Bruno Sofiato

O uso anotações para a definição da lógica de verificação da igualdade entre objetos na linguagem Java

São Paulo
2015

Bruno Sofiato

O uso anotações para a definição da lógica de verificação da igualdade entre objetos na linguagem Java

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Tecnologia de Software.

Área de Concentração:
Tecnologia de Software

Orientador: Prof. Dr. Paulo Sérgio Muniz Silva

São Paulo
2015

Catalogação-na-publicação

Sofiato, Bruno
O uso anotações para a definição da lógica de verificação da igualdade entre objetos na linguagem Java / B. Sofiato -- São Paulo, 2015.
155 p.

Monografia (MBA em Tecnologia de Software) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.JAVA 2.LINGUAGEM DE PROGRAMAÇÃO ORIENTADA À OBJETO I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

Agradecimentos

Agradeço o meu orientador, Prof. Dr. Paulo Sérgio Muniz Silva, pela orientação na elaboração deste trabalho, pelo seu incentivo e ajuda, não apenas na elaboração deste trabalho, mas em todos os aspectos da minha vida acadêmica.

Ao meu pai Antônio Carlos Sofiato e minha mãe Maria Angélica Marques Sofiato, por me darem, quando eu tinha apenas 8 anos, meu primeiro computador, me iniciando assim neste mundo apaixonante da computação.

Agradeço também minha namorada, Rita Tescke, que durante toda a elaboração deste trabalho ficou ao meu lado, me incentivando a dar o meu melhor, e, com o seu humor, transformando os momentos mais trabalhosos em momentos leves.

Agradeço a banca, composta pelo Prof. Dr. Fábio Levy Siqueira e pelo Prof. Dr. Kechi Hirama, pelas considerações e sugestões para melhoria deste trabalho.

Aos amigos e colegas de trabalho pelos momentos de descontração durante todo o período no qual confeccionei este trabalho, e a todas as pessoas que, direta ou indiretamente, me apoiaram e ajudaram durante a sua redação.

"All animals are equal,
but some animals are more equal than others"
(George Orwell)

Resumo

Diversos estudos apontam para a presença de problemas na implementação da lógica de verificação da igualdade entre objetos em *software* escritos na linguagem Java. Apesar de existirem proposições de soluções que mitigam este problema, existem questões, como a integração com *frameworks* e bibliotecas de terceiros, que impedem sua adoção. Outra deficiência encontrada reside no não tratamento do cálculo de ordem natural de comparação, conceito intimamente ligado à igualdade entre objetos.

Este trabalho propõe uma solução para a definição da lógica de verificação da igualdade entre objetos, incorporando tanto mecanismos para o cálculo da ordem natural, de maneira consistente com a semântica da igualdade, como mecanismos para a integração com *frameworks* de mercado. São abordadas neste trabalho questões de uso e de implementação da solução. Por fim, descreve-se um experimento, definido de maneira informal, elaborado para verificar aspectos da solução apresentada, como a expressividade, o desempenho e a corretude.

Palavras-chave: java, igualdade entre objetos, ordem natural de comparação, programação orientada a atributos.

Abstract

Several studies points to the existence of faults in the objects' comparison logic in Java written software. Despite the existence of several solution proposals that mitigate these faults, there are many issues preventing their adoption, such as third-party frameworks and libraries integration. Another shortcoming is the non-handling of the natural ordering, which is deeply connected to the equality concept.

This study proposes a solution for defining an object's comparison logic, featuring both equality-consistent natural ordering and third-party libraries' integration facilities. Issues regarding the proposed solution's usage and design are also addressed. Finally, an informally defined experiment, focused on some of the proposed solution's aspects such as expressiveness, performance and correctness, is discussed.

keywords: java, object equality, natural ordering, attribute oriented programming.

Lista de ilustrações

Figura 1 – Diagrama de Classes - Herança Múltipla e o Problema Diamante . . . 30
Figura 2 – Diagrama de Classe - Simetria 47
Figura 3 – Quantidade de Ciclos vs Tamanho do Ciclo 56
Figura 4 – Perfil de Uso de Coleções 57
Figura 5 – Diagrama de Perfil (VAZIRI et al., 2007) 65
Figura 6 – Diagrama de Perfil (RAYSIDE et al., 2009) 71
Figura 7 – Diagrama de Perfil (GRECH; RATHKE; FISCHER, 2010) 78
Figura 8 – Representação em Registro - Reflexividade 92
Figura 9 – Diagrama de Objetos - Simetria 93
Figura 10 – Criação de Registros - Simetria (Despacho Simples) 93
Figura 11 – Comparação de Registros - Simetria (Despacho Simples) 94
Figura 12 – Criação de Registros - Simetria (Despacho Múltiplo) 95
Figura 13 – Comparação de Registros - Simetria (Despacho Múltiplo) 95
Figura 14 – Comparação de Registros - Simetria (Cálculo de Ordem Natural) . . . 96
Figura 15 – Diagrama de Objetos - Transitividade 97
Figura 16 – Criação de Registros - Transitividade 98
Figura 17 – Comparação de Registros - Transitividade 98
Figura 18 – Diagrama de Objetos - 1ª Premissa de Transitividade (Cálculo de Ordem Natural) 99
Figura 19 – Comparação de Registros - 1ª Premissa de Transitividade (Cálculo de Ordem Natural) 100
Figura 20 – Diagrama de Objetos - 2ª Premissa de Transitividade (Cálculo de Ordem Natural) 100
Figura 21 – Comparação de Registros - 2ª Premissa de Transitividade (Cálculo de Ordem Natural) 101
Figura 22 – Criação de Registros - Não-nulabilidade 102
Figura 23 – Comparação de Registros - Não-nulabilidade) 103
Figura 24 – Cálculo de Registros - Não-nulabilidade (Cálculo de Ordem Natural) . 103
Figura 25 – Diagrama de Objetos - Interdependência entre *equals()* e *hashCode()* . 105
Figura 26 – Criação de Registros - *hashCode()* 106
Figura 27 – Cálculo de código hash através de registros de comparação 106
Figura 28 – Diagrama de perfil - Solução proposta 108
Figura 29 – Diagrama de Classe - Estrutura Básica 117
Figura 30 – Diagrama de Classe - Fábrica de estratégias de comparação (Implementação padrão) 120
Figura 31 – Diagrama de Classe - Estratégias de comparação (Implementação padrão) 121

Figura 32 – Diagrama de Classe - Estratégias de acesso a propriedades (Implementação padrão) 123
Figura 33 – Diagrama de Classes - Decorador de *Caching* (protótipo) 132
Figura 34 – Diagrama de Sequencia - Decorador de *Caching* (protótipo) 132

Lista de Listagens

Listagem 1 – Implementação de *equals()* - *Ponto* e *PontoColorido* 47
Listagem 2 – Geração por assistentes – Netbeans 8.0.2 61
Listagem 3 – Geração por assistentes – Eclipse Kepler 62
Listagem 4 – Declaração – Palavras-chave (VAZIRI et al., 2007) 65
Listagem 5 – Declaração – Anotações (VAZIRI et al., 2007) 66
Listagem 6 – Obtenção de instância de tipo de relação (VAZIRI et al., 2007) . . . 66
Listagem 7 – Comparação baseada em classe (RAYSIDE et al., 2009) 71
Listagem 8 – Comparação baseada em hierarquia (RAYSIDE et al., 2009) 72
Listagem 9 – Comparação baseada em referência (RAYSIDE et al., 2009) 72
Listagem 10 – Comparação baseada em classe (GRECH; RATHKE; FISCHER, 2010) 78
Listagem 11 – Utilização do parâmetro *eqClass* (GRECH; RATHKE; FISCHER, 2010) 79
Listagem 12 – Comparação baseada em referência (GRECH; RATHKE; FISCHER, 2010) 79
Listagem 13 – Idioma de implementação de verificação da igualdade (ODERSKY, 2010) 80
Listagem 14 – Fragmento de ponto de corte para o tratamento de ciclo (GRECH; RATHKE; FISCHER, 2010) 82
Listagem 15 – 1º Exemplo de Utilização - Solução Proposta 112
Listagem 16 – 2º Exemplo de Utilização - Solução Proposta 112
Listagem 17 – 3º Exemplo de Utilização - Solução Proposta 113
Listagem 18 – 4º Exemplo de Utilização - Solução Proposta 114
Listagem 19 – 5º Exemplo de Utilização - Solução Proposta 115
Listagem 20 – 6º Exemplo de Utilização - Solução Proposta 115
Listagem 21 – Pseudocódigo - Algoritmo de Verificação Otimizado 133

Lista de tabelas

Tabela 1 – Estudo de Caso (VAZIRI et al., 2007) - Sumário de Resultados. 54
Tabela 2 – Estudo de Caso (FRASER; ARCURI, 2013) - Sumário de Resultados. . 55
Tabela 3 – Expressividade (RAYSIDE et al., 2009) 74
Tabela 4 – Corretude (RAYSIDE et al., 2009). 75
Tabela 5 – Facilidade de Uso (Anotações) (RAYSIDE et al., 2009). 75
Tabela 6 – Facilidade de Uso (Métodos/Classes) (RAYSIDE et al., 2009) 76
Tabela 7 – Desempenho (RAYSIDE et al., 2009) 76
Tabela 8 – Desempenho (GRECH; RATHKE; FISCHER, 2010) 84
Tabela 9 – Equivalência esperada - Interdependência de operações 105
Tabela 10 – Código *hash* obtidos - Interdependência de operações 107
Tabela 11 – Resultado experimento – Corretude (testes de unidade) 141
Tabela 12 – Resultado experimento – Corretude (análise estática) 142
Tabela 13 – Resultado experimento – Expressividade (métodos removidos) 143
Tabela 14 – Resultado experimento – Expressividade (utilização de anotações) . . . 144
Tabela 15 – Resultado experimento – Expressividade (estratégias customizadas) . . 145
Tabela 16 – Resultado experimento – Desempenho 146

Lista de abreviaturas e siglas

AOP Aspect Oriented Programming

API Application Programming Interface

CLOS Common Lisp Object Systemn

DNS Domain Name System

IDE Integrated Development Environment

IEEE Institute of Electrical and Electronics Engineers

IP Internet Procotol

ISBN International Standard Book Number

ISP Interface Segregation Principle

ITD Inter-type Declaration

JDK Java Development Kit

JRE Java Runtime Environment

JVM Java Virtual Machine

MOP Metaobject Procotol

NaN Not a Number

OMG Object Management Group

PARC Palo Alto Research Center

RAII Resource Acquisition Is Initialization

SOLID Single responsibility, Open/closed, Liskov substitution, Interface segregation, Dependency inversion

SPI Service Provider Interface

UML Unifed Modeling Language

URL Uniform Resource Locator

XML Extensible Markup Language

Sumário

Sumário 19

1 INTRODUÇÃO 21
1.1 Motivação 21
1.2 Objetivo 22
1.3 Justificativa 23
1.4 Estrutura do Trabalho 24

2 FUNDAMENTOS DA ORIENTAÇÃO A OBJETOS 27
2.1 Conceitos básicos da orientação a objetos 27
2.2 Tipagem 36
2.3 A linguagem de programação utilizada na solução proposta 39

3 IGUALDADE DE OBJETOS 41
3.1 O conceito da igualdade de objetos 41
3.2 Contrato de Igualdade 46
3.3 Uso de coleções com restrições de igualdade 51
3.4 Problemas encontrados em implementações 53

4 SOLUÇÕES DISPONÍVEIS 59
4.1 Critérios de comparação das soluções acadêmicas 59
4.2 Soluções analisadas 60

5 SOLUÇÃO PROPOSTA PARA A COMPARAÇÃO DE OBJETOS . 89
5.1 Caracteristicas principais 89
5.2 Contrato de Igualdade 91
5.3 Formas de Uso 107

6 IMPLEMENTAÇÃO DO PROTÓTIPO 117
6.1 Estrutura do Protótipo 117
6.2 Características da implementação 124

7 EXPERIMENTO 137
7.1 Método 137
7.2 Execução 140
7.3 Análise dos Resultados 141

8 CONCLUSÃO E TRABALHOS FUTUROS 147

8.1 Conclusão 147
8.2 Trabalhos Futuros 149

Referências 151

1 Introdução

A palavra igualdade, originária da palavra latina *aequalis* (comparável, similar), é amplamente utilizada no nosso dia-a-dia, em frases como "aquele carro é igual ao carro à sua direita" ou "esses dois produtos são iguais", quando desejamos estabelecer uma relação de semelhança ou equivalência entre duas entidades.

Dependendo do contexto de utilização, o significado da relação de igualdade entre dois objetos pode ser diferente (TARSKI, 1994, p. 56). Por exemplo, no campo da lógica dois objetos só são considerados iguais se ambos tiverem as mesmas propriedades (na prática se forem exatamente o mesmo objeto). Já na aritmética, não é necessário que dois números sejam necessariamente idênticos para que sejam considerados iguais. No que tange à geometria, duas figuras são consideradas iguais (congruentes) se tiverem a mesma forma e tamanho, não importando por exemplo, as suas posições.

Porém, independentemente do contexto de utilização, a relação de igualdade entre dois objetos deve ter determinadas propriedades, consideradas axiomáticas. Essas propriedades são a reflexividade, a simetria e a transitividade. No campo da lógica, essas propriedades podem ser inferidas através da Lei de Leibniz (LEIBNIZ, 1976, cap. 35), já no campo da aritmética, são representadas pelos Axiomas de Peanno (PEANO, 1889, p.1)

No paradigma da programação orientado a objetos, amplamente utilizado nas linguagens comerciais modernas, um programa é composto de objetos e suas relações, que, por sua vez, são representações de elementos reais ou conceituais e abstrações pertencentes ao domínio do problema a ser resolvido.

Essa característica dos programas orientados a objetos, aliada à importância do conceito de igualdade na definição de expressões lógicas, faz com que seja desejável um mecanismo robusto para a definição da verificação da igualdade, na qual sejam mantidas todas as propriedades axiomáticas dessa relação consideradas e, ao mesmo tempo, a semântica da igualdade declarada pelo desenvolvedor.

1.1 Motivação

A implementação das operações de verificação da igualdade em linguagens orientadas a objetos não é trivial. Questões relativas ao despacho de mensagens impossibilitam a manutenção da característica de simetria e de transitividade quando são utilizados os mecanismos de herança (BRUCE et al., 1995).

Além disso, em algumas linguagens de programação (como por exemplo a linguagem

Java), a lógica de verificação da igualdade está disposta em duas operações, que apesar de distintas, são regidas por um contrato que exige uma determinada consistência em função da semântica de verificação da igualdade adotada pelo programador.

Esse quadro se agrava quando o desenvolvedor necessita incluir um mecanismo de cálculo da ordem natural a uma determinada classe. Nesses casos, incorpora-se normalmente ao código-fonte da classe uma terceira operação, que também tem sua implementação regida por um contrato informal de interdependência. Violações na relação entre as operações que compõem o mecanismo de verificação da igualdade, regido por esse contrato informal de interdependência, acarretam defeitos que podem ser de difícil diagnóstico (BLOCH, 2008, p. 35).

Apesar disso, frequentemente não são realizadas, tanto em tempo de compilação quanto em tempo de execução, verificações de consistência entre os algoritmos utilizados nessas operações.

Outra questão comumente encontrada nas implementações das operações de verificação da igualdade reside na natureza das comparações realizadas para determinar se dois operandos são iguais. Como é necessário a verificação em profundidade de todas as propriedades de um objeto, é possível que, na presença de referências cíclicas, a execução do programa entre em recursão infinita, acarretando no término da execução do programa em virtude de estouro da pilha de execução (*stack overflow*).

1.2 Objetivo

O objetivo principal deste trabalho é propor uma solução para o problema da definição da lógica de verificação da igualdade entre objetos, que possa mitigar alguns problemas encontrados em soluções previamente propostas. Para isso serão analisadas, no decorrer deste trabalho, algumas propostas de soluções (tanto do meio acadêmico como de mercado) do problema em questão. A partir dos resultados obtidos dessa análise, será proposta uma solução para o problema. Por fim, realiza-se um experimento informal, que permita a comparação do programa inicial com o modificado pela solução em relação ao desempenho, à corretude e à expressividade do ponto de vista da semântica de verificação da igualdade.

É imperativo que a lógica de verificação da igualdade gerada pela solução aqui proposta cumpra integralmente o contrato que rege a verificação da igualdade e, ao mesmo tempo, seja semanticamente equivalente à lógica expressa pelo desenvolvedor. Também é importante que a solução proposta disponibilize recursos para a definição do cálculo da ordem natural entre objetos, de maneira semanticamente consistente com a lógica de verificação da igualdade definida. Outros pontos são levados em conta na proposição da solução.

a) Mecanismos de integração com bibliotecas de terceiros e classes cujos códigos-fonte estejam indisponíveis ou não possam ser alterados;

b) Compatibilidade com *frameworks* de mercado;

c) Integração com ferramental de desenvolvimento comumente utilizado (como, por exemplo, depuradores);

d) Robustez da lógica gerada na presença de referências cíclicas;

e) Impacto da adoção da solução no desempenho geral do programa.

1.3 Justificativa

Segundo resultados de experimentos encontrados em (VAZIRI et al., 2007) e (FRASER; ARCURI, 2013), foram encontradas, através da análise do código-fonte de diversas aplicações de código-fonte aberto, grandes quantidades de violações do contrato que rege a lógica de verificação da igualdade entre objetos – todos os programas analisados por (VAZIRI et al., 2007), com exceção de Shrike[1] , tiveram problemas na implementação do contrato de verificação da igualdade. Além dessas violações de contrato, foram encontrados por (RAYSIDE et al., 2009) problemas relativos a falhas no tratamento de referências circulares, que, segundo (MELTON; TEMPERO, 2007), são comumente encontradas em programas atualmente em produção.

Apesar das IDE (*Integrated Development Environment*) de mercado incorporarem assistentes capazes de gerar implementações das operações associadas ao mecanismo de verificação da igualdade, as implementações por eles geradas não têm um grau de corretude e robustez adequado (RUPAKHETI, 2010, p. 117), falhando tanto na manutenção da simetria e transitividade quando existe uma relação de herança entre os operandos, impossível em linguagens de despacho simples (BRUCE et al., 1995), quanto no tratamento de referências cíclicas.

Além dos assistentes inclusos nas IDE, foram propostas em (VAZIRI et al., 2007), (RAYSIDE et al., 2009) e (GRECH; RATHKE; FISCHER, 2010) soluções para a definição da lógica de verificação da igualdade na linguagem Java. Essas soluções têm como base a utilização de anotações para a definição, de maneira declarativa, do papel de uma propriedade na lógica de verificação de igualdade. A partir dessas definições são geradas implementações para as operações pertencentes ao mecanismo de verificação da igualdade, de maneira que o contrato que as rege seja cumprido.

Apesar de as soluções propostas resolverem o problema da definição correta da lógica de verificação da igualdade, existem algumas limitações que comprometem a adoção destas

[1] Shrike technical overview - WalaWiki. Disponível em: ⟨http://wala.sourceforge.net/wiki/index.php/Shrike_technical_overview⟩.

soluções. Entre elas, encontra-se a falta de mecanismos para a definição da lógica de verificação da igualdade de classes cujos códigos-fonte estejam indisponíveis[2] e a definição do cálculo da ordem natural, conceito este intimamente ligado ao conceito de igualdade.

Outra limitação encontrada em todas as soluções analisadas, com exceção da solução proposta em (RAYSIDE et al., 2009), é a restrição obrigando que todos os atributos utilizados pela lógica de verificação da igualdade sejam imutáveis. Essa restrição impede a utilização da solução quando se usa de *frameworks* com um alto grau de penetração no mercado, como por exemplo o Hibernate (http://www.hibernate.org/).

1.4 Estrutura do Trabalho

Este trabalho foi dividido em oito capítulos. O primeiro é composto por esta introdução.

O segundo e o terceiro capítulos têm como objetivo a explanação de alguns conceitos teóricos que serão utilizados no decorrer do trabalho. O foco principal do segundo capitulo é tanto a definição de conceitos básicos (como por exemplo, teoria de tipos e contratos, orientação a objetos e etc.), quanto a definição do contexto de uso da solução.

Com os conceitos teóricos estabelecidos, é definido no terceiro capitulo o conceito de igualdade entre objetos, bem como suas premissas, restrições e hipóteses. Também são elencadas nesse capítulo os reflexos encontrados em software hoje em produção, quando existem falhas na implementação das operações de verificação da igualdade sob a ótica desse conceito.

No capítulo quatro é feito um estudo de algumas soluções encontradas (tanto no meio acadêmico como de mercado) para o problema da igualdade. Também são definidos os critérios utilizados na comparação das soluções, tendo como base as premissas e conceitos delineados nos segundo e terceiro capítulos.

Tendo como base o comparativo realizado no capítulo quatro, é proposta uma solução para o problema da definição da igualdade no quinto e sexto capítulos. No quinto capítulo a solução proposta neste trabalho será descrita sob o ponto de vista teórico; já no sexto capítulo serão focalizadas as decisões de projeto adotadas na implementação de um protótipo da solução.

No capítulo sete será descrito um experimento informal, que tem como base a utilização da solução apresentada na definição da lógica de verificação de igualdade de uma ferramenta de código-fonte aberta com alto grau de utilização. Nesse experimento serão comparados o programa original e o modificado para a adoção da solução apresentado

[2] A solução proposta por (GRECH; RATHKE; FISCHER, 2010) cita essa característica, porém, não fornece detalhes sobre a definição da lógica de verificação da igualdade.

neste trabalho, segundo métricas de desempenho, corretude e expressividade, com respeito à semântica de verificação da igualdade.

Por fim, no capítulo oito serão dispostas as conclusões com relação à eficácia da solução proposta neste trabalho na mitigação do problema da verificação da igualdade. Também serão elencados questionamentos encontrados no decorrer da elaboração desse trabalho. Esses questionamentos podem servir de base para trabalhos futuros.

2 Fundamentos da Orientação a Objetos

Este capítulo tem como principal objetivo a exposição de alguns fundamentos e conceitos técnicos, necessários para um entendimento mais profundo do problema da definição da lógica de comparação de objetos, bem como o entendimento das causas e implicações das decisões tomadas na solução proposta nesta monografia para a definição da lógica de verificação de igualdade entre objetos.

Primeiramente será exposto um panorama geral sobre o paradigma de programação orientada a objetos. Além disso, também serão expostos conceitos da programação orientada a aspectos e de programação orientada a atributos.

Depois será feita uma breve digressão tendo como foco a teoria de tipos, bem como as diferenças fundamentais entre o mecanismo de herança, disponível nas linguagens orientada a objetos baseadas em classes, e a noção da relação entre tipos e subtipos, tendo como base a teoria de subtipagem comportamental de Barbara H. Liskov e Jeannette M. Wing (LISKOV; WING, 1994). Por fim, será contextualizado o problema a ser estudado e a solução proposta, no que tange os conceitos e fundamentos teóricos expostos neste capítulo.

2.1 Conceitos básicos da orientação a objetos

No paradigma de programação orientada a objetos um programa é composto pela interação entre uma série de elementos, em que cada elemento incorpora tanto dados quanto algoritmos. Esses elementos são denominados de objetos. Neste capitulo serão explanados alguns conceitos desse paradigma, no intuito de situar o leitor nos termos e conceitos do problema a ser resolvido por esta monografia.

O paradigma de programação orientada a objetos teve início em meados de 1960 com a criação da linguagem de programação SIMULA 67, que introduziu pela primeira vez os conceitos de classes e instâncias de objetos. As ideias incorporadas à linguagem SIMULA 67 fizeram com que Alan Kay e os pesquisadores do PARC da Xerox desenvolvessem na década de 70 a linguagem Smalltalk, linguagem que continua a inspirar nos dias de hoje as linguagens orientada a objetos.

Nos anos 1980, cresceu o uso da orientação a qual foi crescendo e se transformou no paradigma de programação dominante. A popularidade do paradigma fez com que linguagens já estabelecidas adotassem mecanismos para o seu uso (como por exemplo, a linguagem C que ao adotar mecanismos para programação orientada a objetos deu origem ao C++). Nos dias de hoje, o legado da programação orientada a objetos pode ser visto

em linguagens como o C#, Java e Javascript, que são amplamente utilizadas nos sistemas baseados em software atualmente em produção.

2.1.1 Classificações de linguagens orientadas à objetos

Apesar do paradigma da programação orientada a objetos ser um paradigma cujo conceitos teóricos são bem definidos, existem algumas diferenças significativas entre alguns conceito utilizados nas das diversas linguagens de programação orientada a objetos disponíveis. Essas diferenças abrangem tanto questões do sistema de tipos (como por exemplo a forma de definição de objetos), como o mecanismo de escolha do código a ser executado no recebimento de mensagens (também conhecido como mecanismo de despacho de mensagens). Uma classificação pragmática(BRUCE, 2002) dessas linguagens permite identificar mais claramente essas diferenças : linguagens baseadas em protótipos e linguagens baseadas em classes. A seguir, resume-se as propriedades principais dessas classificações, conforme descritas em (BRUCE, 2002) :

Linguagens baseadas em protótipos

Uma forma de criação de novos objetos baseia-se no mecanismo de prototipação. Linguagens de programação utilizando essa forma de orientação a objetos também são chamadas de linguagens baseadas em objetos. Em linguagens baseadas em protótipos, o comportamento e os dados de um objeto são definidos em um objeto, que é usado como protótipo para a criação de outros objetos (daí o nome de linguagens baseadas em protótipo). Esses novos objetos podem ser por sua vez utilizados como protótipos na criação de novos objetos e assim por diante.

As principais críticas às linguagens orientadas a objetos onde a criação de novos objetos se dá através de prototipação são as mesmas aplicadas às linguagems com sistema de tipos dinâmicos. Outra crítica recorrente do uso desse tipo de linguagem é a falta de familiaridade dos desenvolvedores, mesmo aqueles com grande conhecimento do paradigma de programação orientada a objetos :

> "Virtualmente todas linguagens orientadas a objetos populares (por exemplo, Simula 67, Smalltalk, Object Pascal, Eiffel, Objective C, C++, Ada95 e Java) são baseadas em classe. Por outro lado, linguagens baseadas em objetos (como por exemplo Self, Cecil, and Emerald) tendem a ser linguagens de pequisa ou são utilizadas por comunidades relativamente pequenas."(BRUCE, 2002, p. 12, tradução do autor).

Pode-se acrescentar ao rol de linguagens orientadas a objetos baseadas em prototipação o Javascript, que atualmente é a linguagem baseada em protótipos mais popular, em função principalmente do seu papel na programação de aplicações web.

Linguagens baseadas em classes

Outra forma de criação de novos objetos é a utilizada nas linguagens orientadas a objetos baseadas em classe. Nesse tipo de linguagem, o programador cria gabaritos (*templates*) extensíveis denominados classes para a definição do comportamento que os objetos devem ter. Novos objetos são criados através do uso de uma operação de construção de uma classe. Objetos criados a partir de uma classe são chamados de instâncias desta classe.

Quase todas as linguagens orientadas a objetos populares (como por exemplo, Simula 67, Smalltalk, Object Pascal, Eiffel, Objective C, C++, Ada95, and Java) são baseadas em classe. Basicamente a única linguagem baseada em objetos popular atualmente é, conforme citado anteriormente, o Javascript.

Objetos são elementos que englobam tanto comportamento quanto informação. A interação entre objetos se dá através do mecanismo de troca de mensagens: quando uma mensagem é recebida por um objeto, uma operação é selecionada para invocação.

Um conceito importante da orientação a objetos é o conceito de encapsulamento, onde são definidas permissões de acesso a elementos de um objeto. Um mecanismo utilizado por diversas linguagens de computador para implementar o conceito de encapsulamento é o de o modificador de acesso. Existem peculiaridades entre os níveis de controle de acesso disponíveis em cada linguagem, mas em geral pelo menos dois modificadores estão disponíveis : o nível público, onde um elemento de a uma classe pode ser acessado por qualquer outro elemento, e o nível privado, onde o acesso só é permitido a elementos declarados na mesma classe.

Conforme já dito, nesta categoria de linguagem, os objetos são criados através de classes. As classes podem conter atributos, que são utilizados para o armazenamento das informações associadas a cada instância da classe. Podem também conter operações, que definem os tipos de mensagens que as instâncias da classe podem receber, assim como métodos, que definem o código a ser executado quando uma operação é invocada. Instâncias de uma mesma classe compartilham os métodos e operações nela definidos, porém tendo informações distintas armazenadas. Para a criação de novas instâncias, é utilizada uma operação especial de uma classe, o construtor.

Caso se deseje estender o comportamento definido em uma classe, o desenvolvedor pode utilizar o mecanismo de herança, presente nas linguagens de programação orientadas a objetos baseadas em classe. No mecanismo de herança, é criada uma relação entre a classe pai (chamada daqui em diante de superclasse) e a classe filha (chamada de subclasse).

Com relação ao mecanismo de herança, podemos subdividir as linguagens orientadas a objetos baseadas em classe em duas categorias : linguagens com herança simples e linguagens com herança múltipla. Nas linguagens com herança simples (como por exemplo

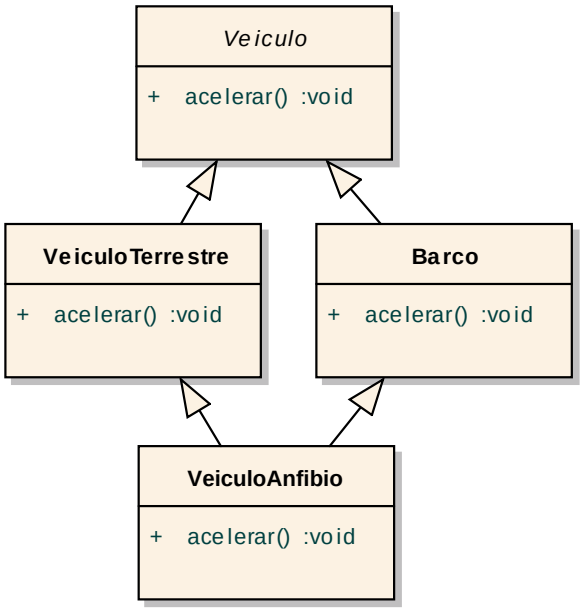
a linguagem Java) só é permitida a extensão de uma única classe pelo programador, restrição esta inexistente em linguagens com suporte à herança múltipla (como por exemplo C++ e Eiffel).

A seguir, serão descritos alguns conceitos ligados a arranjos de objetos e a construtos das linguagens em questão, bem como técnicas que serão referenciadas mais adiante nesse trabalho.

Herança múltipla e o problema do diamante

Ocasionalmente é necessária a composição de múltiplas abstrações em um única classe (MEYER, 1997). Com o intuito de exemplificar as peculiaridades da herança múltipla, segue um fragmento de um sistema hipotético de controle de veículos, composto de barcos, veículos anfíbios e terrestres. No diagrama de classes na figura 1 encontra-se uma possível modelagem desse sistema :

Figura 1 – Diagrama de Classes - Herança Múltipla e o Problema Diamante



Esse tipo de modelagem é permitido em linguagens com suporte à herança múltipla, em que é possível na definição de uma nova classe a declaração de quantas superclasses forem necessárias. Olhando com mais cuidado pode-se identificar alguns problemas. Caso a operação *acelerar()* seja invocada numa instância da classe *VeiculoAnfibio* qual deve ser o comportamento esperado ? Deve ser aumentada a rotação da hélice ou das rodas do veículo ?

Esse tipo de problema recebe o nome de problema do diamante (MARTIN, 1997), em virtude do desenho da hierarquia de classes obtida lembrar um diamante. As diversas

linguagens que dão suporte à herança múltipla têm formas distintas de lidarem com esse problema. Na linguagem Eiffel por exemplo, caso dois elementos pertencentes as superclasses tenham o mesmo nome, é necessária que a nova classe renomeie um deles, no intuito de evitar conflitos (MEYER, 1997, p. 535).

Mixins

Um outro conceito de intimamente ligado ao conceito de herança múltipla é o conceito de *mixin*. Um *mixin* representa um pacote de funcionalidades que podem ser utilizada na composição de novas classes. Segue uma definição formal do que vem a ser um *mixin* :

> "Um *mixin* é uma subclasse abstrata que pode ser utilizada para especializar o comportamento de uma variedade de superclasses. Normalmente isso é feito através da definição de novos métodos que executam algumas ações e depois chamam um método correspondente da superclasse."(BRACHA; COOK, 1990, tradução do autor)

Com o intuito de evitar o problema do diamante, inerente às linguagens com suporte à herança múltipla. É utilizado para a inclusão[1] dos *mixins* a uma classe um processo chamado linearização. Segundo (SNYDER, 1986), no processo de linearização a hierarquia de classes é rearranjada, de modo que as heranças múltiplas sejam eliminadas. A linearização, apesar de resolver o problema do diamante, tem alguns problemas, conforme levantado por (SNYDER, 1986; BRACHA; COOK, 1990) :

a) Existe uma mudança na relação entre uma classe e suas superclasses declaradas em código-fonte e a relação em tempo de execução;

b) Caso existam conflitos de nome em elementos dos diversos *mixins*, um elemento será selecionado (Normalmente de acordo com a ordem de declaração do *mixin* no elemento sendo construído), sem qualquer tipo de aviso ao programador[2];

c) Como podem são inclusas classes entre uma classe e sua superclasse, a passagem de mensagens entre as duas pode ser modificada.

Interfaces e herança múltipla

Como pode-se ver anteriormente, existem alguns casos em que um novo elemento é uma generalização de mais de um elementos. Apesar do aumento da complexidade decorrente do problema da herança em diamante, ainda é possível este tipo de construção em linguagens com suporte à herança múltipla.

No entanto, é necessária uma forma de expressar esse tipo de generalização em linguagens baseadas em herança simples. Para isso foi criado o conceito de interface, no

[1] Esse mecanismo é utilizado pelo CLOS (KICZALES, 1991, p. 246), outras linguagens podem-se utilizar de outros mecanismos

[2] Essa característica é uma característica de implementação

qual são agrupadas operações semânticamente relacionadas. Da mesma maneira que uma classe, uma interface pode ser utilizada para a definição de um tipo (BRUCE, 2002, p. 51), cuja interação com os elementos externos é regida por um contrato bem definido. A relação entre uma classe e uma interface é chamada de realização.

Apesar de vetada a especialização de mais de uma superclasse por uma classe, ainda é possível em linguagens com herança simples a realização de diversas interfaces por uma classe. Como as operações definidas em interfaces não podem ter nenhuma implementação pré-definida, o problema da seleção do método correto em situações de herança com ancestrais em comum (herança em diamante) deixa de ocorrer (MARTIN, 1997).

Polimorfismo

Independentemente do tipo de herança utilizado, é possível a inclusão de novas operações e atributos na subclasse. Também é possível efetuar uma sobrescrita de métodos, onde um método definido na superclasse é substituído por uma nova versão, definida na subclasse. O recurso da sobrescrita de métodos permite que objetos troquem mensagens de maneira homogênea, sem que os objetos que enviam mensagens tenham conhecimento da lógica utilizada para o tratamento delas. Esse recurso, que permite um alto grau de flexibilidade e reuso de código associados as linguagens orientadas a objetos, é chamado de polimorfismo[3].

Despacho simples

Conforme previamente citado, uma forma de categorização de linguagens orientadas a objetos é a de escolha do código responsável pelo tratamento das mensagens recebidas pelos objetos. O primeiro grupo de linguagens é composto pelas linguagens com despacho simples, composto pela grande maioria das linguagens orientas a objetos disponíveis no mercado(por exemplo Java, C++ e C# são linguagens baseadas em despacho simples).

Em linguagens com despacho de mensagens simples, a operação utilizada numa passagem de mensagem para um objeto é definida em tempo de compilação(MUSCHEVICI et al., 2008, cap. 2). Para a definição dessa operação, são utilizadas informações presentes no código-fonte, como por exemplo, os tipos declarados dos parâmetros enviados nas mensagens e o tipo do retorno esperado dessa mensagem.

Despacho múltiplo

Já em linguagens com despacho múltiplo (também chamadas de linguagens multimétodos), a escolha do método para execução a da lógica referente ao recebimento de uma mensagem é baseada tanto no objeto que recebe a mensagem como nos parâmetros

[3] Mais precisamente esse tipo de polimorfismo é chamado por (BRUCE, 2002, p. 25) de polimorfismo estrutural

passados. Como essas informações só estão disponíveis no momento em que o programa é executado, todo o processo de escolha do método é adiado para o tempo de execução.

Pode-se dar como exemplo de linguagens que utilizam o mecanismo de despacho múltiplo as linguagens CLOS (Common Lisp Object System), Dylan e Cecil. Diferentemente da maioria das linguagens, nestas linguagens geralmente os métodos não estão associadas a uma única classe (ou a um objeto, no caso de linguagens baseadas em protótipos). Na implementação de algumas mensagens, os métodos não pertecem a nenhuma classe. Já em outras implementações, um método pode pertencer a nenhuma, a uma ou a muitas classes, dependendo do número de parâmetros (MUSCHEVICI et al., 2008). Essa característica única faz alguns autores classificar as linguagens com métodos múltiplos como um tipo de linguagem distinto das baseadas em classes e baseadas em protótipo (BRUCE, 2002, cap. 7)

Simulação de despacho múltiplo em linguagens com despacho simples

Conforme (BRUCE et al., 1995), existem casos, onde a utilização de multi-métodos se faz necessária. Para a resolução dessa categoria de problemas em linguagens baseadas em despacho simples foram criadas técnicas que permitem a simulação métodos múltiplos.

A primeira técnica a ser estudada foi proposta por Daniel Ingalls (INGALLS, 1986). Na técnica de despacho múltiplo proposta por Ingalls, é implementado pelo programador, no método implementando a operação pertencente ao receptor da mensagem, uma série de seletores, que em função dos argumentos fornecidos à operação, delega o processamento ao método encapsulando a lógica de tratamento adequada para o tratamento da mensagem.

Apesar de ser simples de implementar, a solução proposta por Ingalls tem diversos problemas (MUSCHEVICI, 2009), entre os quais estão a falta de robustez da solução (é relativamente fácil o esquecimento de uma verificação), a quebra da modularidade (é necessário que o elemento receptor conheça todas as implementações possíveis de parâmetros) e a complexidade, que é aumentada exponencialmente quando mais parâmetros são inclusos no processo de seleção de métodos.

