erro de compilação ao tentar atribuir um objeto para uma variável com tipo não compatível ou ao invocar um método com parâmetros que não batem com o que foi especificado. A partir das últimas funcionalidades adicionadas na linguagem, como os tipos genéricos, é possível perceber que cada vez mais se busca uma maior segurança de código a partir de definições adicionais.

As técnicas ensinadas neste livro, de certa forma, são mecanismos que permitem que o desenvolvedor se livre dessas amarras impostas pela linguagem, possibilitando a criação de

componentes mais flexíveis e, consequentemente, mais reutilizáveis. Por mais que os defensores de linguagens dinâmicas digam que para eles o uso dessas funcionalidades é natural, eles estão sujeitos também a diversos problemas que podem ocorrer em tempo de execução. Questiono-me se é positivo que funcionalidades de reflexão estejam disponíveis para utilização de qualquer desenvolvedor em qualquer parte do código. É muito bom ter o poder da flexibilidade da reflexão, mas abusos de poder são sempre perigosos...

Sendo assim, acredito que essas funcionalidades de reflexão devam ser utilizadas dentro de componentes com responsabilidades bem definidas, que encapsulem o uso dessa técnica do resto do código. Dessa forma, os desenvolvedores podem se beneficiar do uso dessa técnica, mas sem ter que lidar diretamente com a complexidade trazida por ela. A possibilidade de reutilização desse tipo de componente que utiliza reflexão e metadados potencializa ainda mais os ganhos que se pode ter com eles. Em outras palavras, utilize reflexão em pontos bem definidos da sua arquitetura e encapsule sua utilização para que não adicione complexidade desnecessária em outras classes.

Procurei passar neste livro um conhecimento que levei diversos anos para adquirir, não apenas através de estudos, mas também através de experiência e pesquisa. Espero sinceramente que ele seja utilizado para o bem e com responsabilidade! Bons códigos!

Capítulo 13

REFERÊNCIAS BIBLIOGRÁFICAS

CAVALCANTI, Lucas. *VRaptor:* Desenvolvimento ágil para web com Java. Casa do Código, 2013.

CHEN, Nicholas. *Convention over configuration.* 2006. Disponível em: http://softwareengineering.vazexqi.com/files/pattern.html.

CORDEIRO, Gilliard. *CDI:* Integre as dependências e contextos do seu código Java. Casa do Código, 2013.

SILVEIRA, Fabio; FERNANDES, Clovis; CORREIA, Diego; GUERRA, Eduardo. *Quality improvement in annotated code.* CLEI Electronic Journal, v. 13, p. 7, 2010.

FERNANDES, Clovis; GUERRA, Eduardo; SILVEIRA, Fabio. *Classmock:* A testing tool for reflective classes which consume code annotations. Workshop Brasileiro de Métodos Ágeis (WBMA 2010), 2010.

FOWLER, Martin. *Fluent interface.* 2005. Disponível em: http://martinfowler.com/bliki/FluentInterface.html.

FOWLER, Martin. *Domain-Specific Languages.* Addison-

Wesley Professional, 2010.

ROHNERT, Hans; SOMMERLAD, Peter; STAL, Michael; BUSCHMANN, Frank; MEUNIER, Regine. *Pattern-Oriented Software Architecture - Volume 1:* A System of Patterns. Wiley, 1996.

GUERRA, Eduardo. *Design Patterns com Java:* projeto orientado a objetos guiado por padrões. Casa do Código, 2012.

LADDAD, Ramnivas. *AspectJ in Action:* Enterprise AOP with Spring Applications. Manning Publications, 2009.

MAES, Pattie. *Consepts and experiments in computational reflection.* Proceedings of the International Conference on Object Oriented Programming, Systens, Languages and Applications, 1987.

TURINI, Rodrigo; SILVEIRA, Paulo. *Java 8 Prático:* Lambdas, Streams e os novos recursos da linguagem. Casa do Código, 2014.

JAVA COMMUNITY PROCESS. *Javabeans(tm) specification 1.01 final release.* 1997. Disponível em: http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/.

SMITH, Brian. *Reflection and semantics in a procedural language.* Tese de Doutorado - Department Electrical and Computer Science Engineering, Massachusetts Institute of Technology, 1982.