Outra técnica para a implementação de multi-métodos é a técnica baseada no padrão de projetos Visitante(*Visitor*) (GAMMA et al., 1994, p.366). Quando o visitante é executado, a escolha do método responsável pela resposta à uma invocação depende tanto do visitante propriamente dito quanto do elemento sendo visitado. Essas características fazem com que o padrão visitante implemente o despacho múltiplo :

> "Essa é chave para o padrão de projetos visitante : A operação que é executada depende, tanto do tipo do visitante quanto do tipo de elementos que ele visita ..." (GAMMA et al., 1994, p.374, tradução do autor)

As desvantagens associadas ao uso do padrão visitante são análogas às inerentes ao uso da solução proposta por Ingalls : A implementação do visitante devem conhecer todas as possíveis classes a serem visitadas, afetando a modularidade de sistema e impactando na sua manutenibilidade.

Referências vs. instâncias

Um conceito que será utilizado no decorrer do trabalho é o conceito de referência a objeto. Referências são utilizadas como mecanismo de acesso a instâncias de objetos disponíveis no sistema de *software*. Algumas linguagens, como por exemplo C++, implementam essas referências como ponteiros para endereços de memória, outras, como C# e Java, incorporam restrições na manipulação destes endereços. É possível uma referência não estar referenciando nenhuma instância em determinado momento do fluxo de execução, nesse caso elas são chamadas de referencias nulas.

Como as referências são basicamente endereços de memória alocada para instâncias de objetos, diversas referências podem estar apontando para uma única instância. Essa caracteristica, aliada a algumas estruturas de codificação, como por exemplo o padrão Compósito (*Composite*) (GAMMA et al., 1994, p. 183), possibilita a criação de ciclos no grafo de referências de um objeto. Esses ciclos, quando presentes, acarretam uma recursão infinita na passagem de mensagens, que leva ao estouro na pilha de chamadas(*stack overflow*).

2.1.2 Programação baseada em atributos (anotações)

A programação baseada em atributos é uma técnica de programação na qual o programador incorpora informações adicionais de forma declarativa sobre elementos presentes no código-fonte (como por exemplo, classes e métodos) (WADA; SUZUKI, 2005). Essas informações são metadados que podem ser utilizadas tanto no processo de compilação, sendo utilizadas por um compilador ou pré-processador para a geração de código-fonte extra, como em tempo de execução, servindo de fonte de informações para *frameworks* e ambientes de execução.

O conceito de programação baseada em atributos já é incorporado em diversas linguagens de programação, como por exemplo Java e C#. A linguagem C# foi criada desde a sua concepção com um mecanismo de programação baseado em atributos. Já na linguagem Java, essa forma de programação foi somente incorporada a partir da versão 1.5 da linguagem, através do mecanismo de anotação[4] . Daqui em diante será utilizado o termo anotação para referenciar o mecanismo de definição de metadados disponível em uma linguagem de programação.

[4] JSR 175: A Metadata Facility for the JavaTM Programming Language. Disponível em: ⟨https://www.jcp.org/en/jsr/detail?id=175⟩.

Conforme (CEPA; KLOPPENBURG, 2005), é possivel traçar um paralelo entre os mecanismos de programação baseados em atributos ao mecanismo de perfis, disponiveis na especificação 2.0 da UML (OMG, 2011). .

O mecanismo de perfil, conforme defindo na versão 2.0 da UML, é um mecanismo de extensão do metamodelo da UML que permite a definição de novas metaclasses, possibilitando a incorporação de novas divisões semânticas e comportamentais no metamodelo a ser utilizado na modelagem. Essas novas metaclasses são definidas através de elementos chamados esteriótipos.

Além da definição de estereótipo propriamente dito, a especificação UML permite que o esteriótipo tenha atributos, também chamados de propriedades[5]. Essas propriedades permitem que o desenvolvedor customize o comportamento dessas metaclasses no momento de uso.

Conforme dito anteriormente, é possível traçar um paralelo entre os elementos utilizados na definição de um perfil UML e os elementos disponiveis em uma linguagem preparada para a programação baseada em atributos. Estereótipos podem ser representados pelas anotações propriamente ditas. Ambos são associados a um elemento distinto (metaclasses no caso dos estereótipos e elementos de código-fonte no caso das anotações), podendo conter um conjunto de propriedades devidamente tipadas.

2.1.3 Programação baseada a aspectos

O paradigma de programação orientada à aspectos (AOP) é um novo paradigma de programação desenvolvido pelo laboratorio PARC da Xerox (o mesmo laborátorio responsável pelo desenvolvimento da linguagem de programação Smalltalk). Esse paradigma foi apresentado pela primeira vez em (KICZALES et al., 1997), onde fora apresentado pela primeira vez a ideia de aspectos para facilitar o reuso de código que pertence a uma gama de classes independentes.

De lá pra cá a teoria de orientação a aspectos foi aprimorada e novos conceitos foram definidos. Abaixo encontram-se explicações sucintas de cada um dos principais conceitos da orientação à aspectos, baseados nas definições expostas em (PÅHLSSON, 2002).

O primeiro conceito a ser apresentado é o conceito de aviso *advice*. Um aviso define a nova lógica de programação a ser injetada em pontos de execução especificos. Um aviso geralmente tem tipos especificos, que determinam a maneira de injeção da lógica definida. Pode-se citar como exemplos de tipo de avisos, os que injetam lógica antes da execução de um método, ou que injetam lógica ao redor da execução de um método.

Para a definição do ponto de execução onde um aviso será injetado, o paradigma de programação orientada a aspecto define duas estruturas, chamadas de pontos de junção

[5] Anteriormente (versão 1.4 da especificação UML) esses atributos eram chamados de *tag-values*

(*join points*) e pontos de corte (*pointcuts*). Um ponto de junção é basicamente a definição de um ponto de execução no código. Pontos de junção têm tipos bem definidos, derivado das estruturas presentes na linguagem de programação (um exemplo de um tipo de ponto de execução é a execução de um método). Diversos pontos de junção podem ser agregados, indicando um ponto de execução exato. A essa agregação é dado o nome de ponto de corte.

Definidos os conceitos de aviso, ponto de junção e ponto de corte, pode-se definir aspecto como um agrupamento modular de avisos, que são injetados em pontos de execução definidos pelos pontos de corte, definidos através de uma agregação de pontos de junção.

Após a definição dos aspectos, é executado o processo de costura (*weaving*) dos aspectos nas classes propriamente ditas. O processo de costura pode ser executado tando em tempo de compilação (através de um compilador especial) ou em tempo de execução. Durante o processo de costura, o código[6] original é varrido a procura dos pontos de corte associados aos avisos declarados nos aspectos sendo costurados. Caso um ponto de corte seja encontrado, a lógica correspondente ao aviso é injetada. Ao final do processo, um novo código, com os aspetos costurados, é emitido.

Uma implementação moderna de uma linguagem para a definição de aspectos é a linguagem AspectJ[7] , devenvolvida pela IBM no projeto Eclipse. O AspectJ permite a utilização de conceitos de programação orientada a aspectos na linguagem Java, de maneira compativel com o ferramental já utilizado pelos desenvolvedores.

2.2 Tipagem

Nesta seção serão apresentados alguns conceitos relativos à teoria de tipos, no âmbito da ciências da computação, no intuito de prover uma base teórica para o entendimento deste trabalho. A teoria de tipos é um tópico abrangente, que envolve campos de estudos como a lógica, a matemática e a filosofia (PIERCE, 2002, p.2). Não faz parte do escopo desta seção um tratamento profundo do tema tanto no âmbito das ciências da computação, quantos em outros campos de estudo.

2.2.1 Sistema de tipos

Um sistema de tipos é um mecanismo que tem como principal é objetivo o aumento da robustez de programas de computador. Segundo (PIERCE, 2002), pode-se definir um sistema de tipos como :

[6] Tanto código-fonte como código-objeto, dependento do momento em que o processo de costura é executado

[7] The AspectJ Project. Disponível em: ⟨http://www.eclipse.org/aspectj/⟩.

> "Um sistema de tipo é um método sintático tratável para a prova da inexistência de certos comportamentos de um programa, de acordo com tipos de valores por ele computados." (PIERCE, 2002, p. 1, tradução do autor)

Além da segurança, a adoção de um sistema de tipos trás inúmeras outras vantagens. Segundo (BRUCE, 2002, p. 7), pode-se citar como vantagens da utilização de um sistema de tipos :

a) Otimização;

b) Documentação;

c) Abstração.

Segundo (BRUCE, 2002), é possível categorizar os sistemas de linguagens orientadas à objetos de acordo com o momento no qual o sistema de tipos executa suas verificações. De acordo com esse método de categorização, pode-se dividir os sistemas de tipos em dois grandes grupos : os sistemas de tipo estáticos e os dinâmicos.

Em uma linguagem de programação munida de um sistema de tipos estático, todas as expressões de um programa podem ser associadas, em tempo de compilação, a um tipo. Com isso, é possível a realização de checagens relativas à manipulação de objetos em tempo de compilação. Já linguagens que utilizam-se de um sistema de tipos dinâmico, muitas checagens podem ser realizadas somente em tempo de execução. Essa característica faz certos problemas de inconsistência entre tipos só sejam descobertos no momento da execução do programa, acarretando defeitos.

Apesar das vantagens da utilização de um sistema de tipos para a criação de programas semanticamente válidos, ainda existem muitos problemas nos sistemas de tipos utilizados pelas linguagens orientadas a objetos comumente utilizadas. Segundo (BRUCE, 2002, p. 4) os sistemas de tipos inclusos nas linguagens de programação comerciais, tendem a ser ou muito inseguros, ou inflexíveis.

2.2.2 Tipo

Segundo (CARDELLI; WEGNER, 1985), pode-se definir um tipo como uma série de restrições bem definidas que regulam as interações com elementos de um determinado conjunto, conjunto este contendo todos os elementos válidos para um determinado tipo. O controle das interações tem como objetivo final o aumento da segurança do sistema como um todo, uma vez que manipulações arbritárias podem acarretar erros sutis, que podem por sua vez gerar falhas desastrosas ao sistema.

2.2.3 Contrato

É possível realizar o agrupamento dessas restrições que regem as interações entre elementos de um tipo em três categorias distintas de restrições, chamadas de pré-condição, pós-condição e invariantes, conforme definido por Bertrand Meyer em seu método de programação baseadas em contrato (MEYER, 1997, cap. 11)

Pré-condições são premissas que devem ser válidas para a correta execução de uma operação, podendo ter como alvo tanto o objeto receptor de uma mensagem quanto seus parâmetros. De maneira análoga às pré-condições, as pós-condições são premissas que têm como alvo tanto o resultado de uma operação quanto o objeto executor da operação. Enquanto pré e pós-condições têm como alvo a regulação do uso de operações, as invariantes têm como alvo o controle da consistência do estado de um objeto entre invocações das operações.

Para efeito de alinhamento de nomenclatura, será chamado de contrato um conjunto de pré-condições, pós-condições e invariantes, associadas a um tipo, sendo semanticamente relacionadas. Um contrato pode ser definido tanto de maneira formal ou de maneira informal.

2.2.4 Subtipo

Conforme descrito anteriormente, um tipo pode ser visto como um conjunto, logo um subtipo pode ser visto como um subconjunto de elementos pertencentes a um tipo (CARDELLI; WEGNER, 1985), em em que os contratos que regem as interações com seus elementos são estendidos seguindo determinada regra. Barbara Liskov e Jannette Wing elaboraram um conjunto de regras (LISKOV; WING, 1994) que devem reger a relação de herança de classes. A regra diz que os contratos associados à superclasse devem ser estendidos pela subclasse de modo que instâncias da subclasses sejam de um subtipo das instâncias da superclasse (essa relação entre extensão de contrato e subtipagem é chamada de Subtipagem Comportamental).

Segundo as regras definidas por Liskov e Wing, para que uma relação de herança equivala a uma relação de subtipo, é necessário que os contratos definidos no supertipo não tenham suas pré-condiçoes fortalecidas (porém podem ser relaxadas). Também é necessário que as pós-condições não sejam relaxadas (porém podem ser fortalecidas) e as invariantes sejam mantidas.

Princípio da substituição de Liskov

Conforme postulado por Barbara Liskov e Jannette Wing (LISKOV; WING, 1994), a substituição de objetos em um programa por outros só é segura quando o objeto sendo

substituído é do mesmo tipo (ou de um supertipo) do objeto a ser utilizado. A esse principio foi dado o nome de Princípio da substituição de Liskov.

Porém, em linguagens com tipagem estática[8], são utilizadas somente as características estruturais pelo sistema de tipos para verificar a validade de uma substituição. Logo, quando a relação de herança não segue as regras definidas na seção anterior, é possível a utilização de instâncias que se comportam de maneira diferente (não sendo pertencentes ao mesmo tipo). Quando isso acontece, diz-se que uma violação do Princípio da substituição de Liskov ocorrera.

2.3 A linguagem de programação utilizada na solução proposta

Tendo em vista o grande leque de variações permitidas pela orientação a objetos, devemos situar mais precisamente o posicionamento da solução.

A solução para o problema de igualdade de objetos apresentada neste trabalho será desenvolvida na linguagem de programação Java. Essa linguagem de programação é baseada na sintaxe e nos principios de programação das linguagems da familia Algol. Sendo assim, os conceitos apresentados aqui podem ser aplicados em outras linguagens dessa família, como, por exemplo, o C# e o C++.

A linguagem de programação Java é uma linguagem orientada a objetos baseadas em classes, com herança simples de classes e múltipla de interfaces., não havendo suporte para *Mixins*.

Java dá suporte à covariância de tipo do retorno de operações (é possível uma classe rescrever um método definido em uma super classe definido um tipo de retorno mais especifico), porém não há suporte para a contravariância de tipos dos parâmetros de uma operação (nesse caso, o novo método é tratado como um método sobrecarregado e não um método reescrito). O sistema de tipos da linguagem Java é estático, e violações do Princípio da Substituição de Liskov podem ocorrer, uma vez que não existe nenhum mecanismo de verificação que obrigue que uma extensão de uma classe gere obrigatoriamente um subtipo.

Conforme previamente comentado, as instâncias de objetos na linguagem Java são acessadas através de referências, que são utilizadas pelo mecanismo de coleta de lixo (*garbage collection*) para a seleção e a remoção de instâncias de objetos não referenciadas.

Além disso, a linguagem Java da suporte à programação baseada em atributos, através do uso de anotações para a definição de metadados sobre as classes e seus elementos. O uso de anotações é regido pelo sistema de tipos, permitindo assim checagens sobre sua correta

[8] Também ocorre em linguagens de tipagem fraca. O único tipo de linguagem onde esse problema não ocorre é em uma linguagem de tipagem forte, onde o Princípio da substituição de Liskov não pode ser violado

utilização em tempo de compilação. Conforme experimento realizado por (GUERRA; FERNANDES, 2013), o uso da programação baseada em atributos acarreta uma maior produtividade e maior facilidade de uso em relação a técnicas tradicionais de programação.

Outro ponto positivo na utilização da linguagem Java é a presença de uma extensão madura para a utilização de conceitos de programação orientada a aspectos, chamada AspectJ[9] . Além dos conceitos da programação orientadas a aspectos, discutidos anteriormente, essa extensão permite a implementação de *mixins* e a utilização de pontos de junções exclusivos para verificação da presença de anotações em elementos estruturais da linguagem Java.

O mecanismo de despacho de mensagens utilizado pela linguagem de programação Java é o mecanismo de despacho simples. Conforme previamente descrito, no momento da execução o corpo do método a ser executado é selecionado em função do objeto recebendo a mensagem. Conforme (BRUCE et al., 1995), essa característica em uma linguagem de programação orientada a objetos acarreta na impossibilidade da implementação de operações simétricas[10], quando se utiliza o mecanismo de herança.

[9] The AspectJ Project. Disponível em: ⟨http://www.eclipse.org/aspectj/⟩.

[10] As operações simétricas seguem o modelo da proposição $\forall x \forall y (P(x, y) \leftrightarrow P(y, x))$, onde a validade da relação $P(x, y)$ implica na validade da relação $P(y, x)$, e vice-versa.

3 Igualdade de objetos

Nesse capitulo, será tratado o conceito da igualdade de objetos, tendo como foco a linguagem de programação Java. Serão expostos os conceitos e restrições no contrato que as operações associadas ao mecanismo de verificação da igualdade devem realizar. Também serão abordados resultados mostrando os impactos associados a falhas na realização desses contratos em diversas aplicações de código aberto disponíveis.

3.1 O conceito da igualdade de objetos

A etimologia da palavra igualdade remete ao latim *aequalis* (comparável, similar) que por sua vez deriva da palavra *aequus* (igual). De maneira informal, a igualdade entre dois elementos é uma relação, onde os participantes são indiscerníveis entre si. A interpretação do que são dois elementos indiscerníveis é dependente do contexto ou campo de estudo, podendo ser mais rígida (no campo da lógica) ou branda (na aritmética ou na geometria).

3.1.1 Definição

No âmbito da lógica, pode-se definir formalmente a relação de igualdade entre dois elementos através da proposição $\forall x \forall y((P(x) \wedge P(y)) \leftrightarrow (x = y))$, ou, em outras palavras, dois objetos[1] (x e y) são considerados iguais se e somente se, tiverem as mesmas propriedades. Essa definição de igualdade foi proposta por Gottfried Wilhelm Leibniz em (LEIBNIZ, 1976, cap. 35), e é, por essa razão, chamada de Lei de Leibniz.

Segundo (TARSKI, 1994), é possível derivar, através da Lei de Leibniz, algumas propriedades da relação de igualdade, que são utilizados como axiomas [2] em provas matemáticas. Entre as propriedades que podem ser derivadas através da Lei de Leibniz encontra-se a propriedade de reflexividade, que dita que, dado um objeto x ele deve ser, impreterivelmente, considerado igual a eles mesmo. Em lógica proposicional de primeira ordem, essa propriedade pode ser expressa através da proposição $\forall x(x = x)$.

Outra propriedade derivada através da Lei de Leibniz é a simetria. A simetria dita que dados dois objetos x e y, caso exista uma relação binária de igualdade entre x e y, também deverá existir uma relação equivalente entre os elementos y e x e vice-versa, conforme expresso pela proposição $\forall x \forall y((x = y) \leftrightarrow (y = x))$.

[1] Nesse contexto, objeto deve ser entendido como qualquer entidade contreta (como por exemplo um carro ou uma pessoa) ou abstrata (como por exemplo um número ou uma expressão algébrica)

[2] Pode-se citar como exemplo os Axiomas de Peanno (PEANO, 1889), que define como axiomas as propriedades de reflexividade, simetria e transitividade nas relações de equivalência entre elementos de $\mathbb{N}$.

Também é possível, através da Lei de Leibniz, derivar a propriedade de transitividade, inerente à relação de igualdade entre dois objetos. Essa propriedade (expressa através da proposição $\forall x \forall y \forall z((x = y) \wedge (y = z) \rightarrow (x = z)))$ dita que, dados três objetos x, y e z, caso x seja considerado equivalente a y e por sua vez seja y considerado igual a z, então x deve ser obrigatoriamente igual z.

Conforme visto, a formulação da igualdade proposta pela Lei de Leibniz (LEIBNIZ, 1976, cap. 35) é baseado no conceito de propriedades, inerente aos objetos participantes da relação, segundo esse conceito, para que dois objetos sejam considerados iguais, ambos devem ter todas suas propriedades equivalentes. Segundo a definição da lei, para a verificação da igualdade são utilizadas todas as propriedades do universo de discurso, o que, na prática faria que só seriam considerados iguais objetos representando a mesma entidade concreta ou abstrata, por esse motivo também é dada a esse lei o nome de Princípio da Identidade dos Indiscerníveis.

Porém, em muitos caso é necessário a adoção de uma visão mais pragmática com relação ao conceito de igualdade. Pode-se citar como exemplo de campos onde foram adotadas visões mais brandas do conceito de igualdade a aritmética e a geometria.

> "Deve se notar, que existiram (e possivelmente ainda existam) matemáticos que se opuseram ao pondo de vista adotado aqui – não identificando o símbolo = da aritmética com o conceito da identidade lógica; eles não consideram que números iguais sejam necessariamente idênticos e então assumem a noção de igualdade como um conceito especificamente aritmético.
>
> [...]
>
> Com relação à noção de igualdade na geometria, tradicionalmente existe um ponto de vista completamente diferente. Muitas vezes é dito que duas figuras geométricas, como por exemplo dois segmentos de linha, ou dois ângulos, ou dois polígonos são iguais no sentido que elas são congruentes, e não que elas têm a mesma identidade. Nesse caso, o interlocutor só quer salientar que as figuras tem o mesmo tamanho e forma, ou figurativamente, dizer que, caso uma das figuras fosse colocada em cima da outra, ela a cobriria perfeitamente." (TARSKI, 1994, p. 56 et seq., tradução do autor)

Tendo esses cenários em vista, é possível refinar a Lei de Leibniz, indicando que as propriedades utilizadas para a verificação da igualdade não são mais compostas por todo o universo do discurso e sim de um subconjunto de propriedades, dependendo do contexto de verificação da igualdade. Porém, o grau de expressividade de proposições em lógica de primeira ordem é insuficiente para a expressão desse conceito.

Utilizando-se da maior expressividade de proposições em lógica de segunda ordem, é possível representar essa definição mais pragmática da igualdade através da proposição $\exists A(P \in A) \rightarrow \forall x \forall y(P(x) \wedge P(y) \leftrightarrow x = y)$ que dita que dois objetos são considerados iguais, se, e somente se, ambos tiverem as mesmas propriedades dentro um conjunto finito

de propriedades (conjunto A), consideradas importantes para um determinado contexto de comparação.

No âmbito das ciências da computação e da orientação a objetos, existem duas vertentes contrastantes no que tange o método de comparação entre objetos. A primeira vertente, representada em (LISKOV; GUTTAG, 1986, p. 93) adota a versão pura do princípio da identidade dos indiscerníveis.

> "Nós adotamos uma convenção reservando a operação *equals()* para a relação de igualdade; deve ser impossível se distinguir entre dois objetos considerados iguais. Logo, dois objetos mutável so podem ser iguais se e somente se eles forem o mesmo objeto; de outro modo seria possível a distinção entre eles através da invocação de uma operação com efeitos colaterais em somente uma das instâncias." (LISKOV; GUTTAG, 1986, p. 93 et seq., tradução do autor)

A outra vertente enfoque baseado na visão mais pragmática a Lei de Leibniz, baseada em conjuntos distintos de propriedades em função do contexto de comparação. Nesse trabalho será dado um foco maior as questões referente à esta segunda vertente de implementação da igualdade entre objetos, que, apesar de ter um grau de complexidade maior na implementação, permite uma maior flexibilidade ao programador e uma facilidade maior de integração em *frameworks* de mercado, conforme será visto.

3.1.2 Igualdade vs Identidade

Definido o conceito de igualdade como a presença de propriedades de um determinado conceito de comparação, pode-se definir também o conceito de identidade entre referências de objetos. Diz-se que, quando duas referências têm a mesma identidade, elas referenciam exatamente a mesma instância pertencente ao conjunto de objetos de um sistema baseado em software. Em outras palavras, as duas referências apontam para o mesmo endereço de memória. Vale notar que, o método de verificação da igualdade padrão da linguagem Java, é baseado na apenas na identidade dos operandos, e não em quaisquer propriedades que eles por ventura tenham.

Assim como a relação de igualdade, a relação de identidade também tem como propriedades a reflexividade a simetria e a transitividade[3]: a comparação entre identidades é executada através da verificação da igualdade entre os endereços de memória alocada pelas instâncias referenciadas pelos operandos. Uma vez que esse endereços são números naturais, pode se deduzir que uma relação de igualdade entre dois endereços tenham as propriedades de reflexividade, simetria e transitividade, uma vez que essas propriedades

[3] Além da dedução baseada na comparação e endereços de mémoria e nos axiomas de Peano (PEANO, 1889), pode-se inferir as propriedades dessa relação através da derivação do princípio da identidade dos indiscerníveis (LEIBNIZ, 1976, cap. 35).

são, conforme (PEANO, 1889, p. 1), consideradas axiomáticas em relações de igualdade entre esse tipo de números.

Para efeito de notação, usaremos o sinal "=" para indicar uma relação de igualdade e "==" para indicar uma relação de identidade.

3.1.3 Estratégias de Definição de Igualdade

Segundo (RUPAKHETI, 2010), pode-se definir três categoria de estratégias de igualdade. A primeira é chamada de igualdade compatível com tipo (no original chamada de *type-compatible equality*), em que são considerados equivalentes somente objetos que são instâncias da mesma classe. A a implementação da lógica do mecanismo de verificação de igualdade utilizando essa estratégia é trivial.

A segunda categoria é chamada de igualdade incompatível com tipo (no original denominada *type-incompatible equality*), em que são considerados iguais objetos pertencentes à mesma hierarquia de classes. Quando é utilizada uma estratégia de comparação pertencente à essa categoria e são definidos novos atributos nas subclasses, torna-se difícil a manutenção do contrato[4] da operação de verificação de igualdade.

A última categoria é a chamada categoria híbrida, em que alguns segmentos de uma hierarquia são utilizam igualdades compatíveis com tipo e outros segmentos utilizam igualdades incompatíveis com tipo. De maneira análoga à categoria de igualdade incompatível com tipo, a utilização dessa estratégia acarreta problemas na manutenção do contrato de igualdade.

3.1.4 Cálculo de Ordem Natural

Uma mecanismo intimamente ligado à verificação da igualdade de objetos é o mecanismo de ordenação natural de objetos. Esse mecanismo é amplamente utilizado pelas classes de coleção da biblioteca do Java nas operações de ordenação.

A implementação desse mecanismo se dá através da realização da interface *Comparable*, que por sua vez define a operação *compareTo()*. Caso o primeiro operando[5] fornecido seja considerado menor que o segundo operando fornecidos à operação, está deverá retornar um valor menor que zero, caso contrário deve ser retornado valor maior que zero. Caso o valor zero seja retornado, pode-se assumir que os operandos foram considerados iguais.

Apesar de retornos inconsistentes com os resultados obtidos através das invocações das operações de verificação da igualdade (*equals()* e*hashCode()*) não configurarem uma

[4] Esse contrato será descrito com maiores detalhes depois.

[5] Para efeito de alinhamento da nomenclatura, tanto o objeto receptor da mensagem quando o objeto passado como parâmetro são considerados operandos.

violação do contrato que rege instâncias realizando interface *Comparable* (conforme documentação[6] da linguagem). Essa prática não é recomendada.

> "Uma classe cujo método *compareTo()* impões uma ordem inconsistente com *equals()* ainda funciona, mas coleções ordenadas armazenando suas instâncias podem não obedecer o contrato da sua interfaces." (BLOCH, 2008, p. 64, tradução do autor)

Em outras palavras, essa inconsistência acarreta, de uma maneira indireta, uma violação do principio da substituição de Liskov, visto anteriormente.

De maneira geral será adotada a premissa de que os resultados obtidos nas invocações dessa operação devem ser consistentes com os resultados obtidos nas invocações da operação *equals()*. Essa consistência entre as operações pode ser expressa através da proposição $\forall x \forall y((x = y) \rightarrow (x.compareTo(y) = 0))$, que exprime o que pode ser chamado de cálculo da ordem natural em Java, que estabele a igualdade de dois objetos x e y : para eles sejam considerados iguais, o resultado da invocação da operação *compareTo()* tendo-os como operandos deverá ser zero.

Cabe aqui uma breve observação da perspectiva teórica. Este trabalho focaliza o tema da igualdade de objetos sob um ponto de vista pragmático. Um enfoque mais rigoroso exigira um tratamento no âmbito da *equivalência de tipos*, um assunto teórico que apresenta varias dificuldades e que está além do escopo deste presente trabalho, sendo um candidato a trabalho futuro.

Uma das questões relativas à equivalência de tipos, de interesse para o problema aqui tratado, diz respeito ao polimorfismo. A equivalência de tipos é a relação mais básica entre expressões de tipos. Muitas vezes tais expressões podem ser especificadas em uma gramática livre de contexto, porém o contexto do problema que este trabalho pretende tratar também necessita a análise semântica dos termos das expressões, conforme (CARDELLI; WEGNER, 1985).

Por outro lado, a necessidade de lidar com o polimorfismo, em suas variadas formas, que têm suporte nas linguagens baseadas em classe mais utilizadas comercialmente, requer *relações de similaridade* entre as expressões de tipo. Relações desse tipo são mais fracas do que as relações de equivalência, por permitirem o uso de subtipos (LISKOV; WING, 1994).

Em suma, um trabalho futuro que requer um estudo teórico mais aprofundado, poderia situar mais rigorosamente o problema estudado nesse trabalho e verificar a solução proposta no capítulo 5 de maneira mais formal. Dentre outras questões, uma de especial interesse é a definição mais rigorosa da lógica da comparação entre objetos aqui propostas

[6] Comparable (Java Platform SE 7). Disponível em: ⟨http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html⟩.

em termos de definições de reduções das propriedades formais da igualdade de objetos em expressões computacionais. Por exemplo, reduções que permitam reduzir as propriedades de simetria e transitividades, inerentes à verificação de igualdade entre objetos, em expressões computacionais tratáveis, como proposto em (PIERCE, 2002).

3.2 Contrato de Igualdade

Na linguagem de programação Java, as operações relativas à igualdade são regidas por um contrato definido de maneira informal (BLOCH, 2008, cap. 3). É de suma importância a correta implementação desse contrato, uma vez que falhas no seu cumprimento podem acarretar falhas sutis quando objetos são utilizados por elementos que dependem do uso das operações de igualdade.

Conforme definido anteriormente, será adotada a premissa de que os resultados obtidos no cálculo da ordem natural serão consistentes com os resultados das verificações de igualdade. Essa característica faz com que o contrato de igualdade tenha impacto direto nas implementações do cálculo da ordem natural.

Pode-se subdividir o contrato que rege as operações pertencentes ao mecanismos de igualdade da linguagem Java em cinco sub-contratos : Reflexividade, Simetria, Transitividade, Consistência e Não-Nulabilidade, cada um abrangendo uma propriedade aspecto especifica. Os subcontratos da Reflexividade, Simetria e Transitividade são relativos as propriedades homônimas derivadas das Lei de Leibniz e assumidas como axiomas, já os subcontratos de Consistência e Não-Nulabilidade tem como foco questões particulares à linguagem Java. Segue uma descrição detalhada delees, bem como as peculiaridades das implementações, tanto da verificação de igualdade quanto do cálculo da ordem natural.

3.2.1 Reflexividade

A reflexividade estabelece que um objeto sempre deverá ser igual a ele mesmo. A implementação do subcontrato da reflexividade é trival, segundo Bloch :

> "É dificil imaginar a violação desse contrato. Se você o violasse e em seguida adicionasse uma instância da sua classe à uma coleção, uma chamada ao método *contains* indicaria que a coleção não contém a instância recém-adicionada." (BLOCH, 2008, p. 35, tradução do autor)

De maneira análoga à operação *equals()*, a reflexividade também deve ser aplicada à operação *compareTo()*, Todas as invocações de *compareTo()*, em que os operandos são referências à uma mesma instância, terão como resultado o valor zero, indicando, de maneira consistente com o resultado da operação *equals()*, que os operandos fornecidos são considerados equivalentes.

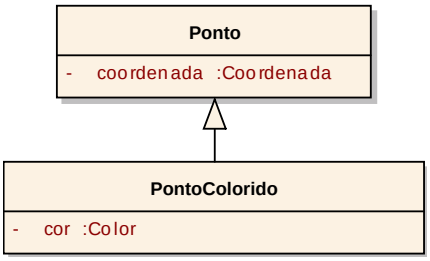
3.2.2 Simetria

O subcontrato de simetria dita que, independentemente da ordem dos operandos, o resultado das operações relativas à verificação de igualdade devem ser consistente. Por exemplo, dado dois objetos (chamados x e y), o resultado obtido pela expressão $x = y$ (em Java essa expressão se traduz na declaração *x.equals(y)*) deverá ser idêntico ao obtido através da expressão $y = x$.

Ao contrário do subcontrato da reflexividade, é factível que um programador quebre não-intencionalmente a simetria da verificação de igualdade. Segundo Bloch (2008, p. 35, tradução do autor) "Ao contrário do primeiro requisito(reflexividade), não é difícil imaginar a violação não-intencional deste.".

Além dos problemas de codificação, existe também a questão da manutenção da simetria quando existe uma relação de herança entre as classes das quais os operandos utilizados são instâncias. Pode-se citar como exemplo uma hierarquia de classes composta de duas classes: *Ponto* e *PontoColorido*). *Ponto* é composto do atributo *coordenada*, já a classe *PontoColorido* também tem o atributo *cor*. Existe uma relação de herança entre as classes *Ponto* e *PontoColorido*, em que *Ponto* é uma superclasse de *PontoColorido* e tanto as informações de coordenada e cor são importantes para a verificação da igualdade. A figura 2 apresenta um diagrama de classe expondo essa estrutura.

Figura 2 – Diagrama de Classe - Simetria



Levando em conta as informações fornecidas e a estrutura estática disposta na figura 2, pode-se implementar o método *equals()* das duas classes com ilustrado na listagem 1[7]:

Listagem 1 – Implementação de *equals()* - *Ponto* e *PontoColorido*

```
public class Ponto {
  // ...
  public boolean equals(Object o) {
    if (o instanceof[8] Ponto) {
```

[7] Para ser breve, foram omitidas do código as declarações de atributos, conversões de tipos e checagens de referências nulas

[8] O operador *instanceof* retorna *true* caso a instância referenciada pelo primeiro operando seja do tipo representado pela classe fornecida como segundo operando (no caso, retornará *true* caso a classe

```
        return coordenada.equals(o.coordenada);
      }
      return false;
    }
  }

  public class PontoColorido extends Ponto{
    // ...
    public boolean equals(Object o) {
      if (o instanceof PontoColorido) {
        return super.equals(o) && cor.equals(o.cor);
      }
      return false;
    }
  }
```

De acordo com o subcontrato de simetria, os resultados das expressões $x = y$ e $y = x$ devem ser iguais. Imaginando que os objetos x e y sejam instâncias de *Ponto* e *PontoColorido* respectivamente, basta, para que a expressão $x = y$ seja verdadeira, que o valor de coordenada de ambas instâncias sejam iguais. Porém a comparação $y = x$ retornará *false* independentemente dos valores associados a cada uma das instâncias, uma vez que na codificação da operação *equals()* definida na classe *PontoColorido* é feita uma verificação obrigando que a instância fornecida como argumento à operação seja do tipo *PontoColorido*.

Conforme exemplificado, a implementação do subcontrato de simetria pode ser complicada quando existe uma relação de tipo e subtipo dos operandos, já que a lógica a ser executada para a verificação depende não somente de um operando, mas de ambos. Sobre a questão Joshua Bloch afirma :

> "Simplesmente não existe uma maneira de estender uma classe concreta e adicionar um aspecto (no caso, uma nova informação para comparação) e preservar contrato da operação *equals()*". (BLOCH, 2008, p. 38, tradução do autor)

Essa afirmação, embora verdadeira, aplica-se apenas a linguagens pertencentes a um subconjunto de linguagens orientadas à objeto : o subconjunto das linguagens com despacho simples. Esse problema, e possíveis soluções, serão examinados com mais detalhes na seção 3.4.2.

A propriedade de simetria da verificação da igualdade também se estende ao cálculo da ordem natural, regido pela operação *compareTo()*. A simetria do cálculo da ordem natural pode ser expressa pela proposição $\forall x \forall y \forall z(((x < y) \leftrightarrow (y > x)) \vee ((x = y) \leftrightarrow (y = x))$, indicando que, dadas duas instâncias x e y, caso x seja considerado menor que y então o resultado da comparação $y > x$ deverá indicar que y é maior que x, ou, caso x seja igual a y, então y também deve ser considerado igual a x.

implementando a instância referenciada por *o* seja, direta ou indiretamente, uma subclasse de *Ponto*).

3.2.3 Transitividade

O terceiro subcontrato do contrato que rege as operações de verificação de igualdade é o subcontrato da transitividade. Conforme exposto anteriormente, o subcontrato da transitividade pode ser expresso através da proposição $\forall x \forall y \forall z((x = y) \wedge (y = z) \rightarrow (x = z))$, ou seja, dados três objetos (x, y, z), se o objeto x for igual ao objeto y e o objeto y por sua vez for igual ao objeto z, então o objeto x deve ser considerado igual ao objeto z.

Da mesma maneira que o subcontrato de simetria, existem complicações na manutenção do subcontrato de transitividade quando existe uma relação de herança entre as classes, e nesta relação de herança são incluídas novas informações a serem utilizadas na comparação de igualdade. Nesses casos, é possível a manutenção ou da simetria ou da transitividade, mas não ambos. Essa afirmação é corroborada por Bloch com alguns exemplos de idiomas de implementação do método *equals()* (BLOCH, 2008, p. 36 et. seq.).

Conforme previamente exposto, a causa dessas complicações reside no mecanismo de despacho de mensagens. Logo é imperativo a utilização (ou simulação) de um mecanismo de despacho múltiplo para o cumprimento do subcontrato de transitividade.

Como em todos os subcontratos do contrato de que rege a verificação da igualdade, a propriedade de transitividade também se reflete no cálculo da ordem natural, através de duas premissas (BLOCH, 2008, p. 63) que devem ser cumpridas para a garantia da transitividade do cálculo da ordem natural.

A primeira premissa da transitividade do cálculo da ordem natural pode ser expressa pela proposição $\forall x \forall y \forall z(((x < y) \wedge (y < z)) \rightarrow (x < z))$, ou, em outras palavras, caso o objeto x seja considerado menor que o objeto y e este, por sua vez, seja considerado menor que o objeto z, então x deve ser impreterivelmente menor que o objeto z.

Já a segunda premissa dita que, dados três objetos (x, y e z), caso x seja igual a y e seja menor que z, então x também deve ser considerado menor que z. Pode-se exprimir essa premissa pela preposição lógica de primeira ordem $\forall x \forall y \forall z(((x = y) \wedge (y < z)) \rightarrow (x < z))$

3.2.4 Consistência

Segundo Bloch, o contrato de consistência dita que :

> "[...] se dois objetos são considerados iguais, eles devem ser considerados sempre iguais, a não ser que um deles (ou ambos) seja modificado.". (BLOCH, 2008, p. 41, tradução do autor)

Para os fins deste trabalho, essa definição será estendida para abranger também o cálculo da ordem natural: Para que a operação seja considerada consistente, caso um objeto x seja considerado menor que um objeto y (ou vice-versa), todas as comparações

subsequentes devem chegar a mesma conclusão, a menos que um deles (ou ambos) foi modificado.

Um exemplo dado por Bloch (BLOCH, 2008, p. 41),com o intuito de ilustrar uma violação de consistência da verificação de igualdade, é o da classe *java.net.URL*, que encapsula uma URL associada a um recurso qualquer. A implementação do método *equals()* dessa classe depende do mecanismo de resolução de nomes (DNS) para converter os nomes dos hosts associados às URLs para endereços IPs, que por sua vez serão utilizados na comparação. Porém, o processo de resolução de nomes não é determinístico, já que não é garantido a obtenção do mesmo endereço IP em invocações futuras.

Essa característica de não-determinismo, inerente ao processo de resolução de nomes, faz com que a operação *equals()* da classe *URL* viole o subcontrato de consistência.

3.2.5 Não-Nulabilidade

O subcontrato da não-nulabilidade, conforme (BLOCH, 2008), dita que, todo e qualquer objeto deve ser considerado diferente de uma referência nula. Em outras palavras, a expressão $x.equals(null)$ deve sempre retornar o valor *false*, independentemente dos valores e da semântica atribuída a x.

Apesar desse subcontrato não apresentar grande complexidade de implementação, ele é muitas vezes esquecido pelos programadores, o que acarretando erros em tempo de execução[9]. O estudo[10] descrito em (VAZIRI et al., 2007) identifica problemas de violação da propriedade de não-nulabilidade nos *softwares* analisados.

O comportamento recomendado pela documentação oficial da operação *compareTo()*[11] recomenda que chamadas à esta operação, tendo como argumento uma referência nula levantem uma exceção do tipo *NullPointerException*. Porém esse comportamento vai contra a premissa de consistência entre as operações *equals()* e *compareTo()*.

Assumindo-se a premissa de consistência, pode-se assumir a validade da proposição $\forall x \forall y(\neg(x.equals(y))) \leftrightarrow (x.compareTo(y) \neq 0)$, logo o resultado experado de $x.compareTo(null)$ é um valor diferente de zero.

Além das operações *equals()* e *compareTo()*, existe uma terceira operação que deve ser implementada, para uma correta implementação do mecanismo de verificação da igualdade disponível na linguagem Java. Essa operação é a operação *hashCode()*, definida na classe *Object*.

De maneira análoga às operações verificação da igualdade e do cálculo da ordem na-

[9] Na forma de *NullPointerExceptions*

[10] Reproduzido na tabela 1

[11] Comparable (Java Platform SE 7). Disponível em: ⟨http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html⟩.

tural, existe uma relação implícita de dependência entre as semânticas nas operações *hashCode()* e *equals()*, que pode ser representada pela proposição $\forall x \forall y (x.equals(y)) \rightarrow (x.hashCode() = y.hashCode())$.

Caso dois objetos sejam considerados iguais, os resultados obtidos nas invocações de *hashCode()* destes objetos também deve ser considerados iguais, a menos que nesse meio tempo alguma informação importante ao processo de verificação de igualdade seja modificada[12]. Caso essa premissa não seja respeitada, ocorrerão falhas quando os objetos forem utilizados em classes importantes dos recursos de coleções da linguagem Java, como por exemplo os *arrays* associativos (*java.util.HashMap*) (BLOCH, 2008, p. 45).

Conforme (BLOCH, 2008, p. 47), não é requerido que objetos considerados diferentes retornem valores diferentes quando invocada a operação *hashCode()*. Levando isso em conta, uma função de *hash* retornando uma constante para todo e qualquer operando é uma função *hash* que semânticamente válida, porém seu uso acarretaria um impacto sensível ao desempenho[13] do *software*.

3.3 Uso de coleções com restrições de igualdade

Uma questão intimamente ligada ao conceito de verificação de igualdade entre objetos na linguagem Java é a utilização desses objetos em coleções pertencentes a sua biblioteca de classes. Problemas na implementação dos métodos pertencentes ao mecanismo de verificação de igualdade da linguagem Java. Sobre isso é citado por Bloch :

> "Uma fonte comum de falhas é a não sobrescrita do método *hashCode()*. É obrigatória a sobrescrita de *hashCode()* em todas as classes que sobrescrevem *equals()*. Falhas no cumprimento dessa premissa resultarão em uma violação do contrato geral que rege *hashCode()*, o que impedirá a classe de funcionar junto com todas as coleções baseadas em *hash*, incluindo *HashMap*, *HashSet* e *Hashtable*." (BLOCH, 2008, p.45, tradução do autor)

Porem, só o cumprimento do contrato igualdade não é suficiente para a utilização de elementos em alguns tipos de coleção. O método *hashCode()* encapsula uma função *hash*, usada por estruturas de dados como tabelas *hash* para indexação interna. Para que uma busca subsequente seja bem-sucedida, é necessário que o valor retornado pela função *hash* seja igual ao valor obtido, ou, em outras palavras, que a chamada à função seja consistente.

[12] Conforme estabelecido pelo subcontrato de consistência das operações de igualdade.

[13] Por exemplo, quando um *array* associativo tem seus itens com essas característica, ele passa a funcionar como uma lista ligada, uma vez que todos os itens serão inseridos em um único *bucket*. Com isso, as operações de busca passam a assumir a complexidade $O(n)$, ao invés de $O(\log n)$ (LISCHNER, 2009, p. 450).

Porém, em razão da mutabilidade de objetos, é possível que essa chamada à função *hash* retornem valores distintos, caso hajam mudanças nas informações utilizadas pela lógica de verificação de igualdade[14] .

Esta característica tem impacto na utilização de algumas coleções pertencentes à biblioteca de coleções da linguagem Java. Coleções que herdam de *Set* têm restrições maiores sobre elementos que podem ser nelas inseridos, sem que haja nenhum tipo de aviso ou restrição de tipo.

> "Como não existe nenhum restrição de tipo para previnir a inclusão de elementos mutável em uma instância de *Set*, programadores devem ser cuidadosos no cumprimento desse contrato ou podem encontrar falahas sutis em seus programas. Java provê mecanismos para a definição de uma relação de equivalente pelo programador, mas coleções como *Set* têm contrato mais restritivo. Essa inconsistência quebra o polimorfismo. [...]" (NELSON; PEARCE; NOBLE, 2010, tradução do autor)

Conforme exposto, existe uma quebra de polimorfismo na relação de extensão entre as interfaces *Collection* e *Set*, que definem o contrato geral a ser cumprido por todas as coleções e o tipo de coleção representando conjuntos. No contrato definido em *Collection*, as operações de inclusão de elementos devem simplesmente inclui-los, sem realizar qualquer tipo de processamento em elementos já inclusos. Já o contrato regendo conjuntos devem remover elementos já incluso que por ventura sejam considerados equivalente a elementos sendo adicionados. Esse cenário indica uma clara violação ao principio da substituição de Liskov (LISKOV; WING, 1994), que dita que pré-condições não podem ser fortalecidas em subclasses.

Tendo esse cenário em vista, é proposto em (NELSON; PEARCE; NOBLE, 2010) uma categorização das coleções pertencentes à biblioteca inclusa na linguagem Java. Nessa categorização, as coleções são dividas em dois grandes grupos :

a) Coleções com restrições de igualdade, onde ficam proibidas alterações em atributos utilizado pela lógica de verificação da igualdade em itens inclusos, durante a permanência deles na coleção;

b) Coleções sem restrições de igualdade, onde tais restrições não existem.

Além da categorização das coleções, também é proposto em (NELSON; PEARCE; NOBLE, 2010) uma categorização de objetos, baseada em seu grau de mutabilidade.

a) Objetos imutáveis, onde é vetada a alteração de atributos de um objeto após a sua inicialização;

[14] Object (Java Platform SE 7). Disponível em: ⟨http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html⟩.

b) Objetos com estado mutável, onde é permitida a alterações de atributos de um objeto após sua inicialização, porém ainda é proibida a alteração de atributos utilizados pela lógica de verificação da igualdade do objeto;

c) Objetos mutáveis, onde não existam quaisquer restrições com relação a alterações de atributos pertencentes a um objeto, sejam eles importantes para lógica de verificação ou não.

Levando em conta as duas classificações apresentadas, pode-se traçar um perfil de uso para as coleções e seus elementos: em coleções com restrições de igualdade é permitido somente a inserção de elemento imutáveis ou com estado mutável, já em coleções sem restrições de igualdade podem ser inseridos quaisquer tipos de objetos.

3.4 Problemas encontrados em implementações

Conforme (VAZIRI et al., 2007), a implementação correta dos métodos associados ao mecanismo de verificação da igualdade da linguagem Java não é um processo trivial. Abaixo serão categorizados os problemas mais comumente encontrados em estudos de caso, abrangendo implementações do mecanismo da verificação da igualdade em *softwares* populares de código aberto.

3.4.1 Contrato não realizado

Em estudos de caso realizados (VAZIRI et al., 2007), (FRASER; ARCURI, 2013) e (RUPAKHETI; HOU, 2008), foram identificados e categorizados diversos problemas de quebra de contrato na implementação do mecanismo de comparação de objetos em projetos de código aberto escritos na linguagem Java.

No estudo de caso conduzido por (VAZIRI et al., 2007) foram escolhidos projetos de código-fonte aberto já consagrados, com um alto grau de uso. Nesse estudo foi detectada uma grande quantidade de falhas na realização do contrato de igualdade (basicamente metade dos projetos analisados tinham problemas na realização do contrato de igualdade). A tabela 1 apresenta um sumário dos resultados obtidos nesse estudo de caso.

Tabela 1 – Estudo de Caso (VAZIRI et al., 2007) - Sumário de Resultados.

benchmark	#equals()			#relation types				bugs/issues
	orig.	removed	%	total	value	sing.	non-value	
ant	12	9	75,00	9	3	0	6	H, M
bcel	20	14	70,00	22	11	5	11	F, H, M
hsqldb	12	2	16,70	2	0	0	2	E, F, M, S
javacup	14	11	78,60	11	8	2	3	H, M, T
jfreechart	46	33	71,70	40	40	1	0	H, M, S
lucene	27	27	100,00	27	23	0	4	E, M
pmd	12	5	41,70	5	3	2	2	E, F, M, N, S
shrike	32	32	100,00	61	55	3	6	M

Fonte: (VAZIRI et al., 2007, tab. 2)

Legenda: E=Implementações de *equals()* levantam exceções com argumentos inesperados; F=Implementações frágeis de *equals()* (ex .: resultado do método *toString()*); H=Contrato que rege interdependência entre *equals()* e *hashCode()* violado; M=Dependência de atributos mutáveis na lógica de verificação da igualdade; N=Violação da propriedade de não-nulabilidade em implementações de *equals()*; S=Violação da propriedade de simetria em implementações do método *equals()*; T=Checagens entre tipos incomparáveis.

O estudo de caso realizado por (FRASER; ARCURI, 2013) é baseado na utilização da ferramenta EvoSuite [15] em cem projetos disponíveis no site SourceForge [16] , escolhidos de maneira aleatória. Esse estudo de caso encontrou problemas na implementação da lógica de verificação da igualdades quanto à simetria e reflexividade, conforme o resumo dos resultados exibidos na tabela 2.

[15] EvoSuite | Automatic Test Suite Generation for Java. Disponível em: ⟨http://www.evosuite.org/⟩.

[16] SourceForge - Download, Develop and Publish Free Open Source Software. Disponível em: ⟨http://sourceforge.net/⟩.

Tabela 2 – Estudo de Caso (FRASER; ARCURI, 2013) - Sumário de Resultados.

Nome	As	Es	EH	EN	ES	HN	JCE	NPE	TSN	UE
GA	69	14	156	0	8	0	5068	2662	236	6352
Random	64	16	159	0	9	0	4938	2575	214	6336
TT	71	15	149	0	9	0	5192	2654	233	6440

Fonte: (FRASER; ARCURI, 2013, tab. 6)

Nota: Os itens As, JCE, NPE, TSN e UE, dispostos na tabela, não são pertinente as implementações de lógica de verificação da igualdade. Porém foram mantidos, para fins de consistência, na reprodução da tabela neste trabalho.

Legenda: Es=Reflexibilidade; EH=Interdependência entre *equals()* e *hashCode()*; EN=Não-Nulabilidade; ES=Simetria; HN=Operação *hashCode()* executada sem exceções.

Já no estudo de caso realizado por Rupaketi (RUPAKHETI; HOU, 2008), foi feita a análise empirica do código-fonte do JDK (versão 1.5) a procura de violações do contrato que rege o mecanismo de verificação da igualdade. Segundo esse estudo, cerca de 28% das implementações do mecanismo de comparação tinham algum tipo de violação de contrato. Também é citado em (RUPAKHETI; HOU, 2008).

> "O entendimento do problema (da verificação da igualdade) é tão vago que até IDEs populares como o Eclipse 3.3 e o NetBeans 6.0 geram código implementando o mecanismo de verificação com falhas através de seus assistentes (*wizards*)."(RUPAKHETI; HOU, 2008, tradução do autor)

3.4.2 Herança vs simetria e transitividade

Conforme previamente dito e exemplificado na seção 5.2, é impossível, em uma linguagem baseada em despacho simples, a implementação dos subcontratos de simetria e transitividade quando existe uma relação de herança entre os operandos, e novos atributos são adicionadas às comparações necessárias para a verificação de igualdade entre eles.

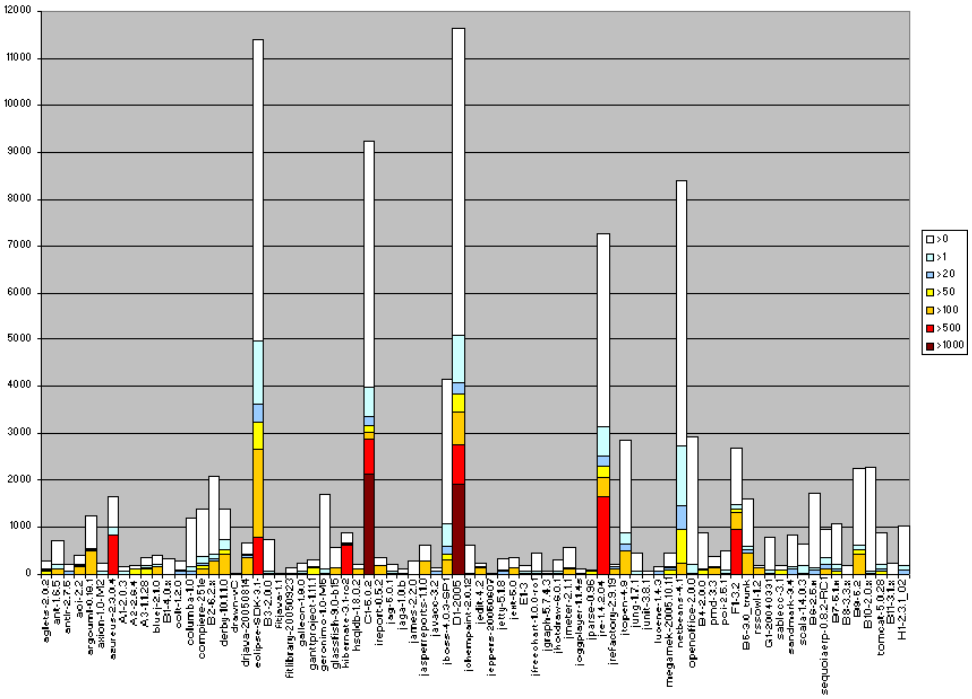
O artigo seminal (BRUCE et al., 1995) explica o conceito de métodos binários, que são métodos nos quais tanto o objeto receptor quanto o passado como parâmetro são utilizados na determinação da lógica computacional que será utilizada para o processamento da invocação da chamada. Em outras palavras, métodos binários são métodos utilizando o mecanismo de despacho múltiplo de mensagens, conforme visto no capítulo 2. Também são abrangidas pelo artigo técnicas, como por exemplo a técnica proposta por Ingalls (INGALLS, 1986), que possibilitam a simulação desses métodos em linguagens com despacho simples.

Mais á frente (seção 5.2), será mostrado que com a utilização do conceito de métodos binários e despacho múltiplo é possível a manutenção dos subcontratos de simetria e de transitividade, mesmo quando novas informações são incorporadas às subclasses.

3.4.3 Referências Cíclicas

Apesar das boas práticas de programação dizerem que a presença de ciclos em referencias de objetos deve ser evitada (MARTIN, 2000), estudos de caso realizados em (MELTON; TEMPERO, 2007) comprovam que este tipo de estrutura é comumente encontrada em softwares escritos em Java em produção atualmente. Esse fato também é contado no estudo de (RAYSIDE et al., 2009), que demonstrou alguns casos em que a lógica de verificação da igualdade de um objeto apresentava erro de estouro de pilha, na presença de referências cíclicas. Na figura 3 encontra-se um gráfico sumarizando os resultados obtidos nesse estudo.

Figura 3 – Quantidade de Ciclos vs Tamanho do Ciclo



Fonte: (MELTON; TEMPERO, 2007, fig. 5)

No gráfico, pode-se verificar que basicamente todos os softwares analisados contêm, em algum momento, referências cíclicas. Mais ainda, algumas dessas referências têm uma grande distância entre si (em alguns casos a distância é de mais de mil referências).

A grande presença desse tipo de estrutura faz com que a escrita dos métodos que definem a lógica de verificação de igualdade entre objetos tenha que apresentar a robustez

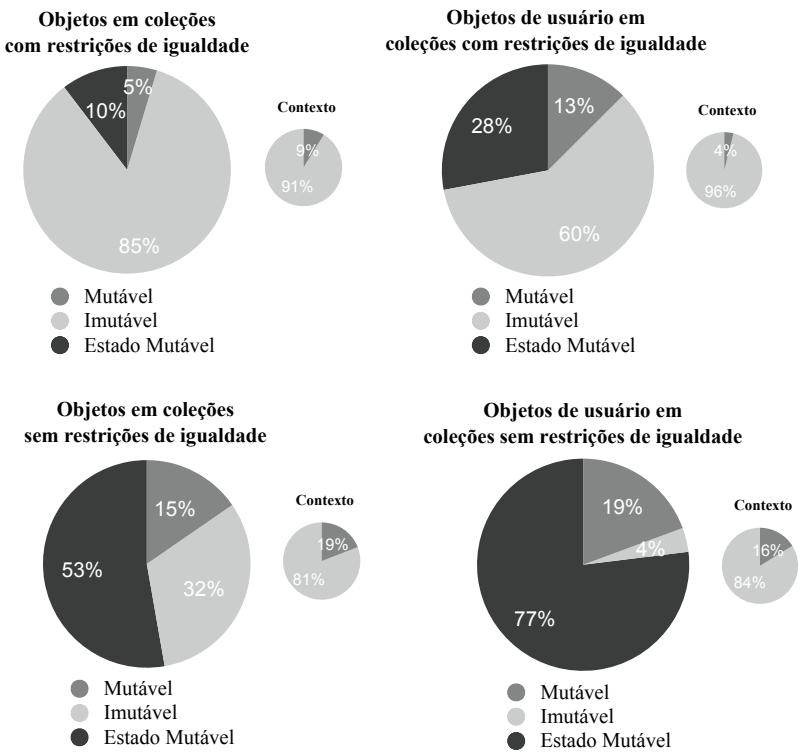
necessária para tratar esse tipo de situação, sem que ocorram erros de estouro de pilha.

3.4.4 Uso de objetos mutáveis em coleções com restrições de igualdade

Conforme visto anteriormente, foi proposta em (NELSON; PEARCE; NOBLE, 2010) uma classificação de coleções baseada na restrições de seus elementos com relação a lógica de verificação da igualdade. Além dessa classificação, foram realizados estudos de caso, tendo como foco o perfil dos elementos inseridos nessas coleções.

Nesses estudos, que tiveram como base de amostragem cerca de trinta projetos de código-fonte aberto escritos em Java, os objetos foram divididos em três grandes categorias; objetos imutáveis (cujo estado interno não pode ser alterado após a sua construção), objetos mutáveis (cujo estado interno pode ser alterado a todo momento) e objetos parcialmente mutáveis (nos quais somente atributos não utilizados pela lógica de comparação de igualdade são alterados). A utilização dessas categorias de objetos foram sumarizadas em função do tipo de coleções que foram incluídas no estudo : coleções com e sem restrições de igualdade). A figura 4 apresenta um sumário dos resultados obtidos pelo estudo.

Figura 4 – Perfil de Uso de Coleções



Fonte: (NELSON; PEARCE; NOBLE, 2010, fig. 4, tradução do autor)

Como se pode ver nos resultados dispostos na figura 4, 40% dos objetos inclusos em coleções com restrições de igualdade mudam seu estado de igualdade após sua inserção. Já em coleções sem restrições, esse número cresce para 77%. Porém o estudo revela que, como ambos os tipos de coleção são intercambiáveis (ambas implementações realizam a interface *java.util.Collection*, apesar de não pertencerem a um mesmo tipo), a alteração indiscriminada da implementação de coleção utilizada poderia acarretar em falhas.

4 Soluções disponíveis

A objetivo deste capítulo é comparar algumas soluções para o problema da definição da lógica de comparação entre objetos, seguindo um critério bem-definido. Serão descritas cinco soluções, disponibilizadas tanto no meio acadêmico (soluções teóricas encontradas em artigos) quanto soluções do meio não-acadêmico (soluções que podem ser obtidas por um programador para a utilização em seus programas). Além das suas respectivas descrições, é realizado, nesta seção, um comparativo entre as soluções acadêmicas.

A estrutura deste capítulo está definida em três seções. A primeira descreve os critérios de comparação que serão utilizados na comparação das soluções acadêmicas, já na segunda seção são expostas as características e peculiaridades de cada solução, tendo como foco os critérios definidos na primeira seção. Na última seção são feitas considerações finais com relação às soluções apresentadas e apresentado um sumário contendo os resultados obtidos.

4.1 Critérios de comparação das soluções acadêmicas

Com o intuito de facilitar a identificação das diferenças entre as soluções para o problema da definição da lógica de verificação da igualdade apresentadas, foram definidos alguns critérios, sob os quais as soluções devem ser analisadas.

O primeiro ponto a ser analisado corresponde às características principais da solução. São considerados sob esse aspecto itens como as características de linguagem utilizadas pela solução, o ferramental utilizado, a tratativa dada ao problema da definição da ordem natural dos objetos e qualquer outra característica única à solução sendo analisada.

O segundo ponto a ser observado pela análise corresponde ao cumprimento do contrato que rege as operações de verificação da igualdade. A análise realizada aborda tanto a questão da manutenção da simetria e transitividade das operações de verificação da igualdade quando existe uma relação de herança entre seus operandos, quanto o contrato informal que rege a interdependência entre as operações *equals()* e *hashCode()*. Também são analisadas nesta seção os mecanismos definidos por cada uma das soluções para a definição da lógica de verificação da igualdade.

Outro ponto analisado corresponde à robustez da solução. Sob essa ótica, são analisados aspectos como a robustez sob a presença de referências cíclicas, a utilização de coleções com restrições de igualdade e a extensibilidade da solução. Também são citadas nessa seção qualquer aspecto de robustez particular à solução analisada.

O último aspecto de análise diz respeito aos resultados obtidos através de experi-

mentos, nos quais alguns programas de código aberto são alterados para a utilização das soluções sendo analisados. Em função dos experimentos não serem iguais[1], a comparação entre os resultados obtidos pelas soluções apresentadas fica prejudicada. Logo, serão somente sumarizados os resultados obtidos pelos experimentos, bem como as conclusões de seus autores.

4.2 Soluções analisadas

Definidos os critérios a serem utilizados no comparativo entre soluções, serão analisadas com detalhes três soluções. Conforme citado, serão comparadas soluções disponíveis no meio acadêmico,representadas por Vaziri et al.(VAZIRI et al., 2007), Rayside et al.(RAYSIDE et al., 2009) e Grech, Rathke e Fischer (GRECH; RATHKE; FISCHER, 2010), dispostas pela ordem cronológica de suas respectivas publicações. Já as soluções de mercado, representadas pelos geradores de código disponíveis em IDEs (*Integrated Development Environment*) de mercado, serão descritas brevemente, com o intuito de fornecer ao leitor um esboço do cenário referente a métodos mais robustos para a definição da lógica de verificação da igualdade.

4.2.1 Geração de Código

Uma forma de resolver o problema da implementação da lógica de verificação de igualdade é através da geração automática dos métodos *equals()* e *hashCode()*, através do uso de assistentes (*wizards*), normalmente disponíveis nas IDEs. Serão aqui analisados os assistentes disponíveis nas IDEs NetBeans[2] (Versão 8.0.2) e Eclipse [3] (Kepler).

4.2.1.1 NetBeans 8.0.2

O primeiro gerador de código analisado é o gerador embutido na versão 8.0.2 do NetBeans. O assistente presente na IDE permite a implementação automática dos métodos *equals()* e *hashCode()*, permitindo a escolha de quais atributos serão utilizados na implementação de cada uma delas. Porém, não existe nenhuma restrição obrigando a seleção desses atributos para a geração das implementações dos métodos *hashCode()* e *equals()*, o que pode provocar violações no contrato informal que rege a interdependência entre as operações, caso atributos diferentes sejam selecionados.

[1] A comunalidade entre eles é a alteração da biblioteca JFreeChart para a utilização da solução. Porém somente os experimentos descritos em (VAZIRI et al., 2007) e em (RAYSIDE et al., 2009) têm uma medida em comum – O número de violações do contrato que rege a verificação da igualdade antes e depois da adoção da solução.

[2] NetBeans. Disponível em: ⟨https://netbeans.org/⟩.

[3] Eclipse. Disponível em: ⟨https://eclipse.org/⟩.

Na Listagem 2, encontra-se um trecho de código, mostrando as implementações geradas pelo assistente para a classe *PontoColorida*, conforme definida no exemplo disposto na seção 3.2.2.

Listagem 2 – Geração por assistentes – Netbeans 8.0.2

```
public class PontoColorido extends Ponto {

  private Cor cor;

  @Override
  public int hashCode() {
    int hash = 7;
    hash = 23 * hash + Objects.hashCode(this.cor);
    return hash;
  }

  @Override
  public boolean equals(Object obj) {
    if (obj == null) {
      return false;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    final PontoColorido other = (PontoColorido) obj;
    if (!Objects.equals(this.cor, other.cor)) {
      return false;
    }
    return true;
  }
}
```

Analisando o código gerado pelo assistente, pode-se notar que as implementações geradas ignoram as suas contrapartidas, definidas na classe *Cor*. Outra peculiaridade das implementações geradas por esse assistente é a utilização da classe de implementação na lógica de verificação da igualdade, o que indica a adoção de uma estratégia de comparação com compatibilidade de tipos, conforme classificação definida em (RUPAKHETI; HOU, 2008; RUPAKHETI, 2010). Também pode-se notar que não é implementado nenhum tipo de tratamento protegendo a lógica de verificação da igualdade de referências cíclicas.

4.2.1.2 Eclipse Kepler

O segundo gerador de código analisado é o gerador utilizado pelo assistente de geração de código, embutido na versão Kepler do Eclipse. Da mesma maneira que o gerador incluso no assistente do NetBeans, é possível a seleção de quais atributos serão utilizados pela lógica de verificação da igualdade na comparação entre objetos. Diferentemente do assistente do NetBeans, não é possível a seleção de um conjunto de atributos diferentes para a geração de implementações do método *equals()* e do método *hashCode()*. Essa res-

trição acarreta um aumento da robustez, uma vez que o programador fica impossibilitado de gerar implementações de *equals()* e *hashCode()* que violem o contrato que rege a interdependência entre elas. Outra diferença importante é a possibilidade da escolha do uso ou não da classe de implementação na lógica de verificação da igualdade, possibilitando o uso das estratégias de comparação com compatibilidade de tipo e com incompatibilidade de tipos, conforme a classificado em (RUPAKHETI; HOU, 2008; RUPAKHETI, 2010).

Na Listagem 3, encontra-se um trecho de código referente à implementação gerada pelo assistente incluso na versão Kepler do Eclipse. A semântica de comparação utilizado é a mesma utilizada no exemplo de geração do NetBeans, com o adendo da opção de não se utilizar a classe de implementação na verificação da igualdade.

Listagem 3 – Geração por assistentes – Eclipse Kepler

```
public class PontoColorido extends Ponto {

  private Cor cor;

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = super.hashCode();
    result = prime * result + ((cor == null) ? 0 : cor.hashCode());
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (!super.equals(obj))
      return false;
    if (!(obj instanceof PontoColorido))
      return false;
    PontoColorido other = (PontoColorido) obj;
    if (cor == null) {
      if (other.cor != null)
        return false;
    } else if (!cor.equals(other.cor))
      return false;
    return true;
  }
}
```

Observando o código na Listagem 3, pode-se verificar que, ao contrário do fragmento de código disposto na Listagem 2, a implementação gerada pelos assistentes do Eclipse faz referência a suas contrapartes na classe *Ponto*. Pode-se notar também que o código gerado pelo Eclipse não consegue manter a característica de simetria[4], quando é utilizada

[4] A violação da simetria quando usado esse idioma de implementação no método *equals()* é descrito com detalhes em (BLOCH, 2008, p. 37).

uma estratégia de comparação com incompatibilidade de tipos e são adicionados novos atributos à lógica de verificação da igualdade nas subclasses. Assim como o código gerado pelo assistente de geração de código do Netbeans, não estão presentes tratamentos contra a recursão infinita presente na existência de referências cíclicas.

4.2.2 Solução proposta por Vaziri et al.

A primeira solução apresentada no meio acadêmico é a proposta para a definição da lógica de verificação da igualdade apresentada em (VAZIRI et al., 2007). Nessa solução é incorporado o construto Tipo de Relação (*relation type*), responsável exclusivamente pelo encapsulamento da lógica de verificação de igualdade e das informações usadas por ela. De maneira semelhante a classes de objetos, um tipo de relação pode ser estendido, no intuito de se criar uma relação entre tipos e subtipos de relação. Segue uma definição informal para tipos de relação :

> "Um tipo de relação assemelha-se a uma classe em Java, exceto pela impossibilidade de o programador sobrescrever os métodos *equals()* e *hashCode()*. Em vez disso, o programador deve designar um (possivelmente vazio) subconjunto de atributos como chaves, utilizando a palavra-chave *key*. Atributos-chave são implicitamente imutáveis e privados. Uma instância de um tipo de relação é uma tupla, e sua identidade é determinada por seu tipo e as identidades das instâncias referenciadas por seus atributos-chave."(VAZIRI et al., 2007, tradução do autor)

De acordo com (VAZIRI et al., 2007), os tipos de relação provêm uma visão semelhante a um banco de dados na memória da plataforma Java, no qual um estado mutável é identificado por um conjunto de informações (representado pelos atributos-chave do tipo de relação). Na prática, os tipos de relação se comportam de maneira semelhante ao padrão de projeto *Flyweight* (GAMMA et al., 1994, p.218), onde os atributos-chave são utilizados na busca (encapsulado em uma operação de busca gerada automaticamente) da instância correspondente. Caso já exista uma instância de tipo de relação relacionada aos atributos fornecidos, ela é retornada, caso contrário uma nova é criada. Essa característica faz com que seja possível se efetuar verificações através de identidade, uma vez que é garantido pelo ambiente de execução que duas referência a tipos de relação com os mesmos atributos-chave são referências à mesma instância.

4.2.2.1 Características Principais

A solução proposta em (VAZIRI et al., 2007) modifica o metamodelo de programação da linguagem Java, introduzindo um novo construto: o tipo de relação. Para isso, são adicionadas algumas extensões à sintaxe da linguagem Java. Essas inclusões são compostas dos seguintes itens:

a) A palavra-chave *relation*, usada para indicar que uma classe (ou interface) representa um tipo de relação;

b) A palavra-chave *key*, utilizada na declaração de um atributo para indicar que ele é um atributo-chave;

c) A inclusão implícita da operação *id()*, que encapsula a lógica de busca de tipos de relação através dos seus atributos-chave.

Apesar de no nível conceitual serem introduzidas novas palavras-chaves à linguagem, na implementação do protótipo foram utilizados os conceitos de programação orientada a atributos, efetivamente substituindo as novas palavras-chaves por anotações. Essa alteração permitiu a utilização do *framework* de refatoração de código JDT[5] como base para a implementação do protótipo de compilador da solução.

Conforme citado, a solução proposta em (VAZIRI et al., 2007) utiliza-se da identidade das referências para a verificação da igualdade entre elas, ao invés de executar as comparações dos atributos contidos em seus atributos-chave. Essa estratégia, apesar ser eficaz[6] na verificação da igualdade, impossibilita o cálculo da ordem natural entre objetos. De fato, não faz parte do escopo da solução proposta em (VAZIRI et al., 2007) a definição da ordem natural de tipos de relação.

4.2.2.2 Cumprimento do Contrato

Conforme citado, a solução proposta em (VAZIRI et al., 2007) utiliza o conceito da identidade para a comparação de instâncias de tipos de relação. Conforme visto na seção 3.1.2, a comparação por identidade se dá através do endereço da memória alocada para o armazenamento de um objeto. Essa comparação pode ser reduzida a uma comparação entre dois números naturais. Pode-se assumir, conforme previamente estabelecido, que os aspectos de simetria, reflexividade e transitividade são considerados axiomas na verificação da igualdade entre números inteiros, logo pode-se, por dedução, inferir que as comparações entre instâncias de tipos de relação também tenham essas características.

Quando usada a solução apresentada, operandos de tipos diferentes sempre são considerados diferentes, mesmo quando existe uma relação hierárquica entre eles, em que não são inclusos novos atributos-chave. Essa peculiaridade é observada por Rupakheti.

> "Vaziri et al.(VAZIRI et al., 2007) estende Java com tipos de relação, com os quais a igualdade pode ser especificada em termos de propriedades de objetos e a implementação de *equals()* pode ser gerada automaticamente. Porém, tipos de relação não levam em conta o comportamento dos objetos e não dão suporte a comparações compatíveis com tipo e nem a comparações híbridas (a implementação gerada usa a

[5] Eclipse Java Development Tools (JDT). Disponível em: ⟨https://eclipse.org/jdt/⟩.

[6] Quando é garantido pelo ambiente de execução que na construção de instâncias os mesmos atributos-chave retornam sempre a mesma instância.

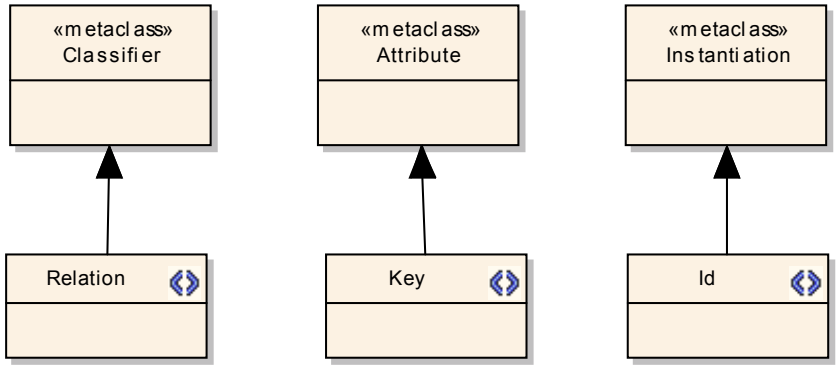
> expressão *getClass*()). Logo, a solução soluciona somente um subconjunto das questões relativas às implementações da operação *equals()*." (RUPAKHETI, 2010, p.130, tradução do autor)

Na implementação da operação *hashCode()*, o próprio endereço de memória é utilizado como código *hash* representando o estado interno da instância de um tipo de relação. Essa implementação é compatível com o contrato que rege a interdependência entre as operação *equals()* e *hashCode()*, uma vez que, garantido pelo ambiente de execução que referências a tipos de relação com os mesmos atributos-chave referenciam a mesma instância, os endereços de memória referenciados por elas serão iguais. Logo elas terão o mesmo código *hash*.

4.2.2.3 Formas de Uso

Apesar de a solução apresentada em (VAZIRI et al., 2007) ser uma extensão da linguagem Java, definindo um novo conjunto de palavras-chave, a implementação do protótipo utilizou-se dos recursos de anotação, definidos na versão 1.5 da linguagem Java, para a declaração de seus construtos. Na figura 5, encontra-se um diagrama de perfil, representando as anotações definidas pelo protótipo.

Figura 5 – Diagrama de Perfil (VAZIRI et al., 2007)



Conforme o diagrama, são definidas pelo protótipo três anotações; *Relation*, *Key* e *Id*. Pode-se traçar um paralelo entre as anotações *Relation* e *Key* e as palavras-chaves *relation* e *key*, que são usadas, respectivamente, para a declaração de tipos de relações e seus atributos-chave. Já a anotação *Id* serve para anotar o construtor da classe representando um tipo de relação, de modo a inicializar os atributos-chave com seus respectivos valores. Na Listagem 4 encontra-se um fragmento de código representando a definição de um tipo de relação *Livro*, contendo os atributos *isbn* e *nome*. O atributo *isbn* deve ser considerado atributo-chave de instâncias de livro.

Listagem 4 – Declaração – Palavras-chave (VAZIRI et al., 2007)

```
relation Livro {

  key ISBN isbn;
  String nome;
}
```

Já no fragmento embutido na Listagem 5, são utilizadas as anotações definidas pelo protótipo para a definição do mesmo tipo de relação definido na listagem 4.

Listagem 5 – Declaração – Anotações (VAZIRI et al., 2007)

```
@Relation public class Livro {

  @Key private final ISBN isbn;
  String nome;

  @Id private Livro(ISBN isbn) {
    this.isbn = isbn;
  }
}
```

Pode-se notar que o construtor anotado com a anotação *Id* tem o modificador de acesso privado. Isto se dá pelo fato desse construtor não ser utilizado diretamente pelo programador e sim pelo método *id()*, gerado automaticamente pela solução. Conforme citado, este método deve ser utilizado pelo programador para a obtenção de novas instâncias de um tipo de relação. No trecho de código disposto na Listagem 6, encontra-se um exemplo indicando a obtenção de uma instância de *Livro*, cujo ISBN é *0-321-35668-3*.

Listagem 6 – Obtenção de instância de tipo de relação (VAZIRI et al., 2007)

```
Livro livro = Livro.id(new ISBN("0-321-35668-3"));
```

Na solução proposta em (VAZIRI et al., 2007) são definidas algumas restrições no que tange às manipulações que podem ser realizadas pelo programador em instâncias de tipo de relação, bem como restrições no uso destas instâncias em expressões da linguagem. Um verificador das restrições foi incorporado à implementação do protótipo, avisando o programador em tempo de compilação de possíveis violações dessas restrições. Essas restrições são :

a) É proibida a conversão entre referências de um tipo de relação para referências do tipo *Object*;

b) Atributos-chave devem ser privados e imutáveis;

c) É proibida a declaração dos métodos *equals()* e *hashCode()* em classes representando tipos de relação;

d) É vetada a utilização de expressões do tipo == ou ! =, em que um dos operando é uma referência a um tipo de relação e o outro não;

e) São proibidas invocações da operação *equals()* contra instâncias de um tipo de relação.

4.2.2.4 Robustez

Conforme relatado, a solução apresentada utiliza-se de comparações da identidade entre instâncias de tipos de relação. Esse tipo de comparação tem alto grau de robustez na presença de ciclos de referência, uma vez que a comparação é feita usando-se apenas o endereço da memória nas quais os operandos encontram-se alocados, ao invés do caminhamento e da comparação das instâncias inclusas no grafo de referências dos operandos.

A utilização de tipos de relação também resolve de maneira eficaz o problema da alteração do estado de elementos inclusos em coleções com restrições de igualdade. Segundo categorização definida por (NELSON; PEARCE; NOBLE, 2010), as instâncias de tipos de relação pertencem ao conjunto de objetos com estado mutável, em que se é possível somente a alteração de atributos não utilizados pela lógica de verificação da igualdade. Conforme visto na seção 3.3, não existem problemas quanto à utilização dessa categoria de objetos em coleções com restrição de igualdade.

No que tange à extensibilidade da solução, não é citado em (VAZIRI et al., 2007) nenhum mecanismo para a definição de lógica de verificação da igualdade de classes cujo código-fonte não estejam disponíveis. A adoção de um novo construto para a definição dos tipos relação dificulta a redefinição da lógica de verificação de igualdade em classes de terceiros, uma vez que, conceitualmente, tipos de relação e classes são elementos distintos.

Outra questão referente à extensibilidade da solução reside no uso de objetos cuja lógica de verificação da igualdade foi definida utilizando-se dos recursos da solução em *frameworks* de mercado (como por exemplo o Hibernate[7] e o Spring[8]), que se utilizam de mecanismos como reflexão e a conversão para o tipo *Object*, proibidos pela solução apresentada em (VAZIRI et al., 2007).

Por último, existe a questão relativa ao processo de compilação definido pela solução, no qual a classe anotada com a anotação *@Relation* tem seu código-fonte modificado de acordo com os seguintes passos :

1. Geração de uma classe-chave aninhada, contendo os atributos-chave declarados em um tipo de relação e a implementação apropriada das operações *equals()* e *hashCode()*;

[7] Hibernate. Disponível em: ⟨http://hibernate.org/⟩.
[8] Spring. Disponível em: ⟨http://spring.io/⟩.

2. Geração de um mapa estático[9] que contém as tuplas de relação;
3. Geração de um construtor que inicializa os atributos-chave, através de parâmetros correspondentes;
4. Geração do método *id()* que retorna uma instância de tipo de relação, caso já exista uma com os atributos-chave correspondente ou, no caso contrário, cria uma nova instância;
5. Atualização de referências aos atributos-chave (esse passo é necessário, uma vez que esses atributos foram movidos para a classe-chave, gerada no primeiro passo).

Essas alterações, realizadas pelo compilador desenvolvido para o protótipo solução, ocasionam uma lacuna entre o código-fonte criado pelo programador e código-objeto sendo executado. Essas lacunas podem acarretar dificuldades na utilização de ferramental como depuradores (*debuggers*) e ferramentas de cobertura de código, que fazem um paralelo entre o código-objeto sendo executado e o código-fonte.

4.2.2.5 Resultados

No intuito de examinar a solução apresentada, é executado em (VAZIRI et al., 2007) um experimento, cujo resultados obtidos encontram-se sumarizados na Tabela 1[10], consistindo na alteração do código-fonte[11] de algumas bibliotecas de código aberto para a utilização da solução apresentada. Para essa conversão, foi adotada a seguinte metodologia.

> "[...] transformamos cada código à mão, transformando cada atributo em privado e final, removendo mudanças de estado e mudanças locais desnecessárias. Apesar de acreditarmos que nossas transformações estão corretas, não temos nenhuma prova automatizada de que as alterações preservam a semântica do código. Rodamos alguns testes dinâmicos em cada código, incluindo: teste de unidade, quando disponíveis, testes da suíte DaCapo (BLACKBURN et al., 2006) e outros testes por nós criados e verificamos que em cada um deles as bibliotecas modificadas comportam-se de maneira idêntica à implementação original. Essa metodologia nos deu confiança que as versões modificadas estão corretas."(VAZIRI et al., 2007, tradução do autor)

Após a alteração das bibliotecas, foi notado no experimento descrito em (VAZIRI et al., 2007) que fôra possível se executar a conversão da maioria das classes candidatas a se

[9] A implementação mais ingênua desse mapa poderia acarretar em um estouro de memória, uma vez que as referências à instâncias de tipos de relação nunca seriam coletadas pelo coletor de lixo. De acordo com (VAZIRI et al., 2007), a implementação do protótipo baseou-se em referências fracas (*WeakReferences*), que permitem a coleta de lixo quando são as únicas referências a algum objeto.

[10] Disposta na página 54 desta monografia.

[11] No caso da biblioteca JFreeChart, foram convertidos somente alguns pacotes. Essa decisão foi tomada em virtude do tamanho do código-fonte dessa biblioteca, que, segundo (VAZIRI et al., 2007), tem mais de 200 implementações diferentes do método *equals()*.

tornarem tipos de relação com relativamente poucas alterações, já que a maioria destas classes representavam objetos sem estado mutável. Ainda, é citado que as classes que não foram convertidas podem ser categorizadas em dois tipos:

a) Classes mas quais o programador originalmente implementara mecanismos de *hash-consing*[12] ou *pooling*. Nesses casos, Vaziri et al. consideram que a reescrita seria muito complexa para o experimento em questão, porém relataram que a adoção dos tipos de relação aliviariam a necessidade desses mecanismo, uma vez que um mecanismo de *hash-consing* já é implementado pela solução;

b) Classes nas quais a lógica de verificação da igualdade depende de atributos que devem ser alterados depois do retorno de seu construtor. Segundo o experimento, muitas dessas classes têm uma fase de inicialização, em que, após a construção, elas têm seus atributos-chaves iniciados. Em (VAZIRI et al., 2007) é dado um exemplo do processo de *parsing* de um documento XML). Nesse caso, é proposto, como trabalho futuro, a inclusão de um mecanismo que permita o suporte a esse tipo de idioma de programação.

4.2.3 Solução proposta por Rayside et al.

A solução proposta por Rayside et al. (RAYSIDE et al., 2009), tem como princípio a utilização de funções de abstração (LISKOV; GUTTAG, 1986) para a definição das informações que serão utilizadas na lógica de verificação da igualdade.

Segundo Liskov e Guttag (1986), funções de abstração são funções que definem a relação entre abstrações e suas representações internas :

> "Qualquer implementação de uma abstração deve definir como os objetos pertencentes a um tipo são representados. Ao escolher uma representação, o implementador tem em mente uma relação entre a representação e a abstração
>
> [...]
>
> Essa relação pode ser definida por uma função, chamada de função de abstração, que mapeia representações para sua respectiva abstração." (LISKOV; GUTTAG, 1986, p. 70, tradução do autor)

A solução proposta por Rayside et al. utiliza-se do resultado obtido através da aplicação de funções de abstração para efetuar a verificação da igualdade de objetos, de maneira desacoplada às suas representações internas.

[12] *Hash-consing* é uma técnica de programação usada para o compartilhamento de valores imutáveis idênticos, mantendo uma única cópia de objetos semânticamente equivalentes (BRAIBANT; JOURDAN; MONNIAUX, 2013). Um exemplo de *Hash-consing* é o padrão de projetos *Flyweight* (GAMMA et al., 1994, p.218).

4.2.3.1 Características Principais

Assim como a solução proposta por Vaziri et al. (VAZIRI et al., 2007), a solução apresentada por Rayside et al. incorpora os conceitos de programação baseada em atributos, através da utilização das anotações da linguagem Java. Porém, ao contrário da proposta de Vaziri et al., ela utiliza mecanismos disponíveis[13] no ambiente Java padrão para a geração de código, característica esta que facilita a integração com a suíte ferramental disponível no mercado. Assim como a solução proposta por Vaziri et al., a solução apresentada por Rayside et al., não disponibiliza nenhum mecanismo para a definição da lógica de comparação da ordem natural entre dois objetos.

4.2.3.2 Cumprimento do Contrato

A solução apresentada por Rayside et al. consegue resolver todos os sub-contratos associados ao contrato que rege a verificação da igualdade entre objetos quando é utilizada a estratégia de comparação baseada em classes. Porém existem restrições quanto ao uso da estratégia de comparação baseada em hierarquia, como observado por (RUPAKHETI, 2010, p. 118), que cita a falta de clareza sobre como sua estratégia de comparação híbrida pode ser implementada com as anotações disponibilizadas pela solução.

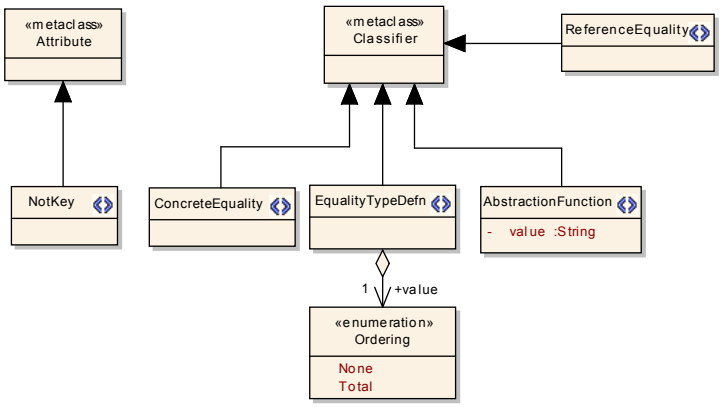
Quando é utilizada uma estratégia de comparação baseada em hierarquia, é vetado ao programador a inclusão de novas informações definidas em subclasses na lógica de verificação da igualdade. Essa restrição deve-se à linguagem de programação utilizada (no caso Java) ser baseada em despacho simples, mecanismo que, conforme citado anteriormente (seção 3.4.2), não oferece suporte à simetria de operações em uma hierarquia de classes.

4.2.3.3 Formas de Uso

A proposta é baseada em um conjunto de anotações, que deve ser utilizado pelo programador para a definição da lógica de verificação da igualdade das classes sendo programadas. Na Figura 6 encontra-se um diagrama, baseado no conceito de perfil (disponível na versão 2.0 da UML), representando as anotações disponíveis e seus atributos.

[13] Fora citado no artigo o mecanismo de reflexão (*reflection*), disponível na linguagem Java

Figura 6 – Diagrama de Perfil (RAYSIDE et al., 2009)



Segundo (RAYSIDE et al., 2009), a maneira mais comum de utilização é anotando uma classe com a anotação *@ConcreteEquality*. Quando essa anotação é utilizada, as informações associadas a todos os atributos dos objetos, bem como a classe que os implementa, são utilizadas pelo mecanismo de verificação da igualdade. Atributos que não devam ser levados em conta pela lógica de verificação de igualdade devem ser anotados com a anotação *@NotKey*. Essa solução representa a utilização da estratégia de comparação baseada em classe, previamente definida na seção 3.1.3. Um exemplo de utilização é fornecido na Listagem 7, na qual é definida uma classe chamado *PontoColorido*, onde somente as informações relativas às coordenadas devem ser levadas em conta na verificação da igualdade :

Listagem 7 – Comparação baseada em classe (RAYSIDE et al., 2009)

```
@ConcreteEquality
public class PontoColorido {

  private Coordenada coordenada;

  @NotKey
  private Cor cor;
}
```

Caso se deseje utilizar a estratégia de comparação baseada em hierarquia, o programador deverá anotar a classe base da hierarquia com as anotações *@EqualityTypeDefn* e *@AbstractionFunction*. A anotação *@AbstractionFunction* é responsável pela indicação da operação responsável pela obtenção do conjunto de informações a ser utilizado para a verificação da igualdade; já a anotação *@EqualityTypeDefn* indica se a ordem destas informações é importante ou não para a verificação da igualdade.

A Listagem 8 apresenta um exemplo da utilização das anotações *@EqualityTypeDefn* e *@AbstractionFunction*, em que são definidas três classes, chamadas de *Set*, *ConcreteSet*

e *AnotherConcreteSet*. A classe *Collection* é a superclasse das classes *ConcreteCollection* e *AnotherConcreteCollection* e define uma operação chamada *iterator()*, que retorna os objetos armazenados.

Listagem 8 – Comparação baseada em hierarquia (RAYSIDE et al., 2009)

```
@EqualityTypeDefn(Ordering.NONE)
@AbstractionFunction("iterator")
public abstract class Set {
  public abstract Iterator iterator();
}

public class ConcreteSet extends Set {
  public Iterator iterator() {
    //...
  }
}

public class AnotherConcreteSet extends Set {
  public Iterator iterator() {
    //...
  }
}
```

No exemplo, para que duas instâncias sejam consideradas iguais, elas devem ter os mesmos objetos armazenados, não importando a ordem na qual os elementos foram armazenados. Caso a ordem de armazenamento dos objetos faça diferença para a lógica de verificação de igualdade, é necessária a alteração da declaração da anotação *@EqualityTypeDefn(Ordering.None)* para *@EqualityTypeDefn(Ordering.Total)*.

Por fim, tem-se a terceira forma de utilização, que considera iguais somente objetos com a mesma identidade. Quando essa estratégia é utilizado, duas referências são consideradas iguais somente se ambas referenciarem a mesma instância[14]. Para a utilização dessa estratégia de comparação, o programador deverá anotar a classe com a anotação *@ReferenceEquality*, conforme Listagem 9.

Listagem 9 – Comparação baseada em referência (RAYSIDE et al., 2009)

```
@ReferenceEquality
public class ReferenceEqualityCalendar extends Calendar {
  //Os métodos equals() e hashCode() em Calendar serão ignorados
}
```

4.2.3.4 Robustez

Conforme mencionado, caso o programador utilize o mecanismo de definição de função da abstração baseado nas anotações *@EqualityTypeDefn* e *@AbstractionFunction*, é ne-

[14] Esse método de verificação é o mecanismo padrão da linguagem Java, em que $\forall x \forall y((x = y) \leftrightarrow (x == y))$

cessária a implementação da função de abstração da maneira tradicional. Porém segundo (RAYSIDE et al., 2009), normalmente o programador não erra na definição do que deseja comparar, mas comete erros sutis no contrato de comparação e proteção contra referências cíclicas, erros esses mitigados pela solução apresentada.

A solução apresentada por Rayside et al. reconhece a necessidade de um tratamento robusto do problema na presença de referências cíclicas em objetos utilizados como operandos nas operações de verificação da igualdade. Na implementação gerada pela solução para o método *equals()*, só são processadas instâncias cuja identidade não pertença a nenhuma outra instância já processada.

Já na implementação do método *hashCode()*, é retornada uma constante caso seja encontrado algum ciclo na estrutura do operando. Essa técnica, apesar de produzir códigos *hash* válidos, tem um impacto negativo no desempenho do programa.

> "É bom porque garante que objetos iguais tenham o mesmo código *hash* e é atroz porque garante que todos objetos tem o mesmo código *hash*. Logo, todo objeto é inserido no mesmo *bucket*, transformando efetivamente as tabelas *hash* em listas ligadas. Programas que deveriam rodar em tempo linear passam a rodar em tempo quadrático. Para tabelas *hash* grandes, isso é a diferença entre funcionar e não funcionar."(BLOCH, 2008, p.47, tradução do autor)

Para a solução do problema da inclusão de elementos mutáveis em coleções com restrição de igualdade. Rayside et al. utilizam a comparação por identidade (*@ReferenceEquality*), de maneira análoga à convenção adotada por Liskov e Guttag :

> "Deve ser impossível diferenciar dois objetos iguais. Então dois objetos mutáveis só podem ser iguais se e somente se eles forem exatamente o mesmo objeto; de outra forma poderia ser possível sua distinção através da invocação de uma operação que altere seu estado em somente um deles." (LISKOV; GUTTAG, 1986, p. 94, tradução do autor)

Quando é utilizada essa estratégia de comparação, não ocorre problema da mudança das informações importantes para a verificação da igualdade de elementos inclusos em coleções com restrições de igualdade, uma vez que a identidade de um objeto não pode ser alterada. Porém quando utilizada essa estratégia, só são considerados iguais duas referências para uma mesma instância, o que pode não ser compatível com a real semântica de igualdade do tipo ao qual esses objetos pertencem.

Não é feita nenhuma menção a mecanismos de extensibilidade na solução proposta. A falta desse mecanismo pode impossibilitar a utilização da solução em alguns casos específicos, como por exemplo:

> "A característica (*feature*) (viii) [Geração de *equals()* e *hashCode()* para tipos que não acessíveis pelo *weaver*, conforme especificado pelo programador] foi adicionada depois de percebemos que uma quantidade

> de classes em bibliotecas externas usadas no código do usuário não tinham uma implementação correta de *equals()*." (GRECH; RATHKE; FISCHER, 2010)[p. 7, tradução do autor]

Essa afirmação é corroborada pelo estudo de caso realizado em (RUPAKHETI, 2010, p. 118), que ao verificar o código-fonte do Apache Lucene[15] , uma biblioteca com código aberto amplamente utilizada no mercado, se deparou com um caso em que não foi possível a definição da lógica da verificação da igualdade com as anotações definidas pela solução.

4.2.3.5 Resultados

Além da solução aqui analisada, é descrito em (RAYSIDE et al., 2009) um experimento que realiza a substituição das implementações dos métodos associados às verificações de igualdade em três grandes projetos (*framework* de coleções do JDK 1.4 [16] , Apache Commons Collections 3.2 [17] e JFreeChart 1.0.0[18]) e alguns aspectos, como a expressividade, a corretude, a facilidade de uso e o desempenho.

No que tange à expressividade, foi possível a substituição da implementação dos métodos *equals()* e *hashCode()* pelas anotações na grande maioria dos casos (cerca de 97% dos casos). Porém, conforme citado anteriormente, algumas ocorrências não puderam ser substituídas em função da falta de pontos de extensão na solução. A Tabela 3 apresenta um sumário da quantidade de métodos relativos à definição da lógica de comparação implementados, antes e depois da adoção da solução.

Tabela 3 – Expressividade (RAYSIDE et al., 2009)

		Antes	Depois
JDK	*equals()*	24	2
	hashCode()	24	3
Apache	*equals()*	56	6
	hashCode()	57	9
JFreeChart	*equals()*	263	0
	hashCode()	114	0

Fonte: (RAYSIDE et al., 2009, tab. 3)

Para a verificação da corretude, foi gerado, de maneira automática através da ferramenta Randoop[19] , um conjunto de testes unitários, executados antes e após a substituição

[15] Apache Lucene. Disponível em: ⟨http://lucene.apache.org/core/⟩.
[16] Java Archive Downloads - Java SE 1.4. Disponível em: ⟨http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javase14-419411.html⟩.
[17] Apache Commons Collections. Disponível em: ⟨http://commons.apache.org/proper/commons-collections/⟩.
[18] JFreeChart. Disponível em: ⟨http://www.jfree.org/jfreechart/⟩.
[19] Randoop - Random test generation. Disponível em: ⟨https://code.google.com/p/randoop/⟩.

dos métodos pelas anotações disponibilizadas pela solução. Foi observado que existiram menos problemas referentes a violações no contrato após a adoção da solução proposta, por exemplo, na biblioteca JFreeChart foram encontradas 32 classes nas quais o método *equals()* fôra definido sem a redefinição do método *hashCode()*. Também foi relatado um aumento da robustez do programa na presença de ciclos de referências nos objetos utilizados como operandos nas operações de igualdade. Na Tabela 4[20] encontram-se os resultados obtidos na execução da suíte de testes.

Tabela 4 – Corretude (RAYSIDE et al., 2009).

	Testes Randoop					Testes Desenvolvedor				
	Total	Original		Anotado		Total	Original		Anotado	
		Erros	Falhas	Erros	Falhas		Erros	Falhas	Erros	Falhas
JDK	20661	1	184	0	0	1366	2	5	2	7
Apache	3415	0	7	0	0	2443	0	0	0	0
JFreeChart	59	2	57	2	0	1038	1	8	0	3

Fonte: (RAYSIDE et al., 2009, tab. 2)

Para a verificação do aspecto de facilidade de uso, foi utilizado um critério de medição subjetivo, que compara o número de anotações utilizado e a quantidade de métodos removidos com a utilização da técnica. No experimento realizado foi detectada uma diminuição de cerca de 36% no número de métodos associados à definição da lógica de verificação da igualdade. As Tabelas 5 e 6 apresentam os resultados obtidos no experimento.

Tabela 5 – Facilidade de Uso (Anotações) (RAYSIDE et al., 2009).

	@ReferenceEquality	*@ConcreteEquality*	*@NotKey*	*@ReferenceEquality*	*@AbstractionFunction*	Total
JDK	23	2	0	4	5	34
Apache	37	18	0	4	3	62
JFreeChart	0	130	27	8	4	169

Fonte: (RAYSIDE et al., 2009, tab. 4)

Na Tabela 5 encontra-se a quantidade de anotações que foram adicionadas para a adoção da solução. Já na Tabela 6 encontram-se as quantidades de classes e métodos que

[20] Na tabela, erros são casos de testes que levantam exceções não esperadas, enquanto falhas são casos de testes onde foram detectadas divergências de resultados

foram removidos (ou adicionados) com o uso da solução.

Tabela 6 – Facilidade de Uso (Métodos/Classes) (RAYSIDE et al., 2009)

	Classes			Métodos		
	Base	+	-	Base	+	-
JDK	159	7	2	1202	50	72
Apache	728	11	0	5544	39	121
JFreeChart	1158	4	0	8960	30	377

Fonte: (RAYSIDE et al., 2009, tab. 5)

O último aspecto observado foi o desempenho relativo do protótipo implementado para a solução. Aqui, foram definidas três suítes de testes, a serem executadas contra o código-fonte do JFreeChart modificado para a utilização da solução :

a) A suíte de testes de unidade original do JFreeChart, inclusa no código-fonte de biblioteca;

b) Uma suíte de testes de unidade alternativa, gerada automaticamente através da ferramenta Randoop;

c) Um teste de desempenho baseado no JFreeChart, disponível na suíte de testes DaCapo(BLACKBURN et al., 2006)[21].

Segundo Rayside et al., as duas primeiras suítes de teste têm um perfil de *microbenchmark*, não representando fielmente o desempenho esperado no uso real, diferentemente da suíte de testes DaCapo, que tem um resultado mais próximo ao uso real. Nas duas primeiras suítes de teste houve um aumento médio de 21% e 32% no tempo de execução, respectivamente, já na suíte de teste DaCapo, foi detectada um aumento médio de apenas 2% no tempo de execução. Na Tabela 7, encontram-se os resultados de desempenho obtidos no experimento.

Tabela 7 – Desempenho (RAYSIDE et al., 2009)

	Randoop			Desenvolvedor			DaCapo			Médias
	Antes	Depois	Delta	Antes	Depois	Delta	Antes	Depois	Delta	
JDK	7374	7699	+4%	4410	3837	-13%	-	-	-	-4%
Apache	2231	2514	+13%	2479	5049	+104%	-	-	-	+58%
JFreeChart	669	984	+47%	20382	21217	+4%	21355	21827	+2%	+18%
Médias			+21%			+32%			+2%	+23%

Fonte: (RAYSIDE et al., 2009, tab. 6)

[21] Foi utilizado o teste de carga de trabalho grande (*large workload*).

4.2.4 Solução proposta por Grech, Rathke e Fischer

A terceira solução examinada é a solução proposta em (GRECH; RATHKE; FISCHER, 2010)[22]. A solução, chamada de JEqualityGen, é conceitualmente bem parecida à solução proposta em (RAYSIDE et al., 2009), descrita na seção anterior. A principal diferença entre elas é a utilização de código-fonte gerado, em deterimento do uso de reflexão. Esse enfoque tem uma série de vantagens:

> "Desenvolvemos um sistema similar [ao de (RAYSIDE et al., 2009)] mas que ao invés de utilizar reflexão, gera código. Isso possibilita a utilização de ferramentas de análise estática e verificação no código gerado. É muito mais fácil verificar e provar propriedades de código gerado, do que de código genérico (e logo complexo) que faz uso de reflexão.
>
> Além de facilitar trabalhos futuros na verificação da corretude, essa decisão traz um aumento de desempenho sensível : tipicamente 162 vezes mais rápido para verificação da igualdade e 31 vezes mais rápido para o cálculo do código *hash*."(GRECH; RATHKE; FISCHER, 2010, tradução do autor)

4.2.4.1 Características Principais

De maneira análoga às soluções propostas (VAZIRI et al., 2007; RAYSIDE et al., 2009), a solução proposta por Grech, Rathke e Fischer utiliza-se de anotações, utilizadas pelo programador, para a criação da lógica de programação representando a semântica da verificação da igualdade.

Tendo como base a semântica definida pelo programador, são gerados pela solução proposta por Grech, Rathke e Fischer aspectos, que têm como responsabilidade a injeção da lógica de comparação nas classes anotadas. Para a geração do código desses aspectos, são utilizadas técnicas de meta-programação, disponibilizadas pelo uso da biblioteca Meta-AspectJ[23] , uma extensão do AspectJ que incorpora recursos de meta-programação ao mecanismo de definição de aspectos.

Assim como as soluções apresentadas por Vaziri et al. e Rayside et al., não faz parte do escopo da proposta de Grech, Rathke e Fischer a disponibilização de mecanismos para a definição do cálculo da ordem natural entre dois objetos, de maneira consistente com a lógica de verificação da igualdade declarada pelo programador.

Uma característica interessante, única à solução proposta em (GRECH; RATHKE; FISCHER, 2010), é a utilização de otimizações em tempo de execução no processo de verificação da igualdade. Essa otimização é implementada da seguinte maneira:

> "As otimizações baseadas no tempo de execução são feitas através da geração de código com lógica para coleta de dados. Essa lógica simplesmente compara cada um dos campos de um objeto com os relativos ao

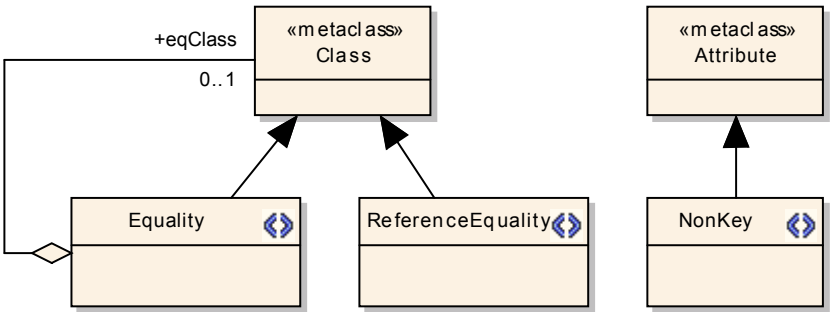
[22] O código-fonte dessa solução encontra-se disponível em http://sourceforge.net/projects/jequalitygen/
[23] Meta-AspectJ Web Page. Disponível em: ⟨http://cgi.di.uoa.gr/~smaragd/maj/⟩.

> objeto sendo comparado. Se dois campos correspondentes são diferentes, então a marcação de *falso* é incrementada, caso contrário, é incrementada a marcação de *verdadeiro*. Essas marcações então são utilizadas para ordenar os campos em função da razão entre os marcadores de verdadeiro e falso. A expressão de verificação de igualdade gerada então é mais propensa a sair por curto-circuito quando invocada." (GRECH; RATHKE; FISCHER, 2010, tradução do autor)

4.2.4.2 Formas de Uso

Da mesma maneira que a solução proposta por Rayside et al. (RAYSIDE et al., 2009), discutida na seção anterior, a solução proposta por Grech, Rathke e Fischer é baseada em anotações. Segue, na Figura 7, um diagrama de perfil representando as anotações disponibilizadas pela solução e suas respectivas relações :

Figura 7 – Diagrama de Perfil (GRECH; RATHKE; FISCHER, 2010)



Examinando o diagrama na Figura 7, pode-se notar algumas semelhanças com as anotações na solução proposta por (RAYSIDE et al., 2009). Por exemplo, ambas definem as anotações *@NotKey* e *@ReferenceEquality*, utilizadas para identificação de atributos que não irão formar a lógica de verificação da igualdade.

A outra anotação disponível na solução proposta por Grech, Rathke e Fischer é a anotação *@Equality*, que deve ser utilizada pelo desenvolvedor para indicar que a solução deverá gerar os métodos *equals()* e *hashCode()* para a classe anotada. Na Listagem 10, encontra-se um trecho de código exemplificando a utilização das anotações *@Equality* e *@NotKey* para a definição da lógica de verificação de igualdade.

Listagem 10 – Comparação baseada em classe (GRECH; RATHKE; FISCHER, 2010)

```
@Equality
public class PontoColorido {
  private Coordenada coordenada;

  @NotKey
  private Cor cor;
}
```

O código disposto na Listagem 10, define uma classe chamada *PontoColorido* que, por sua vez, define dois parâmetros : *coordenada* e *cor*, dos quais somente o atributo *coordenada* é importante para a lógica de comparação de igualdade. A semântica do exemplo é idêntica ao exemplo disposto na Listagem 7, relativo à solução apresentada em (RAYSIDE et al., 2009).

Outro forma de definição da solução proposta por Grech, Rahtke e Fischer é baseada na utilização de um subtipo especifico para a comparação de igualdade. Quando esse recurso é utilizado, todas as comparações de igualdade serão baseadas na semântica de comparação definida no subtipo. Com o intuito de exemplificar a utilização desse recurso, pode-se assumir a hierarquia de classes disposta no digrama na Figura 2. Supondo que seja necessária a comparação entre instâncias de *Ponto* e *PontoColorido*, de modo que objetos do tipo *PontoColorido* sejam comparados como *Ponto*, pode-se usar o parâmetro *eqClass*, conforme exemplo ilustrado na Listagem 11.

Listagem 11 – Utilização do parâmetro *eqClass* (GRECH; RATHKE; FISCHER, 2010)

```
@Equality
public class Ponto {

  private Coordenada coordenada;
}

@Equality(eqClass=Ponto.class)
public class PontoColorido extends Ponto {

  private Cor cor;
}
```

O último modo de utilização, conforme mencionado, baseia-se na utilização da anotação *@ReferenceEquality* para a definição da lógica de igualdade baseada na identidade, de maneira idêntica à utilização da anotação homônima[24], definida em (RAYSIDE et al., 2009). A Listagem 12 apresenta um trecho de código que exemplifica a utilização dessa anotação para a definição lógica de verificação da igualdade.

Listagem 12 – Comparação baseada em referência (GRECH; RATHKE; FISCHER, 2010)

```
@ReferenceEquality
public class ReferenceEqualityCalendar extends Calendar {
  //Os métodos equals() e hashCode() em Calendar serão ignorados
}
```

4.2.4.3 Cumprimento do Contrato

A solução apresentada em (GRECH; RATHKE; FISCHER, 2010), da mesma forma que as soluções apresentadas em (RAYSIDE et al., 2009) e (VAZIRI et al., 2007), im-

[24] Para maiores detalhes, consultar seção 4.2.3.3

plementa automaticamente os métodos *equals()* e *hashCode()* de modo que o contrato informal[25] que rege a interdependência entre eles seja cumprido.

No que tange ao problema da manutenção da simetria e da transitividade quando existe uma relação de herança entre os operandos, a solução proposta em (GRECH; RATHKE; FISCHER, 2010) utiliza-se do idioma definido em (ODERSKY, 2010, cap. 30) nas implementações da lógica de verificação da igualdade. Nesse idioma, é implementada pelo programador uma operação auxiliar, chamado *canEqual()*, que indica se o objeto receptor da mensagem é capaz de se comparar com a instância fornecida como argumento da mensagem. Por sua vez, o método *canEquals()* é utilizado nas implementações de *equals()* geradas pela solução, de modo a verificar se o correspondente simétrico da operação de verificação da igualdade pode ser verificado.

Para um melhor entendimento do mecanismo, pode-se examinar o fragmento de código disposto na Listagem 13[26], em que são definidas as classes *Ponto* e *PontoColorido*, conforme exposto no exemplo da seção 3.2.2.

Listagem 13 – Idioma de implementação de verificação da igualdade (ODERSKY, 2010)

```
public class Ponto {

  private Coordenada coordenada;

  public boolean canEqual(Object o) {
    return o instanceof Ponto;
  }

  public boolean equals(Object o) {
    if (o instanceof Ponto) {
      Ponto op = (Ponto)(o);
      if (op.canEqual(this)) {
        return coordenada.equals(op.coordeanda);
      }
    }
    return false;
  }
}

public class PontoColorido extends Ponto {

  private Cor cor;

  public boolean canEqual(Object o) {
    return o instanceof PontoColorido;
  }

  public boolean equals(Object o) {
    if (super.equals(o)) {
```

[25] Descrito com maiores detalhes na seção 3.2.5

[26] Para efeito de simplificação, foram omitidas checagens de referências nulas nas implementações da operação *equals()*

```
        if (o instanceof PontoColorido) {
          PontoColorido op = (PontoColorido)(o);
          return cor.equals(op.cor);
        }
      }
      return false;
    }
  }
```

Caso sejam definidas duas instâncias x e y, pertencentes às classes *Ponto* e *PontoColorido*, respectivamente, o resultado da expressão $x.equals(y)$ deverá ser *false*, quaisquer sejam os valores associados aos atributos das instâncias x e y. Isso ocorre em função da chamada da operação $y.canEquals(x)$, realizada pelo método *equals()*, definido na classe *Ponto*: como a instância x não é uma instância da classe *PontoColorido*, o resultado da operação é *false*.

Da mesma forma, o resultado da expressão $y.equals(x)$ também será *false*, independentemente dos valores dos atributos das instâncias x e y, já que na implementação do método *equals()*, definido na classe *PontoColorido*, é verificado se o argumento fornecido é uma instância da classe *PontoColorido*.

Tendo como base o cenário descrito, conclui-se que o idioma de implementação proposto por (ODERSKY, 2010, cap. 30) e adotado em (GRECH; RATHKE; FISCHER, 2010) consegue manter os aspectos de simetria e transitividades do contrato que rege as verificações da igualdade.

4.2.4.4 Robustez

Da mesma maneira que a solução apresentada em (RAYSIDE et al., 2009), a solução aqui proposta tem incorporados mecanismos para a detecção e tratamento de referências cíclicas. O algoritmo para o tratamento de ciclos só é incorporando quando é estaticamente possível a presença desses ciclos. Para essa checagem é utilizado um algoritmo simples :

> "Dada uma classe armazenadora particular e uma classe a ser embutida (inicialmente a mesma), o algoritmo itera por todos os atributos da classe armazenadora. Caso o atributo não seja de um tipo primitivo, é verificado se ele pode armazenar instâncias da classe sendo embutida. Esse processo é executado recursivamente, até que todas as classes da hierarquia sejam testadas."(GRECH; RATHKE; FISCHER, 2010, tradução do autor)

Caso seja detectada a possibilidade de presença de ciclos na estrutura estática de uma classe, é criado, através de técnicas de meta-programação, um aspecto cuja responsabilidade é exclusivamente o tratamento desse ciclos. Nesse aspecto é definido um ponto de corte cujo ponto de execução representa qualquer chamada feita ao método *equals()*, realizada no contexto de execução de uma chamada *equals()*, tendo o mesmo objeto como

alvo. Na Listagem 14 encontra-se um exemplo de ponto de corte, criado tendo em vista a premissa acima :

Listagem 14 – Fragmento de ponto de corte para o tratamento de ciclo (GRECH; RATHKE; FISCHER, 2010)

```
execution(boolean equals(Object)) &&
target(self) &&
cflowbelow(execution(boolean equals(Object)) &&
target(org.jfree.chart.ChartRenderingInfo))
```

Quando é detectado um ponto de junção com as características acima, é retornado o valor verdadeiro como resultado da comparação em vez do processamento propriamente dito da comparação. Esse retorno tem como base o conceito de identidade da lógica booleana, que dita que $x \wedge 1 = x$. Caso contrário, o processo é executado da maneira habitual.

Um problema identificado nessa técnica é a utilização da análise em tempo de compilação para a checagem da possibilidade da existência de ciclos. Com o advento de métodos de alteração de código em tempo de execução (como por exemplo o mecanismo de *load-time weaving*, presente no AspectJ), é possível a inserção de constructos que permitem a existência de referências cíclicas após o processo de geração de código. Nesse caso, as operações de verificação da igualdade perdem a robustez e tornam-se possíveis erros de estouro de pilha.

A solução proposta por Grech, Rathke e Fischer utiliza, no que concerne a mutabilidade dos atributos utilizado pela verificação da igualdade, o mesmo conceito da solução descrita em (VAZIRI et al., 2007). Esse conceito dita que, após a construção, é impossível a alteração dos valores importantes à verificação da igualdade. De acordo com a classificação de objetos adotada em (NELSON; PEARCE; NOBLE, 2010) [27], os objetos, cujas classes têm a lógica de verificação da igualdade definida através das anotações disponibilizadas pela solução pertencem à categoria de objetos com estado mutável (em que só é possível a alteração de atributos que não sejam utilizados pela lógica de verificação de igualdade). Conforme visto, esse tipo de mutabilidade não viola o contrato que rege instâncias que possam ser utilizadas como itens em coleções com restrição de igualdade.

O mecanismo utilizado para impedir a alteração dos campos utilizados pela lógica de verificação de igualdade é baseado no levantamento de exceções, que ocorre quando são tentadas alterações dos valores de atributos importantes para a lógica de verificação da igualdade.

> "Idealmente, o ambiente de execução Java checaria se o estado do objeto foi mudado após a primeira invocação de *equals()* ou *hashCode()*

[27] Explicada com mais detalhes na seção 3.3.

> e emitiria um aviso ou levantaria uma exceção."(GRECH; RATHKE; FISCHER, 2010, tradução do autor)

Apesar de ser eficaz, esse mecanismo tem uma série de problemas :

a) O levantamento de exceções em tempo de execução em virtude de alterações em campos pode levar a defeitos inesperados [28];

b) Conforme citado na seção 4.2.2.4, a imutabilidade de campos pode impedir a utilização de *frameworks* com alto grau de penetração no mercado;

c) Há a introdução de uma inconsistência, uma vez que tanto atributos com o modificador *final*[29] como atributos pertencentes à lógica de verificação da igualdade são imutáveis, porém são tratados de forma diferente, em tempo de compilação e de execução, respectivamente.

Uma solução mais adequada para esse problema seria a verificação, feita em tempo de compilação, de que todos os atributos que compõem a verificação da igualdade utilizem o modificador *final*. Caso um ou mais atributos não se enquadre nessa característica, será emitido um erro de compilação, seguido do cancelamento da compilação. Isso é possível de ser feito através de recursos do AspectJ, que conforme (GRECH; RATHKE; FISCHER, 2010), pode disparar avisos quando um ponto de junção é encontrado.

Outra solução é a utilização de decoradores para a reindexação das coleções no momento da alteração de elementos nelas inseridos, conforme descrito na seção 6.2.2.2.

Diferentemente das soluções apresentadas em (VAZIRI et al., 2007) e (RAYSIDE et al., 2009), a solução proposta por Grech, Rathke e Fischer reconhece o problema da definição da lógica de verificação da igualdade de classes cujo código-fonte encontra-se indisponível (ou não pode ser alterado) e provê mecanismos para a sua mitigação. Segundo os autores, esse problema foi identificado no decorrer da execução do experimento descrito no artigo. Para a solução desse problema, o protótipo desenvolvido utiliza-se do mecanismo descrito a seguir :

> "Para se usar o mecanismo de declarações intertipos do AspectJ com implementações atualizadas de *equals()* e *hashCode()*, é necessário que as classes da biblioteca estejam no parâmetro *inpath* do AspectJ, algo que não é sempre possível. Por exemplo, colocar todas as classes pertecentes ao JRE no parâmetro *inpath* travaria o *weaver*. Diferentemente, essas classes são estendidas e é adicionada a lógica de igualdade nas subclasses. Todas as chamadas ao construtor da classe original são interceptadas e uma instancia da classe estendida é inicializada. Qualquer chamada para os métodos *equals()* e *hashCode()* são recebidas pela classe estendida automaticamente gerada."(GRECH; RATHKE; FISCHER, 2010, tradução do autor)

[28] Esse problema foi vivenciado por Grech, Rathke e Fischer em seu experimento, no qual a biblioteca JFreeGraph foi alterada para a utilização da solução por eles apresentada

[29] Modificador da linguagem Java que denota a imutabilidade de um atributo

Apesar de mitigar o problema, a solução definida por Grech, Rathke e Fischer levanta uma série de novas questões :

a) É impossível o uso da técnica caso as classes sejam classes finais, uma vez que é impossível a criação de subclasses a partir de classes desse tipo;

b) A substituição da instância sendo criada é realizada através da interceptação de chamadas ao construtor. Logo, construções com o uso de reflexão ou chamadas ao construtor em código cujo *weaver* não tenha acesso não serão processadas;

c) Como é alterada a classe de implementação das instâncias cuja lógica de verificação da igualdade se deseja substituir, comparações baseadas em expressões do tipo $x.getClass() = y.getClass()$ terão seu resultado alterado.

Vale ressaltar que nesta solução não são descritos os mecanismos a serem utilizados pelo desenvolvedor para a definição da lógica de verificação da igualdade em classes cujo código-fonte não estejam disponíveis.

Resultados

Em (GRECH; RATHKE; FISCHER, 2010) são disponibilizados resultados referentes a um experimento, no qual o código-fonte da biblioteca JFreeChart é alterado para o uso da solução proposta. Os resultados desse experimento foram divididos em dois aspectos : desempenho e corretude.

Sob o aspecto do desempenho, foram feitas medições, através de um *benchmark*[30] especialmente escrito, entre a solução proposta e a solução de (RAYSIDE et al., 2009), que, segundo Grech, Rathke e Fischer, foi a solução encontrada com maior grau de semelhança. De acordo com os resultados do experimento, sumarizados na Tabela 8, a solução proposta teve um desempenho muito superior, chegando a ser 162 vezes mais rápida na verificação de igualdade (método *equals()*) e 31 vezes mais rápida no cálculo de códigos *hash* (método *hashCode()*).

Tabela 8 – Desempenho (GRECH; RATHKE; FISCHER, 2010)

	JEqualityGen	JEqualityGen (otimizações)	Rayside et al.
equals	2297 ms	1108 ms	179856 ms
hashCode	3602 ms	2760 ms	86683 ms

Fonte: (GRECH; RATHKE; FISCHER, 2010, tab. 1)

[30] De acordo com (GRECH; RATHKE; FISCHER, 2010), esse *benchmark* tem o foco na medição do desempenho dos métodos *equals()* e *hashCode()*

Segundo a análise realizada em (GRECH; RATHKE; FISCHER, 2010), foram identificados os seguintes motivos para uma diferença tão díspare no desempenho das duas soluções analisadas:

a) A complexidade e o tamanho do projeto sendo analisado (JFreeChart) faz com que seja muito custoso a utilização dos mecanismos de reflexão em todo grafo de referências;

b) A solução proposta em (RAYSIDE et al., 2009) executa uma grande quantidade de análises em tempo de execução, enquanto que na solução proposta muitas destas análises, como por exemplo a detecção de ciclos, são executadas em tempo de compilação;

c) A solução proposta utiliza-se de técnicas adaptativas (descrita na seção 4.2.4.1), em que a ordem das comparações é alterada em função dos dados inclusos nas instâncias sendo comparadas.

Já no que tange ao aspecto da corretude, conclui-se que a solução proposta substitui eficazmente as implementações dos métodos *equals()* e *hashCode()* inclusos na implementação da biblioteca JFreeChart. Porém, foram levantados os seguintes pontos:

a) Na suíte de testes de unidade, alguns testes passaram a falhar por efetuarem a verificação de um valor de código *hash* especifico, diferente do gerado pela solução;

b) Alguns testes de unidade eram fiéis à lógica de verificação de igualdade implementada de maneira errônea;

c) Em função da checagem em tempo de execução de alterações em propriedades utilizadas pela lógica de verificação de igualdade, alguns testes passaram a falhar após a primeira invocação aos métodos *equals()* e *hashCode()*.

4.2.5 Conclusão

Conforme visto, existe uma gama de soluções que tem como alvo o problema da definição da lógica de verificação da igualdade e sua correta implementação, cumprindo o contrato que rege a lógica desta verificação.

No campo dos geradores de código, foram descritos os geradores embutidos nos assistentes das IDEs NetBeans e Eclipse. Em análise preliminar, ambas implementações tiveram problemas quanto à geração de implementações com um grau adequado de robustez na presença de referências cíclicas e na manutenção das propriedades de simetria e transitividade do contrato que rege as verificações da igualdade quando existe uma relação de herança entre os operandos e na qual são introduzidos novos atributos, importantes para esta verificação.

A solução proposta em (VAZIRI et al., 2007) resolve o problema da verificação da igualdade através de um novo construto, o Tipo de Relação. Tipos de relação são distintos às classes por possuírem um ciclo de vida diferenciado: suas instâncias são obtidas através do método *id()*, que são compartilhadas como no padrão de projeto *flyweight* (GAMMA et al., 1994, p. 218) e, após a fase de construção, ficam proibidas alterações de quaisquer atributos importantes para a lógica de verificação da igualdade (atributos-chave). A adoção desse ciclo de vida diferenciado, aliada ao compartilhamento de instâncias, permite que as verificações de igualdade entre tipos de relação possam ser feitas através da sua identidade, o que permite a manutenção das propriedades de simetria e transitividade inerentes à relação de igualdade e o problema do uso de instâncias mutáveis em coleções com restrição de igualdade.

Porém, é encontrada na solução proposta em (VAZIRI et al., 2007) alguns problemas importantes:

a) As restrições impostas pelo novo construto podem impedir a integração entre tipos de relação e *frameworks* com alto grau de penetração, como por exemplo o Hibernate;

b) As alterações executadas pelo processo de compilação para a geração do código-objeto acarretam em uma lacuna, que pode impedir o uso de depuradores;

c) Não são disponibilizados pela solução nenhum recurso para a definição da lógica de verificação da igualdade em classes definidas por terceiros;

d) Não existe nenhum mecanismo de proteção a recursões infinitas na presença de referências cíclicas[31];

e) Conforme (RUPAKHETI, 2010, p. 117), não é possível, com a utilização da solução, a definição da lógica de verificação da igualdade utilizando tanto a estratégia de comparação híbrida quanto a estratégia de comparação com compatibilidade de tipos.

Já a solução apresentada em (RAYSIDE et al., 2009) é baseada no conceito de funções de abstração (LISKOV; WING, 1994), funções essas que mapeiam o estado de um objeto em um conjunto de valores, que por sua vez pode ser comparado pela lógica de verificação de igualdade; caso os conjuntos sejam iguais, os objetos serão considerados iguais, caso contrário, serão considerados diferentes. De maneira análoga à solução proposta em (VAZIRI et al., 2007), são utilizadas anotações para a definição da lógica de verificação da igualdade. Também foram encontradas nessa solução alguns problemas que impactam a sua adoção:

[31] Apesar de as comparações entre tipos de relação serem feitas através de suas respectivas identidades, ainda são executadas, no método *id()*, verificações tendo como alvo os valor armazenados nos atributos-chave do tipo de relação correspondente.

a) Apesar de efetuar um tratamento de referências cíclicas, o algoritmo utilizado para a definição do código *hash* na presença de ciclos tem um impacto sensível no desempenho do programa;

b) Quando é utilizada uma estratégia de comparação diferente da estratégia de comparação com compatibilidade de tipo (RUPAKHETI, 2010, p. 6 et seq.), é necessário que o programador implemente a função de abstração da maneira tradicional imperativa;

c) A solução, conforme descrita, não apresenta nenhuma alternativa de definição de estratégias de comparação customizadas, o que pode impedir a definição da lógica de verificação da igualdade em alguns casos, conforme estudo realizado em (RUPAKHETI, 2010, p. 118), em que foi impossível a definição da lógica de verificação da classe *org.apache.lucene.search.Query*, pertencente à biblioteca Lucene[32] (versão 3.0).

A solução proposta em (GRECH; RATHKE; FISCHER, 2010) também é baseada no conceito de programação baseada em atributos. Porém, ao invés da geração de código-fonte intermediário (VAZIRI et al., 2007) e do uso dos mecanismos de reflexão (RAYSIDE et al., 2009), sua implementação é baseada nos conceitos de orientação a aspectos combinadas com técnicas de geração de código. A solução proposta em (GRECH; RATHKE; FISCHER, 2010) tem muito em comum com a solução apresentada em (RAYSIDE et al., 2009), com algumas diferenças importantes.

a) Não é necessária a implementação de um método encapsulando uma função de abstração – é possível a definição das estratégias de verificação da igualdade com e sem compatibilidade de tipos (RUPAKHETI, 2010) apenas com o uso das anotações definidas (porém não parece ser possível a utilização da estratégia de comparação hibrida);

b) É incorporado à solução um mecanismo de definição de regras de verificação da igualdade de classes de terceiros, porém, não é citado por (GRECH; RATHKE; FISCHER, 2010) a forma de definição desta lógica de verificação nem seus possíveis contextos de utilização;

c) As instâncias pertencentes a uma classe cuja lógica de verificação de igualdade é definida através de solução proposta são do tipo com estado mutável; logo, toda alteração de atributos utilizados pela lógica de verificação é proibida em tempo de execução;

d) São inclusos mecanismos de otimização dinâmica da operação *equals()*, que têm como foco a reorganização das checagens realizadas na operação, na tentativa de se eliminar verificações redundantes.

[32] Apache Lucene. Disponível em: ⟨http://lucene.apache.org/core/⟩.

Vale citar que nenhuma das soluções até aqui abordadas provê mecanismos ou construtos para a declaração da ordem natural da comparação entre objetos. Em todas as soluções, fica a cargo do programador a implementação do cálculo de ordem natural de maneira consistente com a semântica de igualdade adotada e respeitando as características[33] dessa relação.

Tendo em vista as desvantagens encontradas nas soluções apresentadas, propõem-se neste trabalho uma solução, que, como as soluções analisadas, permite a definição da semântica de verificação de igualdade de maneira declarativa, porém estende este mecanismo de definição, possibilitando a definição do cálculo da ordem natural. A solução do presente trabalho também deverá prover mecanismos para a redefinição da lógica de verificação de igualdade de classes cujo código-fonte estejam indisponíveis, permitindo o uso de classes pertencentes a bibliotecas de terceiros, cujas as implementações das operações *equals()* e *hashCode()* violem o contrato que rege estas operações.

[33] Reflexividade, Simetria, Transitividade, Consistência e Não-nulabilidade (conforme seção 5.2).

5 Solução proposta para a comparação de objetos

Neste capítulo será apresentada uma solução para o problema da definição da lógica de comparação entre objetos. A solução apresentada neste capítulo tem como objetivo a mitigação dos problemas encontrados nas solução previamente propostas, discutidas em detalhes no capitulo anterior.

A apresentação da solução é divida em três seções, de maneira a facilitar ao leitor uma comparação com as soluções discutidas no capítulo anterior. A primeira seção tem como objetivo a descrição das características principais (como por exemplo o ferramental utilizado e as técnicas de programação nas quais a solução se baseia) da solução sendo apresentada.

Na segunda seção é discutido o cumprimento, pela solução, do contrato que rege as operações de verificação da igualdade e seus subcontratos. Essa discussão é apoiada por exemplos hipotéticos, descrevendo o comportamento interno da solução e a linha de raciocínio utilizado para a conclusão de que a solução apresentada cumpre as premissas do contrato de igualdade.

A terceira seção apresenta as formas de uso da solução proposta. Nessa seção são discutidos com detalhes os elementos pertecentes a API com a qual o desenvolvedor se comunica com a solução aqui proposta, bem como exemplos de sua utilização.

Durante todo o capitulo são feitos paralelos entre as características da solução proposta e as decisões tomadas nas soluções prévias (VAZIRI et al., 2007; RAYSIDE et al., 2009; GRECH; RATHKE; FISCHER, 2010), no intuito de justificar decisões tomadas ou exemplificar vantagens da solução proposta.

5.1 Caracteristicas principais

É possível, segundo (CARDELLI; MITCHELL, 1990), se traçar um paralelo entre os conceito de registro e de objetos, adotado no paradigma de programação orientado a objetos. Pode-se definir registro de maneira informal como:

> "Um registro é usado para representar, de uma maneira semanticamente intuitiva, um mapeamento finito entre identificadores e valores, onde esses valores podem pertencer a diferentes tipos. Sintaticamente um registro é uma coleção de campos identificado" (CARDELLI; MITCHELL, 1990, tradução do autor)

Tendo em vista esta definição, cada campo de um registro pode ser visto como um atributo de um objeto arbitrário, cujos campos são manipulado por funções (métodos).

A solução aqui utiliza-se desta representação em registros para a verificação da lógica de comparação ente objetos, porém, ao contrário do paralelo traçado em (CARDELLI; MITCHELL, 1990), os registros de comparação não são construídos apenas em função apenas de um objeto, mas sim do contexto da operação de verificação de igualdade sendo realizada. No decorrer deste capítulo serão fornecidos exemplos indicando os diferentes registros obtidos de acordo com os operandos fornecidos às operações de verificação de igualdade.

De maneira análoga às soluções anteriormente apresentadas (VAZIRI et al., 2007; RAYSIDE et al., 2009; GRECH; RATHKE; FISCHER, 2010), a solução proposta utiliza o mecanismo de suporte à programação baseada em atributos[1]. Conforme discutido anteriormente, a utilização dos conceitos da programação baseada em atributos pode permitir um grau de expressividade maior ao programador, o que acarreta uma diminuição substancial do código a ser escrito e posteriormente mantido, bem como uma diminuição da complexidade do software (GUERRA; FERNANDES, 2013).

Ao contrário das soluções apresentadas por (VAZIRI et al., 2007) e (RAYSIDE et al., 2009), que se utilizam da geração de código-fonte e do mecanismo de reflexão da linguagem Java, respectivamente. A solução aqui proposta utiliza-se de conceitos de orientação a aspectos como *weaving* e *mixin* (vistos no capítulo 2) na sua implementação. A utilização da tecnologia de programação orientada a aspectos permite a utilização do ferramental comumente utilizado pelos desenvolvedores como depuradores, profiladores, bem como a fácil integração da solução no ferramental de compilação utilizado.

Cálculo de Ordem Natural

É recomendado que os desenvolvedores definam a lógica responsável pelo cálculo da ordem natural de instâncias de um determinado tipo, uma vez que a definição desta lógica permite a utilização dos mecanismos de ordenação disponíveis nas bibliotecas da linguagem Java (BLOCH, 2008, p. 62). Diferentemente das soluções apresentadas anteriormente, a solução aqui proposta permite a definição da lógica do cálculo de ordem natural de maneira consistente com a lógica de verificação de igualdade definida pelo desenvolvedor. Não obstante, a implementação do mecanismo de cálculo da ordem natural deve cumprir todas as exigências de contrato definidas na seção 5.2.

Conforme previamente relatado em (RUPAKHETI, 2010, p. 118) na falha ao expressar alguns casos de uso com a solução proposta por (RAYSIDE et al., 2009) e corroborado por (BLOCH, 2006) quando diz que uma API deve ser suficientemente poderosa para satifazer

[1] Também chamado de anotações e disponível desde a versão 1.5 da linguagem Java

a todos os requisitos, é de suma importância que a solução do problema da igualdade entre objetos tenha um grau de expressividade que permita a sua utilização em uma gama variada de casos. Para isso, é disponibilizado para o programador a customização, para o cálculo de ordem natural, do grau de avaliação e da ordem de comparação utilizada (essas opções serão detalhadas e exemplificadas na seção 5.3, tratando das forma de utilização da solução).

Na seção subsequente será exemplificado como a solução aqui proposta trata o mecanismo de cálculo de ordem natural de maneira compatível com os contratos de verificação de igualdade apresentados no capítulo 3.

5.2 Contrato de Igualdade

Nesta seção, são tratados os mecanismos utilizados pela solução proposta para o tratamento do contrato de igualdade e seus subcontratos, definidos com detalhes na seção 5.2. Será dado enfoque às operações utilizadas pela linguagem Java para a definição da lógica de comparação entre objetos (*equals()* e *compareTo()*) e a interdependência entre as suas respectivas implementações. Após o tratamento das premissas dos subcontratos, será dado enfoque ao tratamento da interdependência entre a operação *equals()* e a operação *hashCode()*. Nesta seção também serão estudadas as premissas do cálculo da ordem natural, que tem suporte da solução apresentada.

5.2.1 Subcontratos

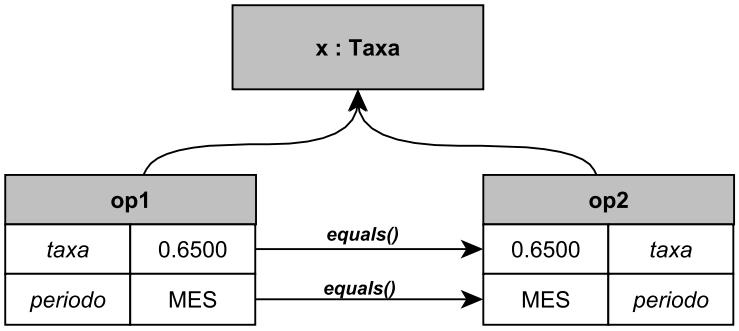
Conforme definido na seção 5.2, podemos dividir o contrato de igualdade em cinco subcontratos cada um regendo um aspecto da verificação de igualdade. Esses subcontratos são dividos em: reflexividade, simetria, transitividade, consistência e não-nulabilidade.

5.2.1.1 Reflexividade

O subcontrato de reflexividade dita que uma instância sempre deverá ser considerada igual a ela mesma. Para que esse subcontrato seja cumprido, basta apenas que o mecanismo utilizado para a construção do registro de comparação seja determinístico.

Para exemplificar, pode-se imaginar uma classe chamada *Taxa*, que por sua vez tem duas propriedades (*percentual* e *periodo*), importantes para a lógica de verificação de igualdade. Supondo que o mecanismo de geração do registro de comparação realmente seja determinístico, pode-se inferir que os registros para comparação gerado serão sempre iguais, desde que não sejam feitam alterações nos valores de *percentual* e *periodo*. Sendo assim, quando efetua-se a operação $x = x$, tem-se internamente a seguinte representação em registros, ilustrada na figura 8.

Figura 8 – Representação em Registro - Reflexividade



Pode-se notar na figura uma equivalência entre as informações dispostas no registro montado com a referência fornecida como o primeiro operando ($op1$) e as informações dispostas no registro formado através do segundo operando ($op2$). Uma iteração que compare as informações contidas em ambos registros chegaria à conclusão que ambas referências são equivalentes.

É possível extrapolar esse resultado para o mecanismo de comparação de ordem natural. Como as informações contidas em ambos registros são iguais, todas as comparações entre as informações dispostas resultaria no valor zero (representando que os operandos têm a mesma ordem natural), o que por sua vez acarretaria no retorno do mesmo resultado após a invocação da operação de comparação, indicando assim que ambos operandos têm a mesma ordem natural.

5.2.1.2 Simetria

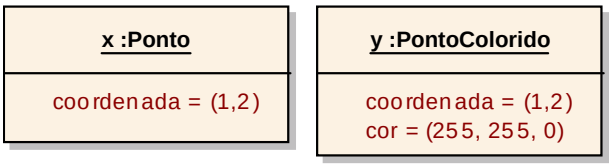
O subcontrato de simetria dita que, independentemente da ordem dos operandos[2], o resultado de uma operação de igualdade deve ser consistente. A implementação da simetria é trivial quando não existe nenhuma relação de herança entre os operandos, ou quando é utilizada a estratégia de definição de igualdade baseada em classe (conforme definido no capítulo 3).

Porém quando existe uma relação hierárquica entre os operandos, a manutenção da simetria das operações de igualdade torna-se difícil. Isso se deve ao mecanismo de despacho utilizado na linguagem Java, baseado no despacho simples. Conforme exposto no capítulo 2, em linguagens de despacho simples um dos operandos é utilizado para a escolha do método a ser executado no recebimento de uma mensagem. Logo, existe a possibilidade de métodos diferentes serem executados em função da ordem dos operandos, quebrando assim o subcontrato da simetria.

[2] Para efeito teórico, tanto o objeto que recebe a mensagem quanto o objeto que é passado como parâmetro são considerados operandos.

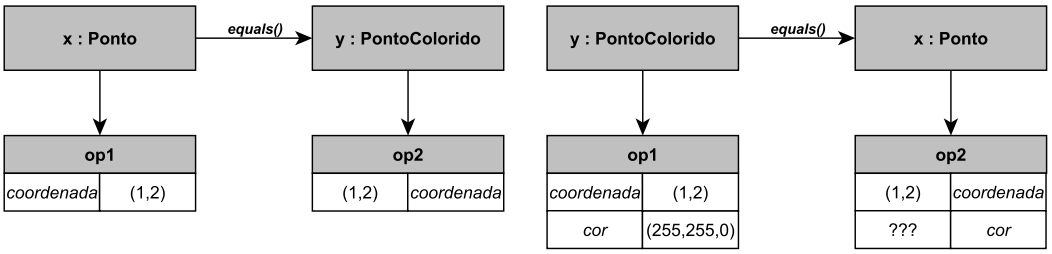
Podemos exemplificar o modo de manutenção da simetria utilizando o exemplo dado na explanação do subcontrato de simetria (seção 3.2.2), utilizando as classes *Ponto* e *PontoColorido*. Além dessa estrutura estática, pode-se assumir a existência de dois objetos distintos (chamados doravante de x e y). O objeto x é uma instância da classe *Ponto*, que tem como valor associado ao atributo *coordenada* a coordenada $(1, 2)$; já o objeto y é uma instância da classe *PontoColorido*, que também tem o valor de *coordenada* $(1, 2)$ e o valor do atributo *cor* $(255, 255, 0)$. A figura 9 apresenta o diagrama de objetos desse caso.

Figura 9 – Diagrama de Objetos - Simetria



Conforme anteriormente exposto, a escolha do método a ser executado em resposta à invocação de uma operação é feita somente em função do objeto contra o qual ela foi invocada. Assumindo que o mecanismo de construção de registros é indiretamente selecionado através do mesmo mecanismo, a figura 10 ilustra uma representação que considera a ordem dos operandos da operação de verificação de igualdade[3].

Figura 10 – Criação de Registros - Simetria (Despacho Simples)

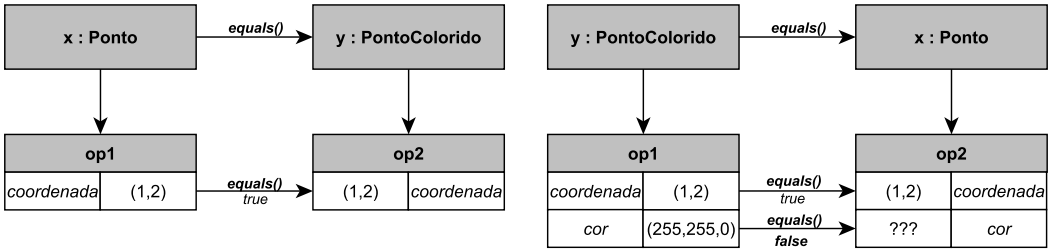


Na figura, pode-se concluir que a alteração da ordem dos operandos acarreta na gerações de registro distintos para a comparação. Quando o primeiro operando é a instância x (instância da classe *Ponto*), o registro de comparação tem somente o elemento coordenada. Já quando o primeiro operando é a instância y (pertencente ao tipo *PontoColorido*), os registros de comparação têm tanto a informação de coordenada quanto

[3] O valor *???* associado a um registro significa que seu valor é desconhecido. O valor desconhecido é diferente de uma referência nula, representada pelo valor *null*.

a informação referente a cor. Vale notar que a instâncias do tipo *Ponto* não contêm informações de cor, portanto o valor referente a essa informação no registro de comparação é considerado desconhecido. A figura 11 apresenta diagrama, que ilustra o processo de verificação de igualdade com a invocação do método *equals()*.

Figura 11 – Comparação de Registros - Simetria (Despacho Simples)



Conforme o diagrama, quando o objeto x é o primeiro operando da operação *equals()*, somente é feita a comparação dos valores associado ao atributo *coordenada*. Como o valor de coordenada de ambos é igual, os objetos são considerados iguais.

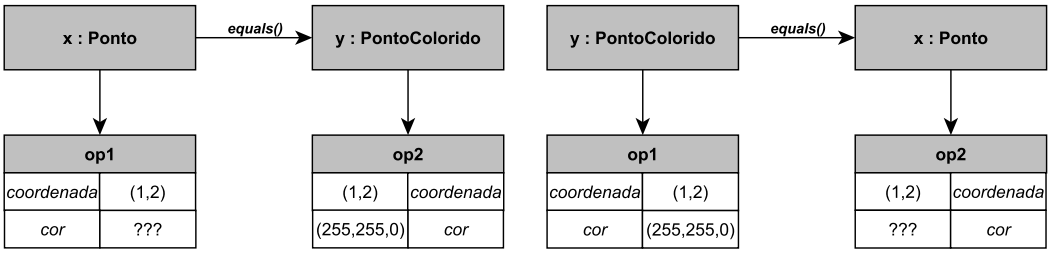
Já quando o primeiro operando é objeto y (pertencente à classe *PontoColorido*), são verificadas tanto as coordenadas dos pontos quanto as cores associadas a eles. Novamente o valor das coordenadas de ambos é considerado igual, porém, no momento da verificação de cor, existe uma discrepância: a cor associada ao objeto y tem um valor definido, enquanto o objeto x não tem essa informação associada. Essa discrepância faz com que os objetos sejam considerados diferentes.

Conforme visto, no caso de linguagens com despacho simples, é impossível manter o subcontrato de simetria, quando existe uma relação hierárquica entre classes, com as subclasses incluindo novas informações. Isso se deve a impossibilidade da superclasse conhecer as especializações inclusas em suas sub-classes.

Conforme exposto por (BRUCE et al., 1995), a resolução para esse problema reside na utilização de não somente do objeto que recebe a mensagem, como também dos parâmetros de uma operação, para a escolha do método responsável pelo processamento de uma operação, implementando assim um mecanismo de despacho múltiplo de mensagens. Supondo-se que um mecanismo de despacho múltiplo estivesse disponivel na linguagem Java[4], teria-se a disposição dos registros de comparação como ilustrada na figura 12.

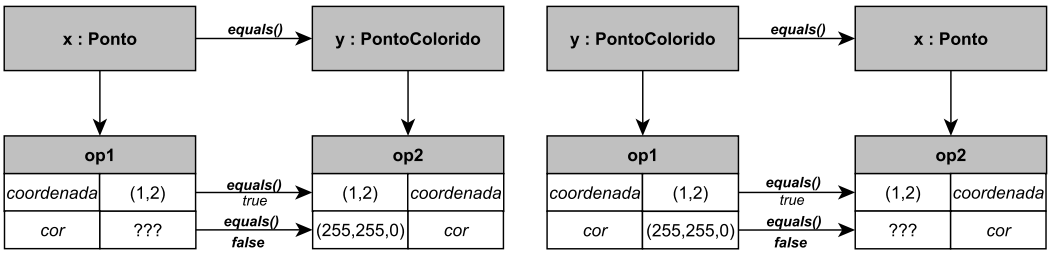
[4] A implementação do mecanismo simulando despacho múltiplo será tratada na seção 6.1.2.1

Figura 12 – Criação de Registros - Simetria (Despacho Múltiplo)



Pode-se ver na figura acima que, quando o mecanismo de despacho múltiplo é utilizado, a ordem dos operandos deixa de ter importância no processo de criação do registro de comparação. Isso ocorre em virtude da escolha do método a ser executado ser feita em função de ambos operandos, e não apenas do primeiro, como anteriormente. Com isso a verificação das invocações exemplificadas seria realizada como na figura 13.

Figura 13 – Comparação de Registros - Simetria (Despacho Múltiplo)



Conforme a figura, os resultados das verificações de igualdade entre os objetos são idênticos, independentemente da ordem dos operandos. No primeiro passo da comparação é verificada a informação referente à coordenada em ambos registros de comparação. Como ambos objetos têm a mesma coordenada, as informações são consideradas equivalentes. Já no segundo passo, são verificados os valores associados à cor. Somente o objeto y (pertencente ao tipo *PontoColorido*) tem um valor de cor associado, logo as informações de cores de ambos objetos são consideradas divergentes. Essa divergência no valor referente à cor dos objetos acarreta no resultado indicando que os operandos são diferentes.

Conforme visto, a utilização de um mecanismo de despacho múltiplo para a seleção da lógica de verificação de igualdade garante a manutenção do subcontrato de simetria, necessário para uma implementação correta das operações de igualdade.

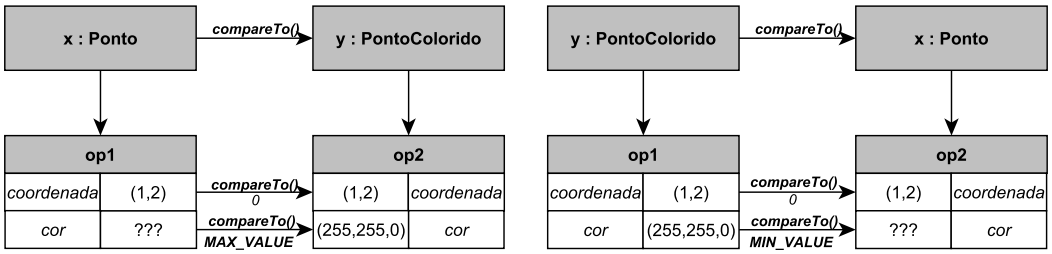
Manutenção de simetria na operação *compareTo()*

Conforme (BLOCH, 2008, p. 63) e descorrido na seção 3.2.2, também existe uma relação de simetria na comparação da ordem natural entre objetos. Essa relação de simetria, regida pela proposição $(\forall x \forall y)((x < y) \leftrightarrow (y > x))$, deve ser válida.

Para mostrar a simetria da operação *equals()* e exemplificar a forma de cálculo de ordem natural utilizada na solução proposta, definem-se dois objetos (x e y), pertencentes às classes *Ponto* e *PontoColorido*, respectivamente. Ambos os pontos têm o valor $(2, 3)$ como o valor de coordenada associado e o ponto y tem o valor $(255, 255, 0)$ associado à sua cor.

A construção do registro de comparação, utilizado pelo cálculo ordem natural, é idêntico aos registros construídos na verificação de igualdade (conforme ilustrado na figura 12). Após a construção dos registros, é efetuado o processo de cálculo da ordem natural utilizando os registros, conforme mostra a figura 14.

Figura 14 – Comparação de Registros - Simetria (Cálculo de Ordem Natural)



A comparação entre as informações associadas ao atributo coordenada é igual nos registros de ambos objetos. Como a informação referente a cor está disponível somente no registro criado a partir do objeto y, a solução considera o ponto x menor que o ponto y (valores desconhecidos são considerados sempre menores que os conhecidos).

Pode-se ver na figura que o valor obtido no cálculo associado à operação *x.compareTo(y)* é a constante *Integer.MIN_VALUE*, que é o resultado oposto ao obtido por *y.compareTo(x)* (*Integer.MAX_VALUE*). Pode-se validar a simetria do cálculo de ordem natural através da expressão $signum^5(x.compareTo(y)) = -signum(y.compareTo(x))$ (BLOCH, 2008,

[5] A função *signum* (também conhecida como *sgn* ou função sinal) retorna o sinal de um número real dado, podendo ser expressa através da equação $sgnx = \begin{cases} -1 & \text{se } x < 0 \\ 0 & \text{se } x = 0 \\ 1 & \text{se } x > 0 \end{cases}$

p. 63), conforme desenvolvimento abaixo :

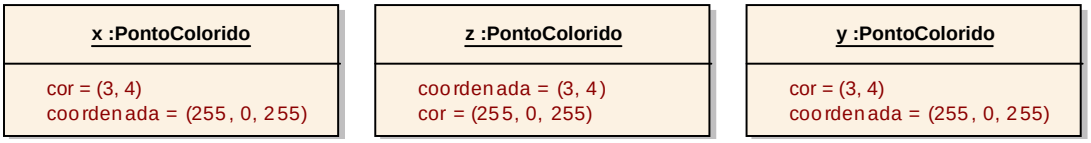
$$
\begin{aligned}
signum(x.compareTo(y)) &= -(signum(y.compareTo(x))) \\
signum(Integer.MIN_VALUE) &= -(signum(Integer.MAX_VALUE)) \\
-1 &= -(1) \\
-1 &= -1
\end{aligned}
$$

5.2.1.3 Transitividade

Como já foi dito, o subcontrato de transitividade diz que se um objeto é considerado igual a um segundo objeto, e esse segundo objeto por sua vez é considerado igual a um terceiro objeto, pode-se inferir que o primeiro objeto também é igual ao terceiro.

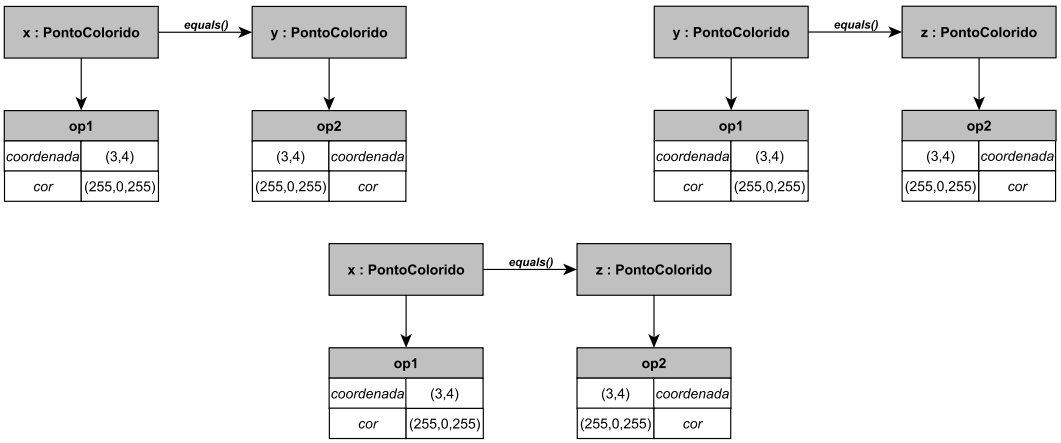
O cumprimento desse subcontrato está intimamente ligado ao cumprimento do contrato de simetria, conforme discorrido no capitulo 3. Através de um exemplo análogo ao exemplo dado na seção anterior, será mostrada a maneira com a qual a solução aqui apresentada cumpre esse subcontrato. Para tanto, definem-se três instâncias (doravante chamadas de x,y e z), todas pertencentes à classe *PontoColorido*. Em todas as três instâncias criadas serão associadas os mesmos valores de *coordenada* e *cor*, respectivamente $(3, 4)$ e $(255, 0, 255)$, conforme mostra a figura 15.

Figura 15 – Diagrama de Objetos - Transitividade



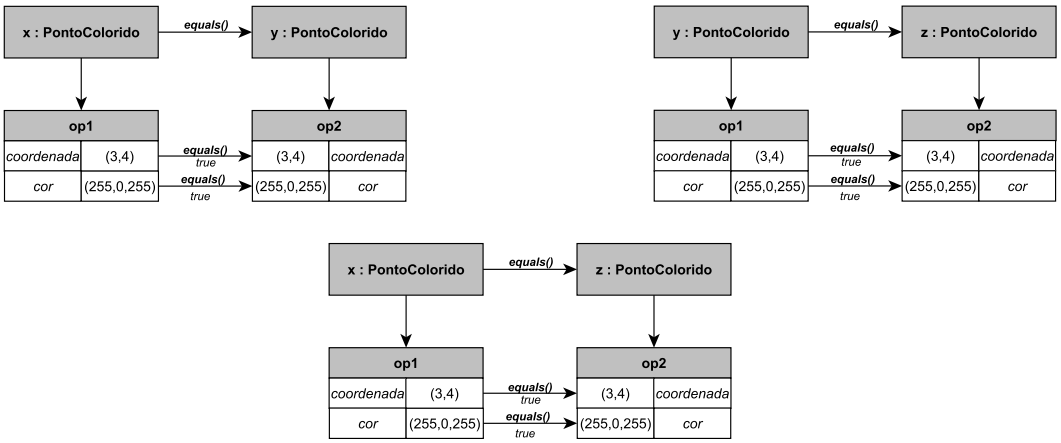
Conforme exposto no capitulo 3, o subcontrato de transitividade pode ser expresso através da proposição $(\forall x \forall y \forall z)((x = y) \wedge (y = z) \longrightarrow (x = z))$. A figura 16 mostra os registros criados para o processo de verificação da igualdade da solução proposta no que diz respeito à transitividade.

Figura 16 – Criação de Registros - Transitividade



O diagrama da figura 16 representa os registros de comparação criados pelas expressões $x = y$, $y = z$ (na parte superior , da esquerda para direita) e $x = z$ (na parte inferior) , respectivamente. Obtidos os registros de comparação, pode-se passar para a próxima fase do processo de verificação da igualdade, no qual serão comparadas as informações dos registros, conforme ilustrado na figura 17.

Figura 17 – Comparação de Registros - Transitividade



Seguindo o diagrama da figura 17, pode-se verificar que tanto as expressões $x = y$ e $y = z$ são verdadeiras, uma vez todos as informações compondo os registros de comparação gerados por essas operações são consideradas equivalentes, do mesmo modo que a expressão $x = z$. Assim, o mecanismo utilizado pela solução proposta para a verificação de igualdade consegue manter a característica de transitividade, inerente ao

contrato que rege as implementações da operação *equals()*, satisfazendo o subcontrato de transitividade.

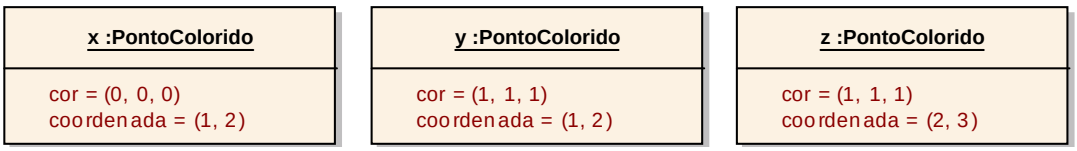
Manutenção de transitividade na operação *compareTo()*

O subcontrato de transitividade também afeta a operação *compareTo()*, conforme préviamente disposto na seção 3.2.3. A solução aqui proposta leva em conta esse efeito colateral no modo de cálculo da ordem natural de objetos.

Conforme (BLOCH, 2008, p. 63), é necessário que duas premissas sejam verdadeiras para que a operação *compareTo()* seja considerada transitiva. A primeira diz que a proposição $(\forall x \forall y \forall z)((x < y) \wedge (y < z) \longrightarrow (x < z))$ deve ser verdadeira (em outras palavras, se x for considerado menor que y e y for menor que z, então x também deve ser menor que z).

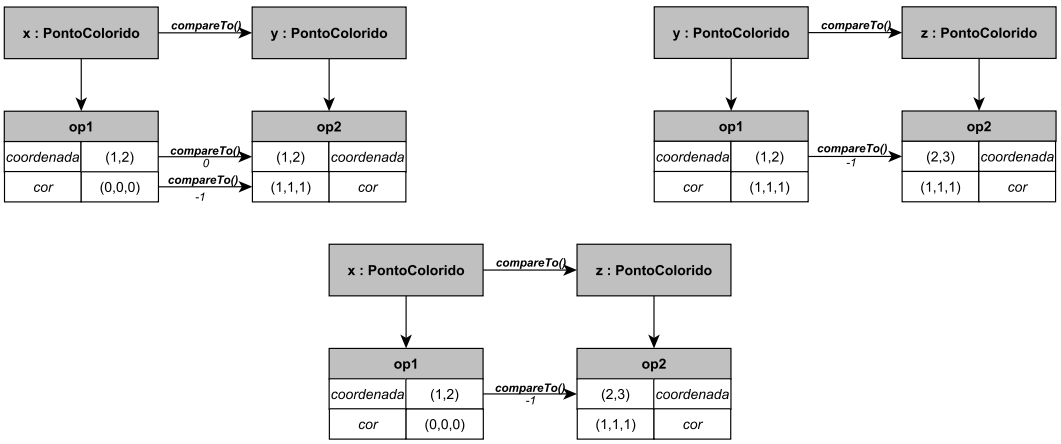
Para mostrar como a forma de cálculo de ordem natural utilizada na solução proposta, será utilizada a mesma estrutura hierárquica de classes dos exemplos anteriores, definindo três objetos (x,y e z), todos pertencentes à classe *PontoColorido*, conforme o diagrama da figura 18.

Figura 18 – Diagrama de Objetos - 1ª Premissa de Transitividade (Cálculo de Ordem Natural)



De acordo com o diagrama acima, os valores de coordenadas para os objetos x, y, z são, respectivamente $(1, 2)$, $(1, 2)$ e $(2, 3)$, já as cores associadas a eles são $(0, 0, 0)$, $(1, 1, 1)$ e $(1, 1, 1)$. Para efeito de exemplificação, pode-se assumir que $(1, 2)$ é considerada uma coordenada menor que $(2, 3)$ e a cor $(0, 0, 0)$ é considerada uma cor menor do que a cor $(1, 1, 1)$. Assume-se, aqui, que o atributo *coordenada* tem um peso maior na definição da ordem natural que o atributo *cor*. Levando em conta o processo de criação de registros de comparação, pode-se representar a forma de cálculo da solução como na figura 19.

Figura 19 – Comparação de Registros - 1ª Premissa de Transitividade (Cálculo de Ordem Natural)

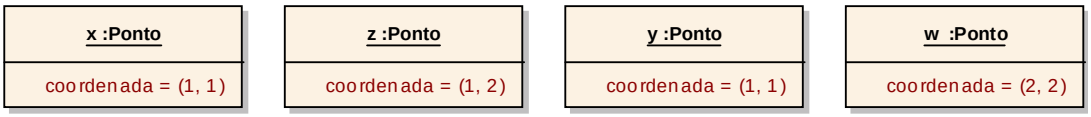


Conforme o diagrama da figura, o mecanismo de cálculo da ordem natural considera o objeto x menor que o objeto y, uma vez que o valor de cor associado ao primeiro é menor que o segundo. Em relação à comparação entre o objeto y e o objeto z, foi considerado pela solução que o objeto y é menor que o objeto z (em função do valor de coordenada associado a y ser menor que o valor de coordenada associado ao objeto z).

Seguindo a premissa, o valor da comparação entre x e z deveria obrigatoriamente considerar que x é menor que z. O resultado obtido através do método de cálculo da solução é exatamente esse (o valor da coordenada de x é menor que o valor da coordenada de z), cumprindo os requisitos da premissa.

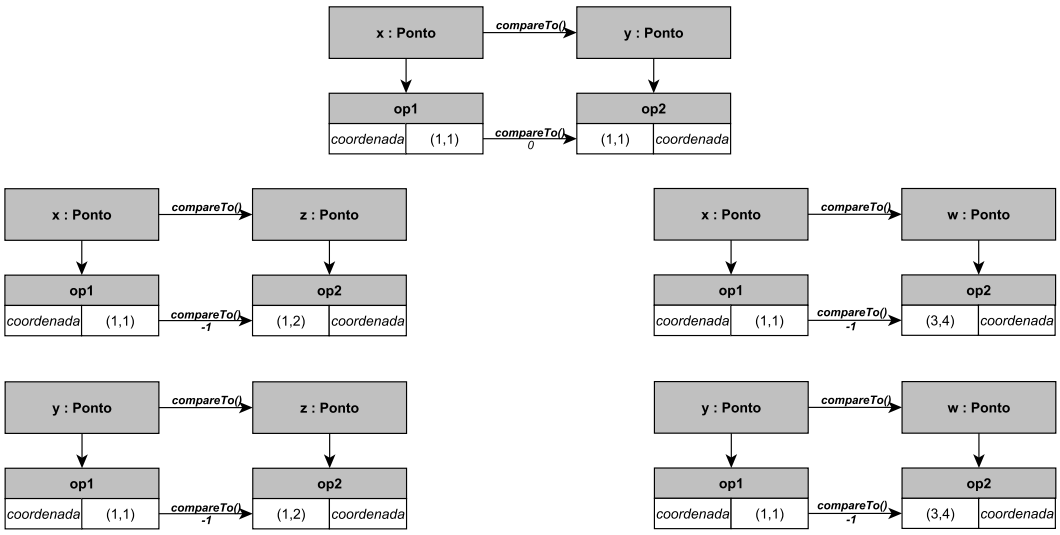
A segunda premissa a ser seguida pela operação *compareTo()* dita que, se dois objetos (x e y) forem considerados iguais, qualquer objeto z considerado maior que x, deve ser também considerado maior que y. Adiante encontra-se um exemplo indicando o cumprimento da segunda premissa pela solução aqui apresentada. Para esse exemplo, são definidas quatro instâncias de *Ponto* (x, y, z, w), aos quais são associados, respectivamente, os seguintes valores de coordenada : $(1, 1)$, $(1, 1)$, $(1, 2)$ e $(2, 2)$. A figura 20 mostra um diagrama de objetos descrevendo essa configuração.

Figura 20 – Diagrama de Objetos - 2ª Premissa de Transitividade (Cálculo de Ordem Natural)



Seguindo o processo de verificação de ordem natural, são construídos registros de comparação para cada um dos operandos envolvidos nas operações de verificação, registros estes que posteriormente serão utilizados para o cálculo da ordem natural dos operandos. Levando-se em conta as comparações necessárias para a validação da segunda premissa ($x = y$, $x < z$, $x < w$, $y < z$, $y < w$), obtém-se os registros de comparação conforme o diagrama da figura 21.

Figura 21 – Comparação de Registros - 2ª Premissa de Transitividade (Cálculo de Ordem Natural)



Analisando diagrama acima, pode-se ver que o objeto x é considerado menor que os objetos z e w (o resultado das invocações $x.compareTo(z)$ e $x.compareTo(w)$ é -1) e considerado equivalente ao objeto y (o resultado da chamada de método $x.compareTo(y)$ é 0). De acordo com a premissa, a equivalência entre o objeto x o objeto y implica que o objeto y também seja considerado maior que os objetos z e w. Pode-se notar no desenho disposto na figura 21, que o valor obtido através das invocações $y.compareTo(z)$ e $y.compareTo(w)$ é -1, o que indica que y é considerado menor que z e w.

5.2.1.4 Consistência

Conforme discutido na seção 3.2.4, o subcontrato da consistência dita que dois objetos considerados iguais, devem sempre assim considerados, a menos que sejam modificados. Tendo isso em vista, pode-se enumerar duas questões importantes necessários para a manutenção do subcontrato de consistências: a construção de registros de maneira determinística e a transitividade das operações de verificação de igualdades subsequentes.

Quanto a questão da construção do registros, pode-se inferir a característica determinística do processo através do método de construção do registro de comparação,

onde, conforme visto, são utilizadas informações referentes aos construtos anotados com a anotação *@Comparable* para a construção do registro. Essas informações são, durante a execução de um programa, imutáveis, logo, dado um objeto x a construção de um registro de comparação será sempre executada da mesma maneira.

Já no que tange a questão da transitividade das suboperações, a solução proposta neste trabalho não provê nenhum mecanismo para a verificação da consistência de quaisquer operações que venham a ser invocadas no processo de verificação da igualdade. Essa deficiência é também encontrada nas soluções apresentadas e analisadas no capítulo 4 (VAZIRI et al., 2007; RAYSIDE et al., 2009; GRECH; RATHKE; FISCHER, 2010). Uma possível forma de mitigação deste problema é a inclusão de mecanismos de análise de código, discutida com mais propriedade na seção 5.2.2.

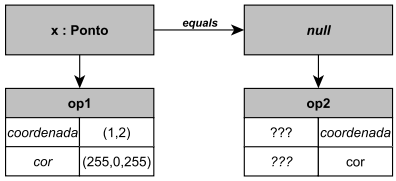
5.2.1.5 Não-Nulabilidade

O subcontrato de não-nulabilidade rege, conforme explicado no capítulo 3, o valor esperado da verificação de igualdade quando um dos operandos é nulo. Segundo o subcontrato, uma instância e uma referência nula devem ser consideradas diferentes. Também, conforme recomendação de (BLOCH, 2008, p. 41), deve ser assegurado que invocações desse tipo não levantem exceções do tipo *NullPointerException*.

De acordo com o subcontrato de simetria, a ordem dos operandos numa operação de verificação de igualdade não deve influenciar o resultado final. Como o mecanismo de passagem de mensagens, inerente à linguagens orientadas a objetos, necessita de um objeto receptor, somente é levada em conta a nulabilidade do segundo operando da operação.

Para a construção do registro de comparação, é utilizado sempre o tipo do operando cuja referência associada não é nula. Por exemplo, dada uma chamada de operação *x.equals(null)* onde x é uma instância da classe *PontoColorido*, com o valor de *cooordenada* e *cor* $(1, 2)$ e $(255, 0, 255)$, respectivamente, tem-se a configuração de registros de comparação representada na figura 22.

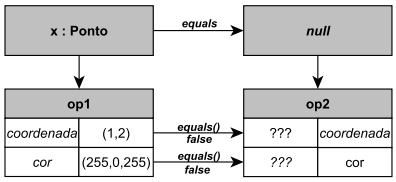
Figura 22 – Criação de Registros - Não-nulabilidade



Pode-se verificar na figura que o registro de comparação relativo ao primeiro operando tem todas as informações associadas a instância x, já o registro de comparação associado

à referência nula considera as informações de cor e coordenada desconhecidas. A segunda parte do processo de comparação é mostrado no diagrama na figura 23.

Figura 23 – Comparação de Registros - Não-nulabilidade)



Como todas as informações incluídas nos registros de comparação criados a partir dos operandos são consideradas diferentes, concluí-se que os operandos (x e $null$) também são considerados diferentes.

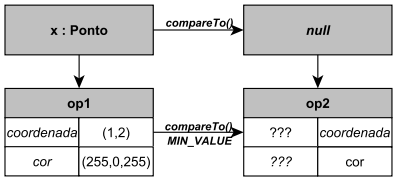
Manutenção de não-nulabilidade na operação *compareTo()*

A especificação da interface *Comparable*[6] diz que não é necessário um tratamento de operandos nulos na operação *compareTo()*[7]. A solução aqui apreesentada implementa um tratamento especial para invocações de *compareTo()* quando a referência passada como argumento for nula.

Conforme visto anteriormente, o contrato da verificação de igualdade dita que a expressão $x.equals(null)$ deve ter como resultado sempre o valor *false*. De acordo com a lógica adotada na seção 3.2.5, o valor da expressão $x.compareTo(null)$ deve ser diferente de zero para que a consistência entre as operações *equals()* e *compareTo()* seja mantida.

A figura 24 exemplifica a configuração criada pela solução no cálculo do resultado da operação *compareTo()* para esse caso, em que o objeto x, definido como no exemplo anterior é comparado com uma referência nula.

Figura 24 – Cálculo de Registros - Não-nulabilidade (Cálculo de Ordem Natural)



[6] Comparable (Java Platform SE 7). Disponível em: ⟨http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html⟩.

[7] Segundoa documentação, uma implementação da operação *compareTo()* deve levantar uma exceção do tipo *NullPointerException* caso seu operando seja nulo.

Como se pode ver, o resultado da comparação entre o objeto x e a referência nula é igual à constante $Integer.MIN_VALUE$, resultado este consistente com o resultado esperado pela premissa definida previamente, em que $x.compareTo() \neq 0$. Logo, pode-se concluir que a forma de cálculo adotada pela solução cumpre os requisitos de não-nulabilidade.

5.2.2 A transitividade das sub-comparações

Conforme exemplos exibidos anteriormente, a solução aqui apresentada consegue cumprir os cinco subcontratos que compõem o contrato de verificação da igualdade. Porém todo o funcionamento desse mecanismo parte de uma premissa básica, que deve ser verdadeira em todos momentos: todas as operações de verificação, que venham a ser executadas pela solução na verificação da igualdade, devem cumprir integralmente o contrato da igualdade.

Essa exigência torna-se necessária em virtude da transitividade das violações de contrato, inerente ao processo de verificação (e de cálculo de ordem natural) utilizado pela solução. A solução normalmente delega a lógica de comparação para a lógica definida nas classes que implementam os elementos que contêm as informações pertencentes aos operandos sob comparação.

Pode-se exemplificar essa transitividade utilizando a classe *Ponto*, utilizada amplamente nos exemplos já dados. Como visto anteriormente, a lógica de verificação de igualdade de *Ponto* depende do resultado de obtido, por delegação, da comparação dos valores associados ao atributo *coordenada* de ambos operandos. Essa comparação é efetuada pelas operações de comparação definidas na classe *Coordenada*, invocadas através de delegação.

Supondo que a implementação dos métodos de comparação definidos na classe *Coordenada* não sejam consistentes: no momento da comparação entre os valores de *coordenada* ocorrerá uma violação no subcontrato de consistência, que se propagará para a comparação original entre os operandos do tipo *Ponto*.

Para a solução desse problema, pode-se implementar uma estratégia de comparação customizada, conforme explicado na seção 5.3.1.2. Porém a solução não provê mecanismos para a detecção de problemas de violação de contrato em classes cuja lógica de comparação não seja definida pelo uso da anotação *@Comparable*. Uma possível solução para esse problema seria a incorporação de módulos de análise estática e/ou dinâmica de código, preparados para a identificação dessas violações.

5.2.3 Interdependência entre as operações *equals()* e *hashCode()*

Conforme previamente disposto no capitulo 3, a implementação da operação *hashCode()* está intimamente ligada à implementação da operação *equals()*. É necessário que,

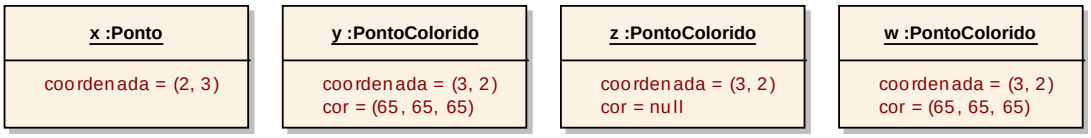
caso duas instâncias sejam consideradas iguais pelo método *equals()*, o resultado retornado pela operação *hashCode()* de ambas também sejam iguais.

Para o cálculo do código de *hash*, a solução aqui apresentada utiliza-se da fórmula que segue, na qual x é o objeto cujo código *hash* se deseja calcular, o conjunto S representa o conjunto de todas as informações importantes para a verificação de igualdade e a função f é a função para cálculo de código hash, associada à cada uma dessas informações :

$$hashCode(x) = \sum_{y \in S} f(y)$$

De maneira semelhante aos outros exemplos, a figura 25 mostra um exemplo descrevendo a operação do mecanismo de cálculo de código *hash*, mostrando que os resultados obtidos nas chamadas de *equals()* e *hashCode()* são consistentes entre si. Define-se quatro objetos, x, y, z, w. O objeto x é uma instância de *Ponto*, já os objetos y,z e w são instâncias da classe *PontoColorido*. A estrutura estática das classes *Ponto* e *PontoColorido* é a mesma utilizada nos exemplos anteriores.

Figura 25 – Diagrama de Objetos - Interdependência entre *equals()* e *hashCode()*



Como mostra o diagrama da figura, o objeto x tem associado ao seu atributo *coordenada* o valor $(2, 3)$, já os objetos y, z e w têm o valor $(3, 2)$ associados às suas coordenadas. A valor de *cor* dos objetos y e w é $(65, 65, 65)$, enquanto o valor armazenado no objeto z é nulo.

Levando-se em conta tanto os valores associados quanto a lógica de comparação, definida tanto na classe *Ponto* como na classe *PontoColorido*, pode-se sumarizar na Tabela 9 os resultados esperados das comparações entre eles :

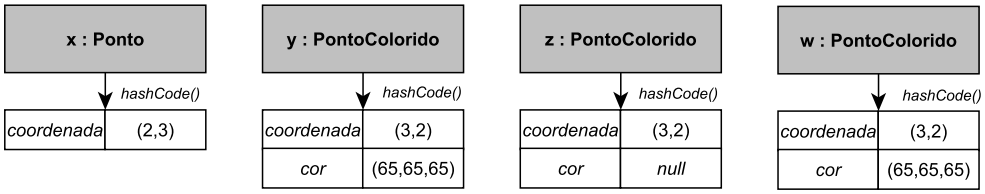
Tabela 9 – Equivalência esperada - Interdependência de operações

	x	y	z	w
x	Igual	Diferente	Diferente	Diferente
y	Diferente	Igual	Diferente	Igual
z	Diferente	Diferente	Igual	Diferente
w	Diferente	Igual	Diferente	Igual

Seguindo o processo de obtenção do código de *hash*, é construído, para cada objeto, um registro de comparação contendo as informações importantes para a lógica de verificação

da igualdade. O diagrama da figura 26 representa os registros obtidos para cada um dos objetos.

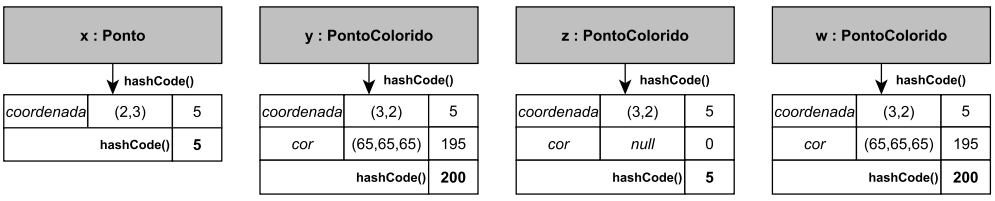
Figura 26 – Criação de Registros - *hashCode()*



Pode-se ver no diagrama que o registro construído pelo processo de cálculo do código *hash* do objeto x, contém apenas o valor de coordenada, ao contrário dos registros obtidos nos exemplos anteriores. Isso acontece em razão da utilização de um só operando na operação para o cálculo de código *hash*. Já os registros obtidos através dos outros objetos tem tanto as informações de coordenada quanto a de cor.

Na próxima fase do processo de cálculo do código de *hash*, são calculados, através da somatória do código de *hash* de todos os elementos inclusos nos registros de comparação, os códigos de *hash* de cada um dos objetos. Para efeito de simplicidade, assume-se que os códigos de *hash* de cores e coordenadas são calculados através da soma de seus componentes. A figura 27 apresenta os valores parciais e totais do código hash de cada um dos objetos :

Figura 27 – Cálculo de código hash através de registros de comparação



Pode-se verificar na figura que o valor dos códigos de *hash* dos objetos x,y,z e w são, respectivamente, 5, 200, 5 e 200. Pode-se assim complementar a Tabela 9 com as informações de código de *hash*, obtendo-se assim a Tabela 10.

Observa-se na Tabela 10 que em todos os casos em que o cálculo de igualdade indicara operandos equivalentes, eles tiveram o mesmo valor de código de *hash* obtido. Esse resultado está de acordo com a relação de interdependência entre as operações *equals()* e

Tabela 10 – Código *hash* obtidos - Interdependência de operações

		x	y	z	w
	hash	5	200	5	200
x	5	Igual	Diferente	Diferente	Diferente
y	200	Diferente	Igual	Diferente	Igual
z	5	Diferente	Diferente	Igual	Diferente
w	200	Diferente	Igual	Diferente	Igual

hashCode(), na qual instâncias consideradas iguais devem ter obrigatoriamente o mesmo valor de código de *hash*.

Também é interessante notar que, apesar de serem considerados diferentes, os códigos de *hash* dos objetos x, z calculados pela solução são iguais. Conforme (BLOCH, 2008, p. 47), esse resultado, apesar de poder em alguns casos degradar o desempenho do *software*, não indica uma violação do contrato que rege a relação entre as operações *equals()* e *hashCode()*.

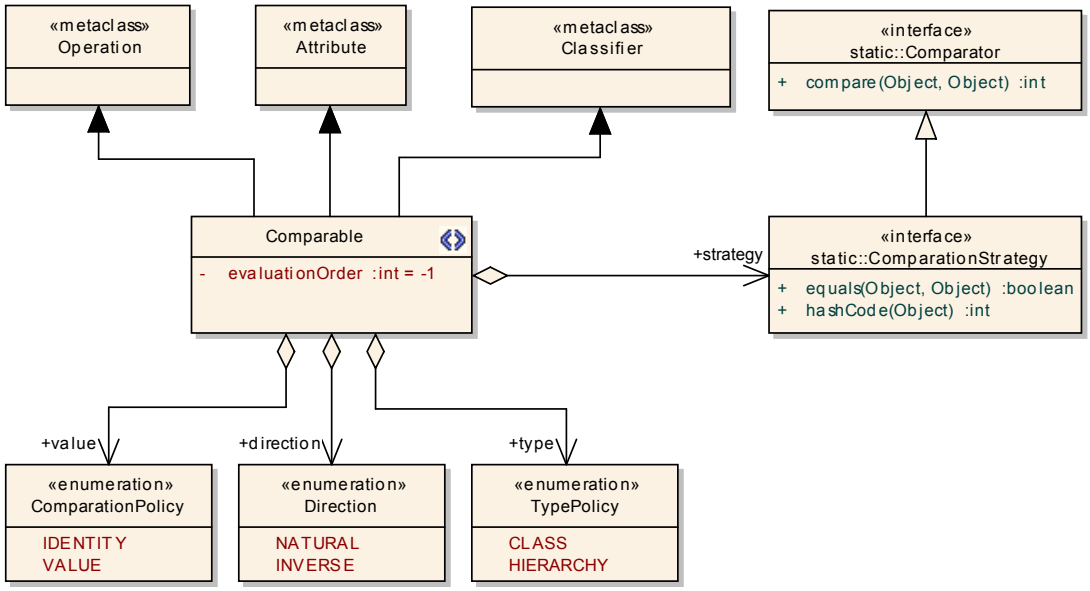
5.3 Formas de Uso

Tendo como o objetivo facilitar o entendimento dos recursos disponíveis, será aqui descrita a solução sob a ótica do modo de uso pelos desenvolvedores. Primeiramente, descrevem-se todos elementos disponíveis na API para a definição da lógica de comparação serão explicadas. Em seguida, apresentam-se exemplos mostrando o uso desses elementos.

5.3.1 Elementos da API

A solução aqui proposta para a definição da lógica de comparação entre objetos baseia-se na utilização de anotações realizadas pelo desenvolvedor. A figura 28 apresenta um diagrama de perfil (baseado no versão 2.0 da UML), representando as anotações disponíveis para a definição da lógica de verificação de igualdade, bem como seus atributos.

Figura 28 – Diagrama de perfil - Solução proposta



Conforme o diagrama da figura, pode-se identificar a anotação *@Comparable*, que deve ser utilizada pelo programador para a definição dos aspectos relativos à lógica de verificação de igualdade dos elementos (atributos, métodos, classes e interfaces) pertencentes a uma classe cuja lógica deseja-se implementar. Além da anotação *@Comparable*, é definida uma interface chamada *ComparationStrategy*, agrupando as operações a serem implementadas quando o desenvolvedor deseja incorporar à solução mecanismos customizados de verificação de igualdade entre objetos. A seguir, ambos elementos serão explicados em detalhes.

5.3.1.1 A anotação *@Comparable*

Ao contrário das solução proposta por (RAYSIDE et al., 2009), (VAZIRI et al., 2007) e (GRECH; RATHKE; FISCHER, 2010), que definem, respectivamente, quatro, três e duas anotações para a definição da semântica de comparação entre objetos, a solução aqui proposta utiliza apenas uma anotação.

Anotando interfaces e classes

A utilização da anotação *@Comparable* em classes ou interfaces indica à solução o desejo do desenvolvedor em definir uma lógica de verificação de igualdade, específica para instâncias do tipo representado pela classe ou interface anotada. Quando utilizada nesse contexto, é possibilitada ao programador a definição, através do parâmetro *strategy*, de uma lógica de comparação customizada para o tratamento de outras lógicas de comparação. Classes (ou interfaces) anotadas com a anotação *@Comparable* automaticamente

realizam (ou estendem) a interface *java.lang.Comparable*, responsável pela definição das operações relativas ao cálculo da ordem natural entre objetos.

Outra opção disponibilizada para os desenvolvedores quando a anotação *@Comparable* é utilizada no contexto de uma classe ou interface, é a estratégia de definição de igualdade, conforme descrito na seção 3.1.3 (opção essa representada pelo parâmetro *type* da anotação). É possível a escolha tanto da estratégia baseada em classe, em que a classe implementando as instâncias que estão sendo comparadas faz parte das informações utilizadas para a verificação de igualdade, quanto a estratégia baseada em hierarquia, em que é possível a comparação entre subtipos, mantendo os subcontratos de simetria e transitividade das operações associadas à lógica de verificação de igualdade.

Segundo a especificação Java, é vetado às subclasses a herança de anotações declaradas em interfaces realizada [8] , uma vez que este tipo de uso poderia causar um problema semelhante ao problema da hierarquia de diamante (comentada na seção 2.1.1), notório em linguagens que permitem a herança múltipla. Porém, a solução apresentada realiza uma análise de toda hierarquia de classes e interfaces para a definição da lógica de comparação de igualdade, contornando essa restrição e verificando possíveis conflitos decorrentes da hierarquia de classes e interfaces utilizada.

Anotando operações e atributos

A segunda forma de utilização da anotação *@Comparable* é na anotação de atributos pertecentes a uma classe. Quando um atributo é anotado, a informação armazenada nele automáticamente é utilizada na lógica de verificação de igualdade. Diferentemente da solução proposta por (RAYSIDE et al., 2009), no qual um atributo não faz parte do conjunto de informações importantes para a verificação de igualdade. Além de eliminar a necessidade de uma anotação para remoção dessa informação da lógica de verificação de igualdade, representada pela anotação *@NotKey* na solução apresentada por (RAYSIDE et al., 2009), a não utilização de uma informação como padrão cumpre o "principio do menor espanto"(do inglês *least astonishment principle*) (BLOCH, 2006): caso toda e qualquer informação fosse por padrão utilizada pela lógica de verificação de igualdade, a inclusão de um novo atributo em uma superclasse acarretaria em mudanças na lógica de verificação da igualdade de suas subclasses.

Quando a anotação *@Comparable* é utilizada em atributos é possível a definição da forma da comparação das informações contidas no atributo. Entre as possíveis formas de comparação estão: a comparação por identidade, em que as informações contidas em um atributo são consideradas iguais caso a identidade dos valores forem as mesmas; a comparação por valor, em que as informações são consideradas iguais se a lógica de comparação

[8] Inherited (Java Platform SE 6). Disponível em: ⟨http://docs.oracle.com/javase/6/docs/api/java/lang/annotation/Inherited.html⟩.

de igualdade entre as instância de seus respectivos tipos considerar as instâncias iguais; e a forma customizada, em que o programador fornece uma estratégia de comparação customizada, a qual é utilizada para a determinação da igualdade entre as informações contidas nos atributos. Caso o programador deseje usar as duas primeiras formas, deve-se utilizar o parâmetro *value*. Caso deseje a utilização da terceira forma, deve ser empregado o atributo *strategy*, de maneira análoga à utilização da anotação no contexto de uma classe.

Nesse mesmo contexto de uso, a anotação *@Comparable* disponibiliza alguns parâmetros para que o programador defina o papel do atributo no cálculo da ordem natural de instâncias da classe cuja a lógica de comparação está sendo definida. É possível a customização tanto da ordem de avaliação quanto da direção a ser utilizada na comparação entre valores. A ordem de avaliação é customizada através do parâmetro *evaluationOrder*, responsável pela definição da ordem de avaliação do atributo no calculo de ordem natural. Atributos com ordem de avaliação menor têm um grau de importância maior no cálculo de ordem natural.

Com o intuito de exemplificar o conceito de ordem de avaliação, considere-se a classe *PontoColorido*, utilizada em exemplos anteriores. Caso a ordem de avaliação do atributo cor seja menor que a ordem de avaliação do atributo *coordenada*, pontos com cores consideradas menores serão considerados de valor inferior a pontos com cores consideradas maiores. Os valores do atributo *coordenada* só serão utilizados em situações em que as cores de ambos operandos sejam consideradas equivalentes.

A definição da ordem de avaliação de um atributo é opcional. Caso ela não seja definida, a solução definirá a ordem de avaliação desses atributos de uma maneira determinística (no protótipo implementado, descrito no capítulo 6, é utilizada a ordem de declaração dos atributos no código-fonte da classe).

Já para a customização da direção de avaliação, é utilizado o atributo *direction*. Essa direção pode ter dois valores: *Natural*, em que a ordem natural é dada da maneira como definida nos tipos dos valores armazenados no atributo, ou *Inversa*, onde a ordem natural dos valores contidos no atributo será utilizada de maneira invertida. Em outras palavras, caso uma instância seja considerada menor que outra, na ordem de invertida ela será considerada maior.

Outra possibilidade de utilização é o uso da anotação *@Comparable* em uma operação, pertencente tanto a uma classe ou a uma interface. Essa forma de utilização é necessária quando o valor empregadona lógica de comparação é um valor derivado, ou quando deseja-se utilizar uma operação de acesso (ou *getter*) para a obtenção do valor de um atributo. Quando utilizada no contexto de uma operação, a semântica da anotação *@Comparable* é idêntica à utilização em um atributo, porém, ao invés de ser utilizada a informação armazenada em um atributo na lógica de comparação, será utilizado o retorno da operação

anotada.

5.3.1.2 A interface *ComparationStrategy*

Faz parte da API da solução aqui proposta interface *ComparationStrategy*, que agrega todas as operações relativas a verificação de igualdade (e cálculo de ordem natural), seguindo o principio da segregação de interface (ISP) (MARTIN, 2000). Essa interface pode ser utilizada pelos desenvolvedores para a implementação de lógicas customizadas de verificação de igualdade, caso a expressividade obtida pelo uso das anotações *@Comparable* não seja suficiente ou cujo código-fonte não esteja disponível.

Conforme o diagrama de perfil da figura 28, a interface *ComparationStrategy* declara três operações encapsulando toda a lógica de verificação da igualdade entre os operandos. Essas operações carregam a mesma semântica das operações homônimas, definidas anteriormente (vide capítulo 3). Vale salientar que a interface *ComparationStrategy* estende a interface *Comparator*, utilizada pelo *framework* de coleções da linguagem Java para encapsular a lógica de cálculo da ordem natural. Essa característica permite a utilização em coleções ou em classes que permitam o encapsulamento da ordem natural em objetos desse tipo.

Também é possível traçar um paralelo entre as instâncias que realizam a interface *ComparationStrategy* e o conceito de multi-métodos definido na linguagem CLOS (BRUCE et al., 1995), em que os métodos são definidos como funções múltiplas. Ao contrário da implementação em CLOS, os métodos aqui implementados não têm acesso privilegiado a elementos privados declarados pelos operandos. Isso pode impossibilitar a definição da lógica de verificação da igualdade caso as informações necessárias para a comparação não estejam disponíveis na interface pública dos operandos. Nesse caso, é possível a utilização de técnicas, como a definida em (PIERCE; TURNER et al., 1993) que permite o acesso ao estado interno sem a quebra do encapsulamento, ou, na pior das hipóteses, o uso do mecanismo de introspecção incluso na linguagem Java para o acesso sem restrições aos elementos privados dos operandos.

Além disso, devem ser tomadas precauções com relação ao cumprimento de contrato do igualdade (conforme definido na seção 5.2), uma vez que os métodos utilizados pela solução aqui proposta para a manutenção deste contrato têm como premissa que as operações de verificação da igualdade cumpram integralmente o contrato (conforme disposto na seção 5.2.2).

Na seção abaixo, encontram-se exemplos de utilização tanto da anotação *@Comparable* quanto do uso de estratégias de verificação de igualdade customizadas, implementadas através da interface *ComparationStrategy*

5.3.2 Exemplos de Uso

No intuito de facilitar o entendimento das possíveis formas de utilização da solução proposta, serão apresentados alguns exemplos de utilização, em cenários casos de uso. Os exemplos são compostos de trechos de código-fonte, bem como uma explicação detalhada da semântica de comparação exercida em cada um deles :

1º Exemplo

Listagem 15 – 1º Exemplo de Utilização - Solução Proposta

```
@Comparable
public class Pessoa {

  @Comparable
  private CPF cpf;
  private String nome;
}
```

Na listagem 15, define-se uma classe chamada *Pessoa*, contendo dois atributos representando o CPF e o nome. A lógica de comparação entre instâncias de *Pessoa* dita que, para que duas instâncias distintas sejam consideradas equivalentes, as duas devem ter o mesmo valor de CPF. Conforme previamente definido, a não utilização da anotação *@Comparable* na declaração do atributo *nome* faz com que o valor associado a ele não seja levado em conta na verificação de igualdade gerada pela solução.

2º Exemplo

Listagem 16 – 2º Exemplo de Utilização - Solução Proposta

```
@Comparable
public class Pessoa {

  @Comparable(strategy=IgnoreCaseStringComparationStrategy.class,
              evaluationOrder=1)
  private String nome;

  @Comparable(direction=INVERSE,
              evaluationOrder=2)
  private Integer idade;
}
```

Na listagem 16, define-se uma classe chamada *Pessoa* que contém dois atributos: o atributo *nome*, representando o nome próprio dessa pessoa (por exemplo, João) e a *idade* dessa pessoa, em anos. Em relação ao atributo *nome*, pode-se notar, através da utilização do parâmetro *strategy* da anotação *@Comparable*, o uso de uma estratégia customizada para a comparação de valores. Já quanto aos valores associados ao atributo *idade*, deve

ser utilizada uma ordem inversa à ordem de comparação natural (do maior para o menor), conforme indicado pelo parâmetro *direction* da anotação *@Comparable*.

Como previamente definido, o desenvolvedor pode definir a ordem de avaliação das informações pelo desenvolvedor no cálculo da ordem natural. No exemplo, o nome tem um peso maior que a idade (conforme mostrado pelo parâmetro *evaluationOrder* em ambas anotações). Por exemplo, caso se tenha três objetos x, y e z da classe *Pessoa*, respectivamente, com os atributos $('Joao', 25)$ e $('Joao', 26)$ e $('Jose', 23)$, a ordem natural seria y, x e z.

3º Exemplo

Listagem 17 – 3º Exemplo de Utilização - Solução Proposta

```
@Comparable(strategy=SortingStrategyComparisionStrategy)
public interface SortingStrategy {

  @Comparable
  boolean isStable();
  <Type extends Comparable> public List <Type> sort(List <Type> list);
}

public class BubbleSort implements SortingStrategy {

  public boolean isStable() {
    return true;
  }
  // ...
}

public class QuickSort implements SortingStrategy {

  public boolean isStable() {
    return false;
  }
  // ...
}

public class MergeSort implements SortingStrategy {

  public boolean isStable() {
    return true;
  }
  // ...
}
```

A listagem 17 define interface chamada *SortingStrategy*, encapsulando um algoritmo de ordenação genérico, nos moldes do padrão de projeto Strategy (GAMMA et al., 1994, p. 349). A interface *SortingStrategy* declara duas operações: a operação de comparação *sort()*, responsável pela lógica de ordenação do algoritmo, e a operação *isStable()*, cujo

resultado indica se o algoritmo de ordenação encapsulado é considerável estável[9] ou não.

Conforme indicado pelas anotações em *SortingStrategy*, é definida pelo desenvolvedor uma lógica de verificação da igualdade customizada para instâncias de *SortingStrategy* (encapsulada na classe *SortingStrategyComparisionStrategy*), baseada na estabilidade do algoritmo de ordenação encapsulado e sua classe de implementação (o código da classe *SortingStrategyComparisionStrategy* foi omitido, mas pode-se considerar que dadas duas estratégias de ordenação, elas são consideradas equivalentes se ambas forem estáveis, ou se ambas forem implementadas pela mesma classe).

Além da interface que define estratégias de ordenação, foram definidos no exemplo acima três classes, encapsulando algoritmos conhecidos de ordenação: o Quicksort, considerado um algoritmo instável, o BubbleSort e o MergeSort, ambos considerados algoritmos estáveis (KNUTH, 1998). Seguindo a lógica definida de comparação customizada de estratégias de ordenação, instâncias do algoritmo Quicksort só serão consideradas equivalentes à outras instâncias do mesmo algoritmo. Já instâncias de BubbleSort e MergeSort são consideradas iguais, uma vez que ambos algoritmos são algoritmos de ordenação estáveis.

4º Exemplo

Listagem 18 – 4º Exemplo de Utilização - Solução Proposta

```
@Comparable(type=CLASS)
public abstract class Pessoa {

  @Comparable
  private Long id;
}

@Comparable
public class PessoaFisica extends Pessoa {
}

@Comparable
public class PessoaJuridica extends Pessoa {
}
```

Na listagem 18 é representada uma hierarquia de classe, cuja superclasse é a classes abstrata *Pessoa*, que define o atributo *id* utilizado pela lógica de verificação da igualdade. Além da classe *Pessoa*, são definidas duas classes: *PessoaFisica* e *PessoaJuridica*, ambas sendo subclasses de *Pessoa*.

Foi utilizado no exemplo a estratégia de definição de igualdade baseada em classe (representada pelo parâmetro *type* da anotação *@Comparable* anotando *Pessoa*), em que só serão considerados equivalentes operandos que são instâncias de uma mesma classe. Por exemplo, dadas duas instâncias (x e y) com o mesmo valor associado a *id*, x e y

[9] Segundo (KNUTH, 1998, p. 5), um algoritmo de ordenação é considerado estável caso registros com chaves iguais mantenham sua ordem relativa original.

só serão consideradas equivalentes se ambas forem instâncias de *PessoaFisica*, ou seja instâncias de *PessoaJuridica*.

5º Exemplo

Listagem 19 – 5º Exemplo de Utilização - Solução Proposta

```
@Comparable
public abstract class Pessoa {

  @Comparable
  private Long id;
}

@Comparable
public class PessoaFisica extends Pessoa {
}

@Comparable
public class PessoaJuridica extends Pessoa {
}
```

O exemplo disposto na listagem 19 é baseado no quarto exemplo (listagem 18), que representa uma hierarquia de classes, na qual existem duas classes concretas (*PessoaFisica* e *PessoaJuridica*), que por sua vez herdam de uma classe abstrata (*Pessoa*). Da mesma maneira, é utilizado o valor do atributo *id* para a verificação da igualdade entre instâncias de Pessoa.

A diferença entre esse exemplo atual e o anterior reside na utilização da estratégia de verificação. No primeiro exemplo é utilizada a estratégia baseada em classe, em que dois objetos, além de terem valores de atributos e propriedades equivalentes, devem ser implementadas por uma mesma classe para que sejam considerados iguais. Já no segundo exemplo é utilizada a estratégia baseada em hierarquia, na qual os operandos são considerados equivalentes se os valores utilizados pela lógica de verificação da igualdade forem iguais. Logo, nesse exemplo, basta que duas instâncias de *Pessoa* tenham o mesmo valor associado ao atributo *id* para que sejam consideradas equivalentes, independentemente de suas classes de implementação.

6º Exemplo

Listagem 20 – 6º Exemplo de Utilização - Solução Proposta

```
@Comparable
public class Ponto {

  @Comparable
  private Coordenada coordenada;
}

public class PontoColorido extends Ponto {
```

```
    @Comparable
    private Color cor;
  }
```

A listagem 20 apresenta um trecho de código referente ao exemplo dado na seção 5.2.1.2, no intuito de demonstrar o cumprimento do subcontrato da simetria. Conforme previamente descrito, nesse exemplo é definida uma hierarquia de classes, composta pelas classes *Ponto* (mais genérica) e *PontoColorido* (mais especializada). Na classe *Ponto* é definido um atributo chamado *coordenada*, utilizado na lógica de comparação, já na classe *PontoColorido* é definido o atributo *cor*, que também é importante para a verificação da igualdade.

6 Implementação do Protótipo

Neste capítulo serão discutidos alguns aspectos críticos encontrados na implementação do protótipo da solução proposta para a comparação de objetos. Pode-se dividir essa discussão em três grandes aspectos; estrutura do protótipo, peculiaridades da implementação e as mudanças no *workflow* de desenvolvimento.

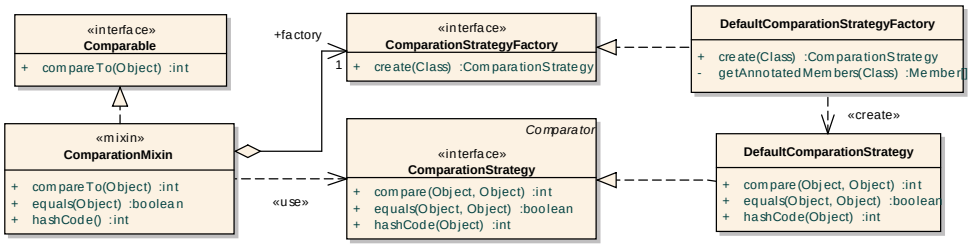
No que tange aos aspectos estruturais, serão detalhadas questões do projeto (*design*) do protótipo, tendo como base os padrões de projetos utilizados e os motivos que levaram a sua utilização. Já na seção abrangendo as peculiaridades da implementação, serão discutidas questões sobre alguns algoritmos utilizados pelo protótipo, bem como otimizações adotadas. Já na ultima seção, cujo foco é o *workflow* de desenvolvimento, são levantadas questões sobre a integração do protótipo com ferramental de desenvolvimento comumente utilizado.

Vale lembrar que o protótipo aqui apresentado é apenas uma forma de implementação da solução apresentada no capítulo 5. Outra forma de implementação, é a substituição do uso da orientação a aspectos pela adoção de protocolos de metaobjetos (MOP), como adotado por Kiczales na implementação do CLOS (KICZALES, 1991). Outra estratégia é implementá-la como uma extensão da linguagem Java, nos moldes adotado pelo Project Coin[1] , que adicionou diversos novos recursos, como por exemplo, a construção *try-with-resources*, que a partir da versão 7, permitiu o uso do idioma RAII na linguagem Java.

6.1 Estrutura do Protótipo

A estrutura adotada pelo protótipo da solução e seus elementos constituintes, criada ten base os princípios definidos pelo SOLID (MARTIN, 2003, cap. 7-12). A figura 29 apresenta um diagrama UML dos componentes da estrutura interna do protótipo da solução implementado.

Figura 29 – Diagrama de Classe - Estrutura Básica



[1] Project Coin. Disponível em: ⟨http://openjdk.java.net/projects/coin/⟩.

Conforme expresso no diagrama, o ponto de acesso da solução proposta ao restante do código percentence a um programa é feito através do *mixin ComparationMixin*, que é introduzido em todas as classes anotadas. O *mixin* de comparação, por sua vez, utiliza a fábrica de estratégias de comparação, cuja responsabilidade é analisar os operandos e construir estratégias semanticamente compatíveis com a semântica expressa pelo desenvolvedor.

6.1.1 Mixin de Comparação

Conforme visto no diagrama de classe da figura 29, a interação da solução com código cliente se dá através do *mixin* de comparação, que deve ser incorporado às classes cuja lógica de verificação de igualdade foram definidas através do uso da anotação *@Comparable*.

O *mixin* de comparação declara três operações (*equals()*, *hashCode()* e *compareTo()*) pertencentes ao mecanismo de verificação da igualdade da linguagem Java. Além dessas operações, o *mixin* também realiza a interface *Comparable*, que habilita a inserção de itens em coleções ordenadas. Há também uma associação entre o *mixin* de comparação e a fábrica de estratégias (descrita com detalhes na seção 6.1.2), utilizada para a criação de estratégias que posteriormente serão utilizadas para a verificação da igualdade, através de delegação.

Caso fosse possível, seria necessária apenas a incorporação (através, por exemplo, de um mecanismo de herança múltipla) desse *mixins* à classes para que estas passassem a incorporar a lógica de verificação da igualdade. Porém, conforme visto anteriormente, a linguagem Java não tem suporte nativo a *mixins*. No entanto, é possível a simulação desse recurso através de um conceito pertencente à programação orientada a aspectos – o conceito de Declarações de Intertipos (ITD).

Conforme visto na seção 2.1.3, é possível, com os recursos fornecidos pelo ITD, a incorporação de novos métodos e operações às classes, bem como a realização de novas interfaces. Com a incorporação tanto dos métodos declarados pelo *mixin* como a realização da interface *Comparable*, tem-se, efetivamente, a incorporação do *mixin* nas classes anotadas.

Para a utilização do recurso de ITD, foi definido um aspecto, utilizando-se o AspectJ, [2] na implementação do protótipo, responsável pela incorporação do *mixin* de comparação nas classes anotadas. Na definição desse aspecto foi definido um ponto de corte que, na presença da anotação *@Comparable* na declaração de uma classe, incorpora nela o *mixin* de comparação. Após a incorporação do *mixin* em uma classe anotadas, esta classe declara todos os métodos de verificação da igualdade. De maneira análoga ao *mixin*, as

[2] The AspectJ Project. Disponível em: ⟨http://www.eclipse.org/aspectj/⟩.

classes modificadas passam a realizar a interface *Comparable*, habilitando a inclusão de suas instâncias em coleções ordenadas.

6.1.2 Fábrica de Estratégias de Comparação

Como descrito anteriormente, o processamento da construção da estratégia de verificação de igualdade é delegado pelo *mixin* a uma fábrica, responsável por examinar os operandos da operação e construir uma estratégia de comparação semanticamente coerente com a lógica declarada pelo desenvolvedor, utilizando-se dos mecanismos de verificação descritos com detalhes no capítulo 5.

Conforme disposto no diagrama de classes representado na figura 29, as estratégias de comparação retornadas pela fábrica apenas devem realizar a interface *ComparationStrategy*, cujo contrato é descrito com detalhes na seção 5.3.1.2. A adoção de abstrações representando tanto as fábricas quanto as estratégias por elas criadas permite a implementação do padrão de fábrica abstrata (GAMMA et al., 1994, p. 99).

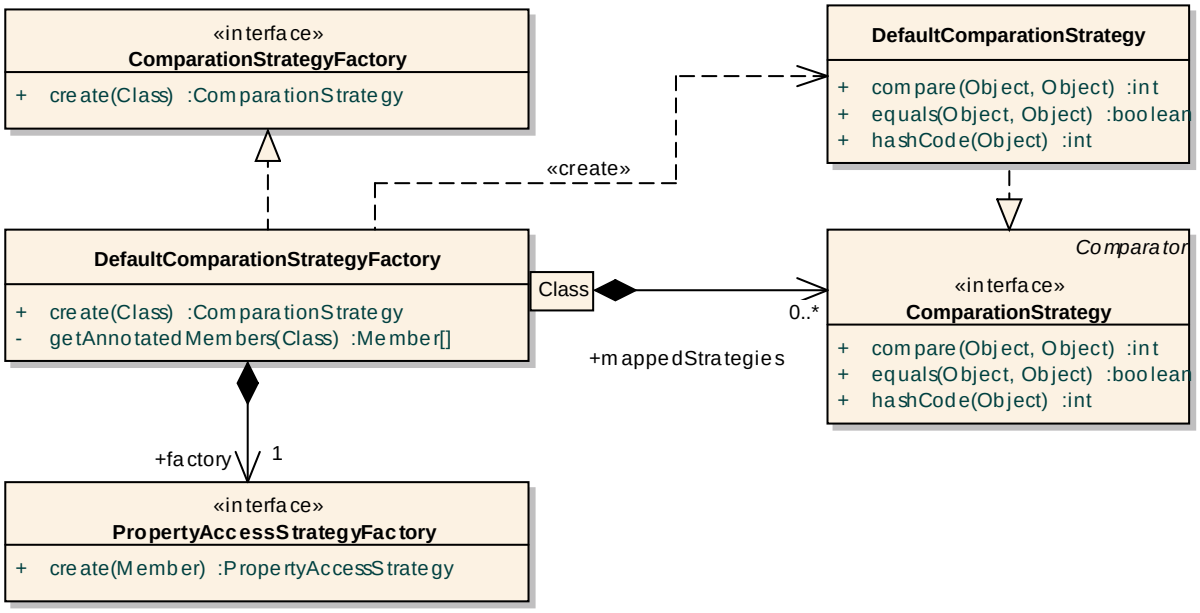
O descoplamento entre o código responsável pela criação das estratégias e o código que os utiliza, bem como definição de interfaces com responsabilidades únicas e bem definidas, permite a extensão da solução. Essas duas interfaces (*ComparationStrategy* e *ComparationStrategyFactory*) compõe a Interface de Prestação de Serviço (SPI) do protótipo, formada por um conjunto de interfaces estáveis que pode ser utilizada na extensão da solução de maneira controlada, seguindo o princípio aberto-fechado.

Nas seções 6.2.4.1 e 6.2.2.1, que descrevem, respectivamente, o mecanismo de *caching* de estratégias criadas e o mecanismo de tratamento de referências cíclicas, é exemplificado como a SPI do protótipo, aliada a padrões como o Decorador (GAMMA et al., 1994, p. 196), pode ser utilizada para a incorporação de novas funcionalidades à solução.

6.1.2.1 Implementação padrão

No desenvolvimento do piloto foi implementada uma fábrica padrão, preparada para a criação de estratégias de comparação que encapsulam todos os aspectos semânticos que podem ser expressos através da anotação *@Comparable*, que, ao mesmo tempo, cumprem o contrato que rege as operações de verificação de igualdade de maneira integral. Para que isso fosse possível, foi necessária a associação entre a implementação e diversos elementos, responsáveis por encapsular funcionalidades necessárias à construção das estratégias de comparação. A figura 30 apresenta um diagrama de classes expondo as principais dependências e associações relativas à implementação padrão da fábrica.

Figura 30 – Diagrama de Classe - Fábrica de estratégias de comparação (Implementação padrão)



Tendo em vista os elementos dispostos no diagrama de classes da figura, serão descritos nas seções subsequentes o papel de cada um deles na implementação de fábrica de estratégias padrão do protótipo implementado.

Escopos de estratégias de comparação customizadas

Outra característica que pode ser observada no diagrama de classes da figura 30 é a associação entre a fábrica de estratégias e um conjunto de estratégias de comparação, devidamente mapeadas por uma classe.

A criação dessa estrutura permite uma solução mais eficiente do problema da definição de uma nova lógica de verificação da igualdade para uma classe cuja lógica de original não possa ser mudada, seja pela indisponibilidade do código-fonte ou por motivos de compatibilidade. Apesar de ser possível o mapeamento da estratégia de comparação customizado, através do uso do parâmetro *strategy* pertencente à anotação *@Comparable*, o uso deste mecanismo é deveras ineficiente quando é necessária a alteração da lógica de verificação da igualdade em todas as comparações de instâncias de uma classe. Nesse cenário, seria necessária a declaração da estratégia personalizada em todas as declarações de propriedades importantes para a lógica de comparação.

No intuito de facilitar a utilização da solução nesse cenário, foi incorporado ao protótipo um escopo global de estratégias customizadas. Com sua incorporação, é criado um registro global de estratégias, em que estratégias de comparação customizadas são associadas

às classes utilizadas na verificação da igualdade. Esse registro é utilizado pelo processo de construção de novas estratégias da seguinte maneira: caso um dado referente a um campo do registro de comparação seja instância de alguma classe previamente registrada, será utilizada pela solução a estratégia associada, em vez dos métodos que porventura estejam definidos[3].

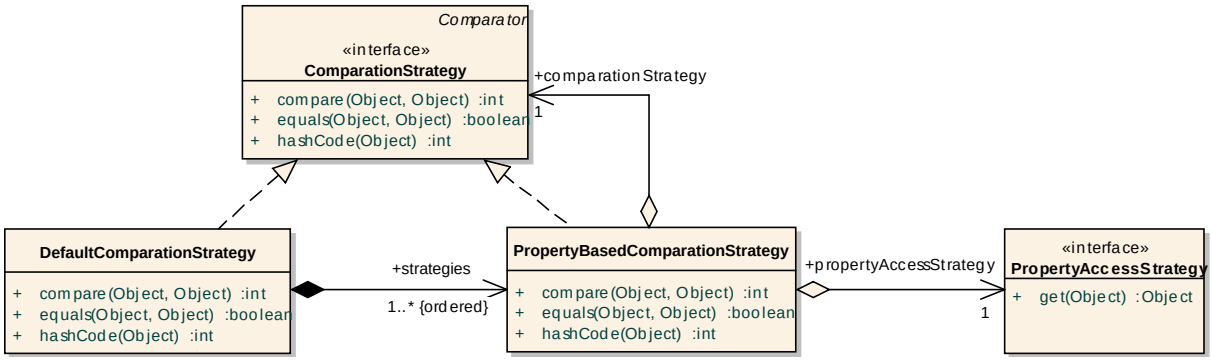
Com a adição desse mecanismo de registro global, definem-se três escopos para o registro de estratégias de comparação customizadas: propriedade (quando a estratégia deve ser utilizada apenas na comparação entre instâncias associadas a uma determinada propriedade), tipo e global (utilizadas quando a estratégia deve ser utilizada para a comparação entre instâncias de um determinado tipo).

A principal diferença entre os escopos de utilização por tipo e global é que, no escopo de utilização por tipo, ainda é necessária a alteração do código-fonte da classe que define o tipo (conforme previamente citado, deve ser utilizado quando a lógica de comparação entre objetos é tão exótica que os mecanismos usuais de definição não são expressivos o suficiente). Já o escopo de utilização global é empregado quando não se dispõe do código-fonte da classe cuja a lógica de comparação se deseja definir.

Estratégias de verificação padrão

Pode-se notar no diagrama na figura 30 a relação de criação entre a implementação padrão de fábrica e a classe *DefaultComparationStrategy*, responsável por encapsular o processo de criação e checagem dos registros de comparação, conforme definido no capítulo 5. No diagrama de classes da figura 31 estão definidas as características estruturais da classe *DefaultComparationStrategy*, bem como suas dependências e elementos com os quais ela interage.

Figura 31 – Diagrama de Classe - Estratégias de comparação (Implementação padrão)



[3] Vale salientar que esse comportamento se dará apenas em comparações no contexto de uma classe cuja lógica de comparação foi definida utilizando os elementos da solução

Examinando o diagrama, pode-se notar a presença de um elemento importante, utilizado para a composição da implementação de estratégia padrão propriamente dita : a estratégia de comparação baseada em propriedades, representada pela classe *PropertyBasedComparationStrategy*.

Essa implementação encapsula a verificação da igualdade no contexto de uma única propriedade. Para isso, a implementação faz uso dos serviços disponibilizados por dois outros objetos: a estratégia de acesso a propriedades, responsável pela extração do valor de uma propriedade específica de dos operandos fornecidos, e uma estratégia de comparação, responsável pela verificação da igualdade dos valores de propriedade obtidos. Em outras palavras, essa implementação age como um Mediador (GAMMA et al., 1994, p. 305), encapsulando a iteração necessária para a verificação dos valores associados às propriedades de operandos.

Ao se analisar a associação entre as classes *DefaultComparationStrategy* e *PropertyBasedComparationStrategy*, pode-se notar que a primeira é composta por uma série ordenada de instâncias pertecentes à segunda. A ordem de cada estratégia nessa série é dada pela ordem de avaliação (definidas através do parâmetro *evaluationOrder*, conforme seção 5.3) das propriedades utilizadas para sua construção.

6.1.3 Fábrica de Estratégias de Acesso a Propriedades

Os últimos elementos dispostos no diagrama da figura 30 são a estratégia de acesso a propriedades e a sua respectiva fábrica, que por sua vez são utilizadas pela fábrica de estratégia de comparação e pelas estratégias de comparação baseadas em propriedades, na construção de novas estratégias e na obtenção de valores associados às propriedades.

A finalidade básica da estratégia de acesso a propriedades (representada pela interface *PropertyAccessStrategy*) é a abstração do mecanismo de obtenção de valores associado a propriedades[4]. Para a obtenção desses valores, é declarada a operação *get()*, que tem como resultado o valor associado à propriedade do objeto fornecido como parâmetro.

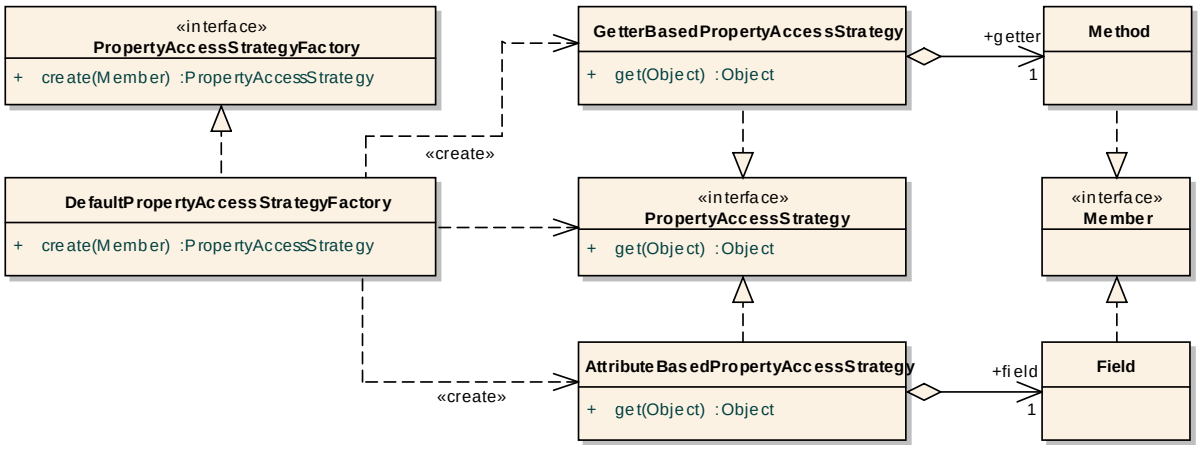
Para a construção dessas estratégias, é utilizada uma fábrica específica, que, de mesma maneira que a fábrica de estratégias de comparação, adota o padrão de projetos Fábrica Abstrata (GAMMA et al., 1994, p. 99), isolando o processo de criação de estratégias do processo de uso, e permitindo o intercâmbio de implementações de estratégias deste tipo. Para a construção dessas estratégias, é fornecido à fábrica uma instância da classe *Member*, definida pelo mecanismo de reflexão da linguagem Java. Essas instâncias (que podem representar tanto métodos quanto atributos) são então utilizadas pela fábrica para a instanciação das estratégias de acesso de forma adequada.

[4] Na linguagem Java, uma propriedade pode ser representada tanto por um atributo, como por um método de acesso.

6.1.3.1 Implementação padrão

A implementação padrão de mecanismos para acesso a propriedades implementado no protótipo são baseadas no mecanismo de reflexão, disponível na linguagem Java. Com o intuito de abranger todos os cenários de uso permitidos para a anotação *@Comparable*, foi adotada a estrutura descrita no diagrama de classe da figura 32.

Figura 32 – Diagrama de Classe - Estratégias de acesso a propriedades (Implementação padrão)



Examinando o diagrama, pode-se identificar duas implementações distintas de *PropertyAccessStrategy*: a *GetterBasedPropertyAccessStrategy* e a *AttributeBasedPropertyAccessStrategy*.

A primeira estratégia concreta (*GetterBasedPropertyAccessStrategy*) utiliza-se da invocação de um método, através de reflexão, para a obtenção do valor de uma propriedade de um objeto. Essa estratégia é indicada para a obtenção de propriedades derivadas (cujos valores são obtidos por um cálculo encapsulado em um método) ou propriedades às quais existem métodos de acesso associados.

A segunda estratégia implementada é representada pela classe *AttributeBasedPropertyAccessStrategy*, encapsulando o mecanismo de obtenção de propriedades armazenadas em atributos de uma classe. Como a estratégia anterior, são utilizados na sua implementação recursos disponíveis no mecanismo de reflexão da linguagem Java.

Por último, pode-se identificar a implementação da fábrica propriamente dita, que, em função da instância da classe *Member* fornecida, escolhe a estratégia concreta mais adequada. Para fazer essa escolha, são utilizados :

a) O tipo da instância de *Member* fornecida;

b) Caso a instância fornecida seja uma instância de *Field*, verifica-se a presença de

métodos, seguindo a nomenclatura JavaBean[5] .

A utilização do mecanismo de reflexão

Conforme dito previamente, a implementação padrão de estratégias de acesso a propriedades é baseada no mecanismo de reflexão. Essa decisão acarreta algumas vantagens e desvantagens. A principal desvantagem reside no *overhead* intrínseco ao uso de operações reflexivas, conforme apontado por Bloch :

> "Invocações de métodos via reflexão são muito mais lentas que a invocação tradicional. É difícil precisar o quão mais lenta, pois existem muitos fatores envolvidos. Na minha máquina, a diferença de velocidade pode ser da ordem de dois, no melhor caso, ou da ordem de cinquenta, no pior caso."(BLOCH, 2008, p. 230, tradução do autor)

Apesar da desvantagem no quesito de desempenho, a utilização do mecanismo de reflexão tem uma grande vantagem; É possível, através do seu uso, o acesso (desde que permitido pelo gerenciador de segurança da plataforma Java) à elementos privados de uma classe. Com isso é possível mitigar o problema levantado por (BRUCE et al., 1995), que coloca a necessidade de acesso privilegiado quando métodos binários são simulados. Caso fosse adotado outro mecanismo, como por exemplo a criação em tempo de execução de novas classes encapsulando o acesso aos atributos ou a invocação a métodos, seria necessária a adoção de técnica complementares para habilitar ao acesso a esses elementos caso os mesmos fossem privados, como por exemplo a técnica proposta por (PIERCE; TURNER et al., 1993) ou a geração de automática de métodos de acesso, conforme utilizado por (GRECH; RATHKE; FISCHER, 2010).

6.2 Características da implementação

Nesta seção serão tratados itens relativos a detalhes de algumas decisões de implementação do protótipo da solução. Ao contrário da seção anterior, que abordou questões estruturais, descrevem-se as decisões relativas aos algoritmos e otimizações identificadas.

Também serão abordadas questões relativas à robustez e extensibilidade da solução, de maneira análoga aos critérios adotados para a análise das soluções previamente apresentadas.

[5] JavaBeans Spec. Disponível em: ⟨http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html⟩.

6.2.1 Mecanismo de Simulação de Despacho Múltiplo

Conforme exemplificado nas seções 5.2.1.2 e 5.2.1.3 (Simetria e Transitividade), é impossível a manutenção dos subcontrato de simetria e transitividade em linguagens de despacho simples (como a linguagem Java), quando existe a relação de herança entre os operandos da operação. Em cenários como esse, é necessário o suporte a um mecanismo de despacho múltiplo, em que o método escolhido para a execução de uma operação é escolhido através de todos operandos da operação e não somente de um único (BRUCE et al., 1995).

De acordo com (BRUCE et al., 1995), existem diversos mecanismos que podem ser utilizados em linguagens baseadas em despacho simples para a simulação de um mecanismo de despacho múltiplo. Uma solução apresentada é a solução de Ingalls (INGALLS, 1986), previamente apresentada na seção 2.1.1. Conforme descrito, a solução de Ingalls tem dois problemas: a quebra da modularidade, uma vez que é necessário que se conheça de antemão todos as classes estendendo a classe de um parâmetro, e o tamanho de casos distintos que devem ser implementados quando a hierarquia de classes é grande.

A solução aqui apresentada utiliza-se de um mecanismo customizado para a simulação de despacho múltiplo: no momento da invocação de uma operação de verificação da igualdade, é construída pela fábrica de estratégias uma estratégia encapsulando a lógica de comparação correspondente com acesso privilegiado às informações armazenadas nos operandos, sejam elas privadas ou não. O processamento da operação é então delegado para essa estratégia, construída através da introspecção de uma classe fornecida como argumento.

Para a seleção da classe que deve utilizada, é examinado o nível de hierarquia das classes cujo operandos fornecidos são instâncias. Será utilizada para a construção da estratégia a classe com o nível de hierarquia mais baixo (mais especializada) entre as classes dos operandos fornecidos, partindo do pressuposto que esta classe terá todos os atributos da classe mais genérica, mais os atributos que ela porventura declarar.

Utilizando a hierarquia de classes dos exemplos anteriores (baseadas nas classes *Ponto* e *PontoColorido*), caso ambos operandos sejam do tipo *Ponto*, o nível hierárquico das classes implementando os operandos é o mesmo e os registros de comparação serão construídos com base nos atributos definidos na classe *Ponto*. Já no caso de um dos operandos ser uma instância da classe *PontoColorido*, ambos registros de comparação serão construídos com base nas informações associadas à classe *PontoColorido*, pois ela tem o nível de hierarquia mais baixo do que a classe *Ponto*.

Em virtude de a lógica de seleção da classe para a construção da estratégia ser dependente do nível hierárquico da classe dos operandos e não da sua ordem, pode-se concluir que tanto a operação $x.equals(y)$ quanto $y.equals(x)$ são processados pela mesma es-

tratégia, simulando assim um mecanismo de despacho múltiplo.

6.2.2 Robustez

A solução apresentada incorpora diversos mecanismos para aumento da robustez, conforme pontos definidos em soluções previamente apresentadas (VAZIRI et al., 2007; RAYSIDE et al., 2009; GRECH; RATHKE; FISCHER, 2010) e na análise critica delas (RUPAKHETI, 2010). Além do enfoque na corretude da solução é dado um enfoque especial na robustez dos mecanismos de extensão da solução, que aumentam sensivelmente o grau de expressividade da solução, permitindo a utilização mesmo em projetos com lógicas de verificação de igualdade mais exóticas.

Seguem as decisões relativas a robustez tomadas, bem como a base teórica delas e um relato de como essas decisões se relacionam com as decisões tomadas nas soluções previamente apresentadas.

6.2.2.1 Referências Cícliclas

A necessidade da robustez das operações de verificação de igualdade na presença de referências cíclicas em operandos é citada tanto por (RAYSIDE et al., 2009) como por (GRECH; RATHKE; FISCHER, 2010), cujas soluções incorporam mecanismos para o tratamento de referências cíclicas. Essa necessidade é corroborada pelos estudos realizados por (MELTON; TEMPERO, 2007), que conforme anteriormente citado, mostram a presença de referências cíclicas em uma grande gama de softwares atualmente em produção.

A solução aqui proposta também tem incorporados mecanismos para detecção e tratamento de referências cíclicas em operandos de operações de verificação da igualdade entre objetos. Porém, ao contrário da solução apresentada por (GRECH; RATHKE; FISCHER, 2010), que se utiliza, conforme descrito na seção 4.2.4.4, de um aspecto para a inclusão de código de detecção e tratamento do ciclo, a solução proposta utiliza-se de um Decorador (GAMMA et al., 1994, p. 196), incorporado pela fábrica de estratégias de comparação.

A utilização de um Decorador ao invés de um aspecto[6] tem como vantagem a possibilidade de se tratar ciclos em referências de objetos não anotados, mas cuja lógica de comparação é tratada por uma estratégia customizada de comparação.

Segue discussões do mecanismo utilizado para a detecção dessas referencias, bem como da lógica de tratamento destas referências em cada uma das operações associadas à lógica de verificação de igualdade.

[6] É possível o tratamento com aspectos, através da técnica de *load-time weaving*, porém esse tratamento pode ter impactos negativos no desempenho e na segurança.

Detecção e tratamento

O mecanismo de detecção de referências cíclicas utilizado na solução apresentada é bastante simples, e tem como base algumas premissas que devem ser verdadeiras para uma correta detecção e tratamento destes ciclos. Apesar de ser importante para o bom funcionamento do mecanismo de detecção de ciclos a manutenção dessas premissas, outros componentes da solução já realizam o trabalho de manutenção delas, retirando esta responsabilidade do programador.

Premissas de funcionamento

A primeira premissa a ser cumprida refere-se à alteração dos valores de propriedades de objetos envolvidos nas operações de verificação da igualdade. Durante a execução dessas operações, deve ser vetada a *threads* diferentes do *thread* corrente a alteração das propriedades de objetos sendo examinados, em outras palavras, os objetos sendo comparados devem ter como característica serem *thread-safe*.

Já a segunda premissa tem como alvo o comportamento das operações envolvidas na lógica de comparação da igualdade. Essa operações devem ser operações sem nenhum efeito colateral (como por exemplo, a alteração do valor de uma propriedade), ou seja essas operações devem ser operações de consulta. As operações de verificação da igualdade criadas pela solução através do uso das anotações previamente discutidas já têm essa característica, mas deve-se levar em conta essa premissa quando se utiliza dos mecanismos de extensão para a escrita de lógica de comparação mais exóticas.

Mecanismo de detecção

Para a detecção dos ciclos de referência, é criada, no momento da primeira chamada a uma operação pelo mecanismo de detecção uma pilha. Essa pilha funciona de maneira análoga a uma pilha de chamadas de função tendo como o escopo o *thread* de execução corrente.

A cada chamada de operação, o mecanismo de detecção de ciclos inclui nessa pilha uma tupla contendo os operandos[7] sendo utilizados na operação. Caso já exista uma tupla na pilha cujos operandos tenham a mesma identidade, é detectado um ciclo e o mecanismo de detecção de ciclo dispara a lógica de tratamento de ciclo correspondente à operação sendo executada. Caso contrário, continua-se a execução da operação correspondente. Após o fim da execução, a tupla é retirada da pilha e a execução segue seu fluxo normal.

Conforme citado, no momento da detecção de um ciclo, executa-se uma lógica de tratamento de ciclo própria para cada uma das operações relacionadas com o processo

[7] São considerados como operando tanto o objeto que recebe a mensagem quanto os objetos passados como parâmetros

de verificação de igualdade. Segue uma descrição dos métodos de tratamento utilizados, bem como digressões indicando o correto resultado obtido por elas.

Tratamento de ciclos

Quando é detectado um ciclo na operação *equals()*, a solução apresentada ignora o mecanismo normal de verificação da igualdade destes operandos e retorna um elemento neutro (no caso o valor verdadeiro) como resultado da operação. O raciocínio adotado nesse mecanismo baseia-se na lógica booleana e na representação adotada para o cálculo da igualdade. Conforme já visto, a operação de verificação de igualdade consiste na verificação sistemática de todos os itens inclusos nos registros de comparação. Essa verificação pode ser generalizada por $y = x_1 \wedge x_2 \wedge x_3 \wedge \ldots \wedge x_n$.

Seguindo o mesmo conceito, pode-se criar subregistros representando elementos especificos[8] contidos nos registros de comparação. Nesse processo de criação, é possivel, caso a estrutura estática dos objetos permita, a ocorrência de repetições de comparações já sendo executadas. A expressão $y = x_1 \wedge (x_2 \wedge x_3 \wedge x_4) \wedge x_5 \wedge x_2 \wedge \ldots \wedge x_n$ representa uma expressão de verificação de igualdade, em que foi encontrado um ciclo de comparação. Na realidade, expressão descrita é apenas um exemplo teórico. A recursão encontrada faria essa expressão ser infinita.

Segundo a lógica booleana (GERSTING, 2002, cap. 7) como há idempotência do operador $\wedge$ (que dita que $x \wedge x = x$) e da identidade (representado por $x \wedge 1 = x$), pode-se simplificar a expressão substituindo a segunda checagem por verdadeiro sem alteração do resultado final, conforme segue :

$$
\begin{aligned}
y &= x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5 \\
y &= x_1 \wedge \underbrace{(x_4 \wedge x_6 \wedge x_7)}_{\text{Expansão de } x_2} \wedge x_3 \wedge x_4 \wedge x_5 \\
y &= x_1 \wedge (x_4 \wedge x_6 \wedge x_7) \wedge x_3 \wedge 1 \wedge x_5 \\
y &= x_1 \wedge (x_4 \wedge x_6 \wedge x_7) \wedge x_3 \wedge x_5
\end{aligned}
$$

Esse mecanismo de tratamento de referências cíclicas no método *equals()* também é utilizado pela solução proposta por (GRECH; RATHKE; FISCHER, 2010) no tratamento de ciclos :

> "Se o alvo corrente do ponto de corte está presente na pilha, então a execução chegou a um ciclo. Nesse caso, o a checagem de igualdade

[8] No caso da solução proposta, elementos cujo tipo é anotado com a anotação *@Comparable*

> cuja iteração causou um ciclo deve ser considerada verdadeira, pois caso contrário ela terminaria por curto-circuito, retornando falso." (GRECH; RATHKE; FISCHER, 2010, tradução do autor)

De maneira análoga ao tratamento do método *equals()*, o tratamento dos métodos *hashCode()* e *compareTo()* retornam um elemento neutro (zero para ambas operações) quando um ciclo é detectado. Esse tratamento, conforme exemplificado anteriormente, permite que a solução proposta retorne um resultado semanticamente válido quando ciclos são encontrados.

O mecanismo de detecção de referências circulares incluso na solução tem como foco principal a simplicidade de implementação e depuração. Existe uma grande gama de otimizações que podem ser incorporadas ao mecanismo de detecção, desde otimizações das estruturas de dados utilizadas, como a habilitação do mecanismo apenas em invocações cujas características estruturais dos operandos permitam a presença de ciclos, de maneira análoga à solução de (GRECH; RATHKE; FISCHER, 2010).

6.2.2.2 Uso em coleções com restrições de igualdade

Apesar de o problema da utilização de elementos mutáveis em coleções com restrições de igualdade ser um conhecido e ainda persistir em códigos-fonte dos mais variados níveis de complexidade, conforme estudo realizado em (NELSON; PEARCE; NOBLE, 2010), não é proposto pela solução aqui apresentada nenhum modo de mitigação deste problema.

Apesar de os mecanismos propostos tanto por (VAZIRI et al., 2007) como por (GRECH; RATHKE; FISCHER, 2010) poderiam ser facilmente integrado na solução proposta, através de checagens realizadas durante o processo de *weaving* dos aspectos de comparação no código-fonte, foi decidida a não utilização do mecanismo, pois o alto grau de restrição imposto impossibilitaria a adoção da solução proposta em muitos cenários válidos de utilização, conforme exemplos citados na seção 4.2.2.4.

Uma solução menos invasiva para esse problema seria a utilização de técnicas da programação orientada a aspectos, juntamente com um processo de *weaving* em *load-time*[9], para a inserção de nova lógica tratando a mutabilidade de elementos em coleções com restrições de igualdade.

Para a implementação dessa solução seriam necessárias mudanças tanto nos elementos a serem armazenados pelas coleções quanto nas coleções propriamente ditas. Nos elementos é necessária a implementação de Observadores (GAMMA et al., 1994, p. 326), responsáveis pelo disparo de eventos quando propriedades importantes para a lógica de comparação entre objetos são modificadas.

[9] Pode-se utilizar qualquer tipo de agente que intercepte o mecanismo de carregamento de classe da JVM, seguindo moldes definidos nos serviços de instrumentação Java

Já para a alteração das coleções com restrições de igualdade, seria necessária a definição pontos de corte, representando a criação de objetos cujos tipos são coleções com restrições de igualdade. Nesses pontos de corte seriam injetada a lógica responsável por decorar (agregar a um objeto um Decorador) a coleção recém-criada. Os métodos de manipulação desse Decorador efetuam o registro da coleção no mecanismo de observação dos elementos sendo inseridos ou removidos. No momento em que uma alteração nas informações importantes para a comparação entre objetos é executada, o Decorador dispara uma reindexação dos elementos da coleção decorada, mantendo-a íntegra.

Para a adoção desse mecanismo na solução proposta, é necessária uma análise mais aprofundada do impacto do sistema de observadores no desempenho das operações nas coleções, tanto no âmbito do tempo de processamento quanto no uso de memória. Também é necessária uma análise mais aprofundada da maneira mais adequada de tratamento de propriedades derivadas, que possam eventualmente ser utilizadas na definição da lógica de verificação da igualdade entre objetos. Outro ponto a ser considerado é a utilização do mecanismo de *load-time weaving*, que pode ter sua utilização vetada em determinados ambientes produtivos.

6.2.3 Extensibilidade

A solução apresentada, da mesma maneira da que a solução descrita por (GRECH; RATHKE; FISCHER, 2010), provê mecanismos que permitam o seu uso tanto em casos de exceção, em que os mecanismos para a definição da lógica de comparação entre objetos não são suficientemente expressivos para a definição da lógica de comparação desejada, quanto em casos em que o código-fonte da classe pertence a bibliotecas cujos códigos-fonte não estejam disponíveis ou não possam ser alterados (como por exemplo, classes pertencentes ao JDK).

A necessidade de mecanismos permitindo a definição de lógicas de comparação em classes de terceiros identificada por (GRECH; RATHKE; FISCHER, 2010) é corroporada por (RUPAKHETI, 2010, p. 118), na sua análise da solução proposta por (RAYSIDE et al., 2009), na qual foi impossível a utilização da solução para definição da lógica de verificação da igualdade em uma classe em que era necessária a comparação de dois valores em ponto flutuante seguindo a representação em inteiros deles conforme a norma IEEE 754.

Conforme visto na seção 5.3.1.2, quando o desenvolvedor deseja customizar o modo de verificação de igualdade entre objetos, ele deve implementar uma interface responsável pela definição de uma estratégia de comparação. Nessa interface são definidos os métodos para a verificação de igualdade e a comparação da ordem natural entre instâncias de objetos, bem como o cálculo do código *hash* de instâncias de objetos. Implementada essa interface, é possível o registro de suas instâncias através dos mecanismos definidos pela

API ou pela fábrica de estratégias de comparação utilizada. Após a sua implementação, essa regra pode ser utilizado em um dos três escopos : escopo de propriedade, tipo ou global, conforme descritos na seção 6.1.2.1.

A utilização dessa interface bem definida para a definição das lógicas customizadas de comparação e a definição de pontos de extensão bem definidos implementam a boa prática da segregação de interface e do principio aberto-fechado, respectivamente. O princípio aberto-fechado garante que o desenvolvedor não efetue mudança no comportamento esperado dos componentes sendo estendidos. Já o principio da segregação de interface garante que o programador agrupe métodos semanticamente relacionados em um elemento único, garantindo um melhor entendimento do software e um maior grau de coesão.

6.2.4 Otimizações

Além do aspecto da manutenção de contrato de verificação da igualdade e dos aspectos de estruturais da solução apresentada, foram identificadas durante a implementação do protótipo algumas questões de desempenho e possíveis otimizações gerais. Abaixo serão discutidas algumas destas otimizações com mais detalhes.

6.2.4.1 *Caching* de Estratégias de Comparação

Anteriormente fora exposto que a fábrica de criação de estratégias de comparação utiliza-se do mecanismo de introspecção da linguagem Java para a criação das estratégias de comparação, percorrendo, durante o processo de construção da estratégia, toda a hierarquia de classes e interfaces associada à classe implementando um operando.

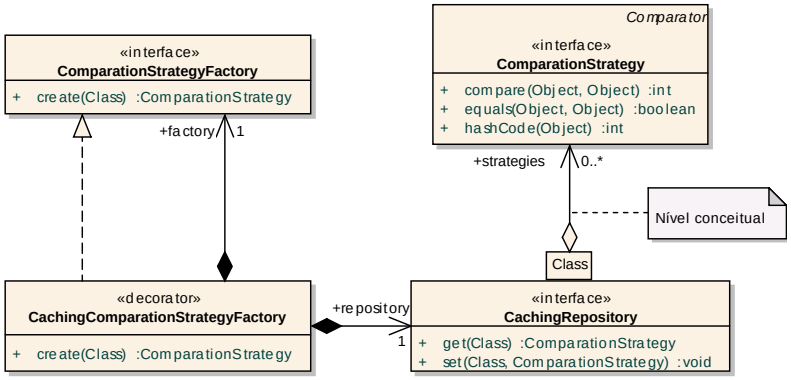
Segundo (TUDOSE; ODUBăşTEANU; RADU, 2013), essa iteração na hierarquia de classes é muito custosa, comparativamente ao custo de processamento associado à criação e cálculo dos registros de comparação. Além disso, a iteração é baseada em elementos imutáveis do programa (definições de classes e interfaces). Logo, não existem alterações na semântica de comparação obtida quando são utilizadas as mesmas classes como parâmetro de construção.

Essa característica faz com que esse processo seja candidato ao uso de um mecanismo de "cacheamento" de resultados, uma vez que esse tipo de estrutura consegue reduzir drasticamente a utilização de recursos computacionais e o tempo de processamento, ao custo de uma maior utilização de memória.

Para a implementação do mecanismo de *caching*, é criado um Decorador (GAMMA et al., 1994, p. 196), realizando a interface *ComparisionStrategyFactory*. Esse Decorador associa à fábrica de estratégias de comparação padrão um repositório, utilizado para o armazenamento das estratégias já criadas. O repositório pode utilizar algoritmos distintos de invalidação, baseados tanto na frequência de uso quanto no tempo decorrido desde a

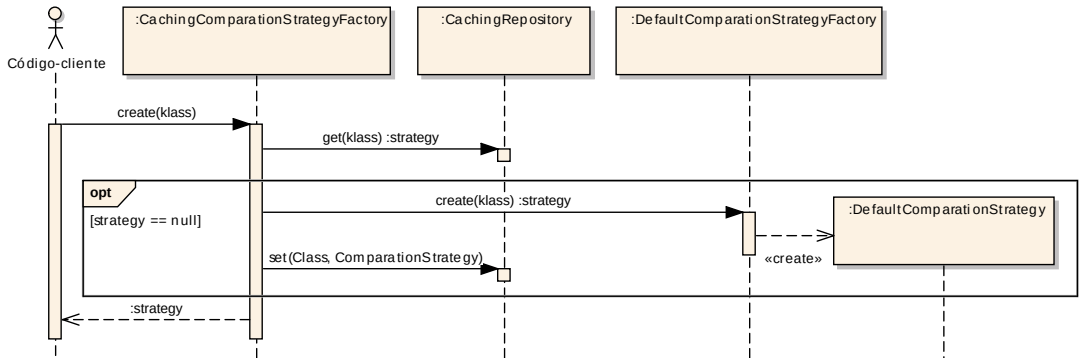
última atualização. A estrutura e as dependências desse Decorador responsável pelo cacheamento de estratégias pode ser expresso pelo diagrama de classes da figura 33.

Figura 33 – Diagrama de Classes - Decorador de *Caching* (protótipo)



Para efetuar o "cacheamento", o Decorador, no momento de uma solicitação de criação de uma estratégia, verifica se já existe no repositório a estratégia criada a partir da classe fornecida. Caso não exista, o processo de criação é delegado para a fábrica decorada e a estratégia recém-criada é armazenada no repositório. Quando forem feitas novas solicitações de construção, tendo como parâmetro a mesma classe fornecida, será retornado o resultado armazenado no repositório. Pode-se representar esse processo através do diagrama de sequência de mensagens da figura 34.

Figura 34 – Diagrama de Sequencia - Decorador de *Caching* (protótipo)



6.2.4.2 Verificação de Igualdade

Conforme os exemplos dados anteriormente, o processo teórico de verificação de igualdade de objetos utilizado pela solução consiste de duas fases distintas; a criação do registro de comparação, em que são criados registros de comparação com base nos operandos da

operação de verificação da igualdade, e a fase de cálculo, em que a solução compara as informações contidas nos registros de comparação, retornando se ambos operandos são considerados equivalentes ou não.

Uma das desvantagens desse algoritmo de verificação é a quantidade de memória necessária para o armazenamento dos registros de comparação recém-criado, inexistente nas implementações convencionais das operações de verificação de igualdade.

Outra desvantagens inexistente nas implementações convencionais é a necessidade de duas iterações para se verificar a igualdade entre dois operandos. Normalmente, o programador utiliza-se dos princípios de avaliação minima[10] para otimizar a lógica de comparação, evitando a avaliação desnecessária de declarações lógicas.

Utilizando-se do mesmo principio da avaliação mínima, é possível otimizar o mecanismo de verificação de igualdade da solução. No algoritmo otimizado, não existem mais duas fases (criação e avaliação de registros de comparação). Na primeira iteração, normalmente responsável pela criação de elementos do registro já são feitas as comparações entre as informações associadas ao elemento, caso as informações não sejam consideradas equivalentes, os operandos já são considerados diferentes e o processo é terminado. Caso não existam mais informações a serem obtidas, os operandos são considerados iguais. A listagem 21 apresenta, em pseudocódigo, o algoritmo descrito.

Listagem 21 – Pseudocódigo - Algoritmo de Verificação Otimizado

```
while existem novos elementos do
   o1 ← valor elemento referente a op1
   o2 ← valor elemento referente a op2
   if o1 ≠ o2 then
      return false;
   end if
end while
return true;
```

Com a otimização acima, não é mais necessário o armazenamento em memória dos registros de comparação criados a partir dos operandos da operação de verificação da igualdade. Também não é mais necessária a verificação de todos as informações para chegar-se a uma decisão sobre a igualdade entre os operandos fornecidos à operação. Para otimizar ainda mais, pode-se utilizar o parâmetro *evaluationOrder*, que controla a ordem de avaliação das informações na lógica de verificação de igualdade, avaliando primeiramente informações com um maior grau de espalhamento.

[10] Também chamada de avaliação de curto-circuito ou avaliação de McCarthy

6.2.4.3 Verificação da Ordem Natural

Da mesma maneira que a verificação da igualdade, o processo teórico do cálculo de ordem natural é composto por duas fases. A primeira fase do processo é constituída pela criação dos registros de comparação, enquanto na segunda fase é executada a verificação do cálculo da ordem natural dos operandos fornecidos.

As mesmas desvantagens encontradas no processo original de verificação de igualdade entre objetos se aplicam ao cálculo da ordem natural. O processo de cálculo teórico tem um uso de memória maior que a utilização de uma solução comumente escrita por um desenvolvedor, que normalmente utiliza-se de princípios de avaliação mínima para otimizar o código, evitando verificações desnecessárias.

Também é possível a utilização do mecanismo de avaliação mínima para a otimização do processo de cálculo da ordem natural da solução aqui apresentada. No algoritmo otimizado é executada apenas uma iteração, baseada na ordem de avaliação definida pelo desenvolvedor. Em vez de serem armazenadas em registros para posterior comparação, essas informações são prontamente comparadas. Caso essas informação sejam consideradas diferentes, o processo é abortado e a ordem natural obtida é retornada. Caso o processo chegue ao final sem que sejam encontrados divergências nas informações comparados, os operandos são considerados equivalentes.

6.2.4.4 Otimização dinâmica da lógica de comparação

Outra possível otimização identificada no estudo do problema e das soluções já apresentada é a solução adotada por (GRECH; RATHKE; FISCHER, 2010) na implementação da sua solução. Nessa solução, descrita em com mais detalhes na seção 4.2.4, são feitas análises do perfil de execução do programa e, a partir desta análise, é alterada a ordem de execução das checagens, no intuito de eliminar checagens desnecessárias.

Essa técnica, apesar de poder ser incorporada à solução aqui proposta, conflita com o mecanismo de cálculo da ordem natural adotado, pois este último necessita de uma ordem de avaliação estável para manter a consistência com a semântica definida pelo desenvolvedor.

Uma possível maneira de se incorporar essa técnica pode ser através da introdução de duas ordens de avaliação. Uma interna, utilizada pela solução proposta nas chamadas ao método *equals()* e outra externa, utilizada nas chamadas a *compareTo()*. A ordem de avaliação interna pode ser dinamicamente alterada, de maneira a aumentar o desempenho evitando checagens desnecessárias. Já ordem de avaliação externa é consistente durante toda a execução do programa.

De maneira análoga, as possíveis soluções para o problema de uso de objetos mutáveis em coleções com restrições de igualdade, expostas na seção 6.2.2.2, são necessárias análises

mais aprofundadas dos possíveis impactos na incorporação dessas otimizações à solução e ganhos obtidos com essa otimização em testes de desempenho com carga de trabalho mais semelhantes ao esperado no mundo real[11].

[11] Como por exemplo, os testes pertencentes ao *benchmark* DaCapo (BLACKBURN et al., 2006).

7 Experimento

No intuito de mensurar tanto a eficácia da solução na definição da lógica de verificação da igualdade quanto os ganhos obtidos através de sua adoção, é descrito neste capítulo um experimento informal, que estuda a adoção da solução aqui apresentada em um *software* já existente.

Como objeto da conversão, foi escolhida a versão da biblioteca JFreeChart[1] (versão 1.0.0) utilizada por desenvolvedores interessados na incorporação de gráficos em programas escritos na linguagem Java. O tamanho dessa biblioteca, (composta de 478 classes, distribuídas em aproximadamente 67.000 linhas de código), aliado à existência de um teste de desempenho específico (BLACKBURN et al., 2006), fazem com que esta biblioteca seja um bom candidato a objeto do experimento. Outro ponto positivo é sua utilização, de maneira integral ou parcial, nos experimentos descritos em (VAZIRI et al., 2007; RAYSIDE et al., 2009; GRECH; RATHKE; FISCHER, 2010).

Pode-se dividir o experimento em duas fases distintas: na primeira fase é executada a conversão do código-fonte da biblioteca JFreeChart propriamente dita, enquanto na segunda fase são realizadas análises comparativas entre o código-fonte alterado e o código-fonte original. Os métodos utilizados tanto para a obtenção das medidas quanto para a sua análise também estão descritos neste capítulo.

7.1 Método

Nesta seção, são discutidas as premissas utilizadas para a realização da conversão do código-fonte para a adoção da solução proposta neste trabalho. Também são aqui descritos os indicadores que serão utilizados para a medição dos ganhos e eventuais perdas na adoção da solução, em relação à versão original da biblioteca.

Para a definição do escopo da tarefa de migração do código-fonte da biblioteca, é definida uma série de premissas que devem ser consideradas no processo de migração:

a) É vetada a modificação do código-fonte de classes cujo código não pertençam ao JFreeChart (classes disponibilizadas na biblioteca JCommon[2] não devem ser modificadas);

b) Para efeito do experimento, a biblioteca será considerada correta caso a suíte de testes de unidade apresente uma taxa de sucesso maior ou igual à taxa da versão original;

[1] JFreeChart. Disponível em: ⟨http://www.jfree.org/jfreechart/⟩.
[2] JCommon. Disponível em: ⟨http://www.jfree.org/jcommon/⟩.

c) As alterações realizadas no código-fonte devem se resumir a:

- Exclusões de métodos;
- Inclusões da anotação *@Comparable*;
- Implementações de estratégias customizadas.

d) Não serão feitas quaisquer tentativas de correção em defeitos que não estejam diretamente relacionados à lógica de verificação da igualdade, que porventura sejam encontrados.

Conforme citado, um dos objetivos do experimento é a identificação das vantagens e possíveis desvantagens da adoção da solução apresentada numa base de código-fonte real. Para isso, serão utilizado três indicadores, compostos de medidas mensuradas tanto no código-fonte original da biblioteca (*baseline*), quanto no código-fonte modificado.

Cabe aqui uma observação: apesar de a inclusão das medidas referentes às soluções apresentadas em (VAZIRI et al., 2007; RAYSIDE et al., 2009; GRECH; RATHKE; FISCHER, 2010) ser interessante, no momento da redação deste trabalho, tanto os protótipos quanto os experimentos neles descritos estavam indisponíveis[3]. Optou-se então por se fazer as comparações contra o código-fonte original, em vez do modificado para uso das soluções previamente apresentadas.

7.1.1 Corretude

Os indicadores de corretude indicam a observância do código-fonte com relação ao processamento esperado (dada uma entrada, a saída obtida deve ser condizente). Ao contrário de métodos formais para a verificação da corretude, como a Lógica de Hoare (HOARE, 1969), serão utilizados, no contexto deste experimento, mecanismos informais para a verificação do nível de observância do programa aos seus requisitos funcionais.

O primeiro mecanismo utilizado para a medição da corretude é composto da suíte de testes de unidade inclusa no pacote de código-fonte do JFreeChart, no qual são realizadas requisições à biblioteca e os resultados obtidos são comparados com valores esperados. Ao fim desse processo é calculada uma taxa de sucesso, indicando a quantidade de verificações bem-sucedidas. Com o uso desse mecanismo, é possível à verificação da consistência entre os resultados obtidos no código-fonte original e os valores calculados para a versão modificada, verificando assim a observância aos requisitos funcionais.

Já o segundo mecanismo usado é composto da análise estática da biblioteca. Nesse tipo de análise, tanto o código-fonte quanto o código-objeto compilado são percorridos a procura de padrões que indiquem defeitos na lógica de programação. Para a realização

[3] Com exceção do protótipo referente à (GRECH; RATHKE; FISCHER, 2010), disponível em http://sourceforge.net/projects/jequalitygen/

dessa análise, será utilizada a ferramenta FindBugs[4] (versão 3.0.0), que permite a categorização das ocorrências em grupos de violação, bem como a filtragem de ocorrências em função de suas categorias e construtos afetados.

Para e execução das análises, será utilizada a opção de esforço máximo (*effort="max"*), que indica à ferramenta que deve ser realizada uma análise de procura de defeitos mais criteriosa. Além disso, no intuito de se minimizar o relato de violações não relacionadas à lógica de verificação da igualdade, a análise será restringida a violações afetando os métodos *equals()*, *hashCode()* e *compareTo()*, ou relativas à categoria *EC*, compreendendo checagens de invocação do método *equals()* com parâmetros inválidos.

Uma questão que deve ser levada em conta na utilização de ferramentas de análise estática é a geração de alarmes falsos (AYEWAH et al., 2008). No intuito de se obter resultados mais fidedignos, deve ser verificado se cada ocorrência reportada realmente corresponde a um real defeito no código-fonte. Após a exclusão dos alarmes falsos, é possível a obtenção do número de defeitos encontrados, tanto no código-fonte original quanto na versão modificada.

7.1.2 Expressividade

Outra categoria de métricas a ser analisadas neste experimento é referente ao grau de expressividade da solução no que tange as definição da semântica de verificação da igualdade. No contexto deste experimento, entende-se expressividade como a capacidade de se representar diferentes semânticas de comparação através dos mecanismos disponíveis na solução aqui proposta.

Com o objetivo de mensurar o grau de expressividade da solução, serão usadas duas métricas básicas: a quantidade de métodos relativos a definição da semântica de comparação, codificados da maneira usual, e a quantidade de anotações usadas para a expressão da lógica de verificação de igualdade, agrupadas pelo tipo de construto anotado. Também serão realizadas medições relativas à utilização de estratégias de comparação customizadas e definições de ordem de avaliação (usadas principalmente para o cálculo da ordem natural). Porém, para efeitos de comparação, é considerada mais expressiva a solução com menos métodos implementados.

7.1.3 Desempenho

Os indicadores de desempenho estão relacionados a diferenças no tempo de execução necessário para o processamento de requisições, entre a versão original da biblioteca e a versão pós migração, para a solução proposta. Não são abordadas neste experimento as

[4] FindBugs™ - Find Bugs in Java Programs. Disponível em: ⟨http://findbugs.sourceforge.net/⟩.

análises de desempenho relativas ao perfil de utilização de memória e à escalabilidade da biblioteca.

Para a obtenção dos indicadores de desempenho serão realizados dois *benchmarks* distintos, um com o perfil de *microbenchmark* (em que trechos específicos do programa têm seu desempenho mensurados isoladamente), e outro em que o programa é exercitado como um todo. O primeiro *benchmark* é composto pelo conjunto de testes de unidade inclusos no código-fonte da biblioteca, enquanto o segundo é formado por um teste pertencente à suíte de testes de desempenho (BLACKBURN et al., 2006), especialmente escrito para medições do tempo de resposta do JFreeChart. No caso do segundo teste, serão executadas dez iterações no processo de *warm up*[5] e, tanto as execuções de *warm up* quanto a execução mensurada, usam o maior conjunto de entrada de dados disponível (opções de execução "*-n 10 -s large*").

No intuito de minimizar eventuais desvios nas medidas obtidas, são realizadas dez iterações, nas quais ambos *benchmarks* são executados. Ao fim deste processo, são calculados as médias dos tempos de execução de ambos os testes. A partir dessas médias, obtém-se as diferenças entre o tempo de execução da versão original e da versão cujo código-fonte foi modificado pela adoção da solução.

7.2 Execução

Conforme foi dito, a primeira parte do experimento é composta da alteração do código-fonte da biblioteca JFreeChart para o uso da solução descrita neste trabalho. Para isso, foi gerada uma listagem contendo todas as classes que declaram algum dos métodos *equals()*, *hashCode()*, *compareTo()*, e, para cada ocorrência desta lista, são executados os seguintes passos:

a) Análise de qual estratégia de comparação é mais indicada para a classe sendo alterada;

b) Análise de quais propriedades são importantes para a lógica de verificação de igualdade encapsulada no método sendo removido, bem como sua ordem de avaliação, quando a classe sendo alterada incorporar o cálculo de ordem natural;

c) Anotação da classe, de seus atributos e de seus métodos, em função de sua semântica de comparação;

d) Remoção dos métodos de relativos ao mecanismo de verificação da igualdade que porventura estejam declarados na classe sendo migrada;

e) Verificação da corretude da classe alterada, através da execução do conjunto de testes de unidade a ela associado.

[5] Execuções não-mensuradas, realizadas no intuito de preparar o sistema para a execução do teste.

No decorrer da execução da conversão do código-fonte, foi necessária a criação de estratégias de comparação personalizadas para algumas classes. Essas classes são pertencentes à biblioteca JCommon, cujo código-fonte, conforme relatado anteriormente, não fora alvo de conversão. Também não foi possível a alteração de duas classes (*DateTickUnit* e *NumberTickUnit*) para a utilização da anotação *@Comparable*, em razão delas serem subclasses de uma classe que já tem definido os métodos de verificação de igualdade, o que impede a incorporação do *mixin* de comparação.

Outra peculiaridade encontrada foi que, para a utilização da biblioteca na suíte de testes DaCapo (BLACKBURN et al., 2006), foi necessária a implementação de uma lógica de *bootstrap*, responsável tanto por executar as configurações iniciais das fábricas de estratégia de comparação quanto pelo registro das estratégias de comparação personalizadas no escopo global. Nessa implementação, foi utilizado o mecanismo de extensão disponibilizado pela API da suíte de testes, onde o desenvolvedor pode fornecer uma classe de *callback*, cujos métodos são executados em momentos específicos do ciclo de vida da bateria de testes.

7.3 Análise dos Resultados

Após o término da conversão do JFreeChart para a utilização da solução aqui proposta, o código-fonte convertido foi submetido a análises com o foco na corretude, expressividade e no desempenho da solução.

7.3.1 Corretude

Conforme indicado na seção anterior, são utilizadas duas fontes de dados distintas para a construção dos indicadores de corretude: os resultados obtidos através da execução da suíte de testes de unidade do JFreeChart e o resultado do processo de análise estática do seu código-fonte. Na Tabela 11, encontram-se os resultados obtidos através da execução da suíte de testes de unidade, tanto na versão original (*baseline*), quanto na versão modificada.

Tabela 11 – Resultado experimento – Corretude (testes de unidade)

Baseline				Solução			
Testes	Falhas	Erros	% de Sucesso	Testes	Falhas	Erros	% de Sucesso
2141	6	2	99,63	2141	6	0	99,72

Nota: Entende-se por falhas divergências entre resultados esperados e obtidos, já erros são comportamentos inesperados em tempo de execução (como por exemplo, o levantamento de uma exceção não esperada).

Analisando os resultados dispostos na Tabela 11, pode-se notar que existem, na execução da suíte de testes baseada no código-fonte original, falhas e erros reportados. Dessas ocorrência, quatro têm relação, direta ou indireta, com a lógica de verificação da igualdade: duas falhas são relativas à implementação do método *hashCode()*, pertencentes a classes *LegendGraphic*, e dois erros são relativos ao método *hashCode()* da classe *XYImageAnnotation*.

Conforme resultados obtidos, após a migração do código-fonte para o uso da solução, pode-se notar um quase desprezível aumento da taxa de sucesso da suíte de testes. A partir de análise do relatório de execução da suíte de testes de unidade, foi detectado que tanto o erro referente ao método *hashCode()* declarado em *XYImageAnnotation* quanto a falha referente à implementação do método *hashCode()* da classe *LegendGraphic* foram eliminados após a adoção da solução.

Porém, após a migração da biblioteca, foi introduzida uma nova falha, também relativa ao método *XYImageAnnotation.hashCode()*: a lógica de verificação da igualdade falha quando é efetuada a verificação da igualdade entre uma instância construída através do operador *new* e uma desserializada. Após a checagem do código-fonte, detectou-se que a raiz da falha se encontrava não na lógica de verificação da igualdade, mas sim no código responsável pela desserialização de instâncias de *XYImageAnnotation*.

No que tange à análise estática do código-fonte do JFreeChart, estão dispostos, na Tabela 12, os resultados obtidos através da análise tanto do código-fonte original quanto do código-fonte modificado.

Tabela 12 – Resultado experimento – Corretude (análise estática)

Categoria	Baseline	Solução		
	Ocorrências	Detectadas	Alarmes Falsos	Total
Eq	18	10	10	0
HE	142	2	0	2
NP	0	0	0	0
FE	3	0	0	0
RV	1	0	0	0
Total	164	12	10	2

Legenda: Eq=Problemas na definição do método *equals()*; HE=Problemas de interdependência entre *equals()* e *hashCode()*; NP=Violações relativas à checagem de parâmetros nulos nas invocações à *equals()* (não-nulabilidade); FE=Comparação errônea entre valores em ponto flutuante e a constante *NaN*; RV=Checagem por valores específicos em resultados de invocações à *compareTo()*.

Através dos resultados dispostos na tabela, pode-se notar uma expressiva diminuição das violações encontradas no código-fonte (de 164 violações para apenas 12), sendo en-

contradas somente ocorrências de violações dos grupos *Eq* e *HE*.

Com relação à categoria *Eq*, chegou-se a conclusão que as ocorrências se tratavam de alarmes falsos, já que o algoritmo de checagem associado a essa categoria considera uma violação quando, na presença de uma relação de herança, a subclasse redefine o método *equals()* sem delegar parte da verificação ao método definido na superclasse. Essa premissa não se aplica à forma usada pela solução na geração do método *equals()*, na qual a lógica de comparação é delegada para uma estratégia construída por uma fábrica.

Ao contrário das violações da categoria *Eq*, as ocorrências encontradas do tipo *HE* são realmente problemas de implementação, relativas a violações do contrato de interdependência entre as operações *equals()* e *hashCode()* das classes *MonthDateFormat* e *QuarterDateFormat*, que, conforme relatado na seção 7.2, não puderam ser alteradas para o uso da anotação *@Comparable*, em função da existência de uma relação de herança entre elas e uma classe cujo código-fonte estava indisponível para alteração.

7.3.2 Expressividade

A métrica relativa à expressividade da solução é baseada na quantidade de métodos associados ao mecanismo de verificação da igualdade removidos e a quantidade de anotações utilizadas para a expressão da semântica neles encapsulada.

Para a obtenção dessas medidas, foi implementado um programa que examina o código-fonte alterado e o original à procura dos métodos removidos e das utilizações da anotação *@Comparable*, emitindo, ao final da execução, as medidas obtidas. Após a execução desse programa, foram obtidos os resultados dispostos nas Tabela 13, 14 e 15, apresentando, respectivamente, dados relativos à remoção de métodos, à utilização da anotação *@Comparable* e à implementação de estratégias de comparação customizada e de seu perfil de utilização.

Tabela 13 – Resultado experimento – Expressividade (métodos removidos)

Método	Baseline	Solução	Δ
equals()	215	2	-213
hashCode()	68	0	-68
compareTo()	20	1	-20

Nota: Não são levados em conta métodos declarados em estrátegias de comparação customizadas.

Analisando-se os dados da Tabela, 13, nota-se uma diminuição sensível dos métodos implementados para lógica de verificação da igualdade das classes pertencentes à biblio-

teca modificada. Com exceção de três métodos, toda a semântica comparativa pode ser expressa com o uso dos mecanismos disponíveis na solução proposta. Pode-se agrupar esses métodos em duas categorias, em função do motivo para o insucesso na remoção.

A primeira categoria é composta pelos métodos *equals()*, declarados nas classes *MonthDateFormat* e *QuarterDateFormat*, que não puderam ser removidos por já existirem suas definições nas superclasses, pertencentes a uma biblioteca de terceiro, cuja modificação do código-fonte não faz parte do escopo do experimento (conforme exposto na seção 7.3.1).

A segunda categoria é composta pelo método *compareTo()*, declarado na classe *TickUnit*. Esse método não pôde ser substituído em razão de uma discrepância das lógicas implementadas nas operações *equals()* e *compareTo()* de suas subclasses (*DateTickUnit* e *NumberTickUnit*). Nessas classes, a lógica encapsulada no método *equals()* utiliza-se de uma propriedade (*formatter*), porém, o tipo desta propriedade não é um subtipo do tipo definido pela interface *Comparable*.

A priori, poderia-se resolver esse problema com a utilização dos mecanismos de estratégias de comparação customizadas, porém existem nesse cenário características que impedem a adoção desse mecanismo, da maneira com a qual foram implementados pelo protótipo: a utilização de estratégias customizadas não trata questões estruturais, como a realização automática da interface *Comparable*, impossibilitando assim a invocação da operação *compare*(), já que ambos parâmetros fornecidos à essa operação devem realizar a interface *Comparable*.

Tabela 14 – Resultado experimento – Expressividade (utilização de anotações)

Construto Anotado	Ocorrências
Classe	113
Atributo	862
Método	12

Na tabela 14 estão dispostas as medidas correspondentes à utilização da anotação *@Comparable*, em função do construto por elas anotado. Através da análise dos dados na tabela e dos dados relativos à remoção de métodos (Tabela 13), pode-se verificar que, em média, foi necessária a utilização de aproximadamente oito anotações por classe.

Comparando com os resultados obtidos pelo experimento em (RAYSIDE et al., 2009) (dispostos na Tabela 5), houve um aumento sensível no número de anotações utilizadas. Pode-se atribuir essa discrepância à diferença na maneira pela qual as soluções indicam se uma informação deve, ou não, ser utilizada na lógica de verificação da igualdade. Conforme citado, na solução proposta em (RAYSIDE et al., 2009) todas as propriedades de um objeto, por padrão, são consideradas pela lógica de verificação da igualdade na

sua verificação (caso a informação não deva ser utilizada, o desenvolvedor deve anotar o atributo associado com a anotação *@NotKey*). Já na solução aqui proposta, é necessário que o desenvolvedor indique de maneira explícita que determinada informação deve ser utilizada na lógica da verificação da igualdade.

Tabela 15 – Resultado experimento – Expressividade (estratégias customizadas)

Escopo de Utilização	Ocorrências
Propriedade	3
Tipo	3
Global	8
Estratégias implementadas	13

Também podem ser extraídas da Tabela 15 informações referentes à utilização de estratégias de comparação customizadas. Para a migração, foram escritas treze estratégias de comparação customizadas, utilizadas para a definição da lógica de verificação da igualdade de classes definidas na biblioteca JCommons. Oito dessas estratégias escritas foram utilizadas no escopo de utilização global, enquanto as cinco estratégias restantes foram usadas no escopo de atributos (três ocorrências) e de classe (mais três ocorrências).

7.3.3 Desempenho

A último aspecto da migração analisado neste experimento é o aspecto referente aos impactos no desempenho da biblioteca após a adoção da solução aqui proposta. Conforme descrito anteriormente, para a obtenção dos indicadores de desempenho, são realizadas duas medições distintas.

a) Tempo de execução da suíte de testes de unidade, fornecida com código-fonte da biblioteca (perfil de *microbenchmark*);

b) Tempo de execução do *benchmark* associado ao JFreeChart (*chart*) incluso na suíte de testes DaCapo. Esse *benchmark*, segundo (BLACKBURN et al., 2006), reflete melhor os resultados esperados em um ambiente produtivo.

Na Tabela 16 encontram-se dispostos os resultados obtidos através de sucessivas medições[6]. Pode-se notar um aumento expressivo do tempo de execução da suíte de testes de unidade (cerca de 85%) e um pequeno aumento, de cerca de 6%, no tempo de execução no *benchmark* DaCapo (BLACKBURN et al., 2006).

[6] Essas medições foram efetuadas o seguinte ambiente: Intel i7-2600K 3,40 GHz (quad-core), 16Gb RAM e JDK 1.8.0_05-b13 (*mixed mode*). Não foram feitas alterações nas configurações do coletor de lixo e do *heap*.

Tabela 16 – Resultado experimento – Desempenho

	DaCapo			Testes de Unidade		
Iteração	Baseline (s)	Solução (s)	Δ	Baseline (s)	Solução (s)	Δ
1	4,808	5,084	–	6,933	12,814	–
2	4,808	5,097	–	7,127	13,012	–
3	4,780	5,204	–	6,949	12,963	–
4	4,803	5,170	–	6,928	12,937	–
5	4,804	5,135	–	7,004	12,987	–
6	4,812	5,120	–	7,019	13,033	–
7	4,794	5,122	–	7,090	12,824	–
8	4,756	5,118	–	7,069	12,820	–
9	4,986	5,037	–	6,846	12,990	–
10	4,852	5,060	–	6,878	13,033	–
Média	4,820	5,115	6,11%	6,984	12,941	85,29%

Essa diminuição do desempenho do programa, embora esperada, pode ser mitigada através da investigação dos gargalos que impactam no desempenho da biblioteca quando adotada a solução proposta. Entre esses possíveis gargalos encontra-se a utilização do mecanismo de reflexão da linguagem Java para obtenção dos valores associados às propriedades, impedindo otimizações como a expansão de métodos em linha (*method inlining*). Para a mitigação desse problema, pode-se fornecer uma nova implementação da fábrica de estratégias, que, utilizando-se do mecanismo de geração de *bytecode* em tempo de execução, crie estratégias que acessem as propriedades dos operandos sem o uso da reflexão.

Outra técnica que pode ser incorporada para o aumento do desempenho é a técnica de otimização utilizada na solução apresentada em (GRECH; RATHKE; FISCHER, 2010), na qual, através de técnicas adaptativas, a solução altera a ordem de avaliação dos métodos gerados, no intuito de eliminar checagens desnecessárias.

8 Conclusão e Trabalhos Futuros

8.1 Conclusão

Neste trabalho foram apresentados alguns conceitos referentes às propriedades inerentes da relação de igualdade entre duas entidades, bem como o impacto destas propriedades na implementação correta das operações de verificação da igualdade em linguagens orientadas a objetos. Além disso, foram apresentadas pesquisas, realizadas por vários autores, indicando o impacto real desse problema em programas atualmente em produção.

Também foi abordado o cenário atual do ferramental disponível para a definição correta da lógica de verificação da igualdade, tendo como foco principal a linguagem Java. Apesar da existência no âmbito acadêmico de soluções para definição da lógica de comparação (VAZIRI et al., 2007; RAYSIDE et al., 2009; GRECH; RATHKE; FISCHER, 2010), foram detectadas características que dificultam a suas respectivas adoções. Entre essas características pode-se citar a falta de um suporte adequado à integração com bibliotecas de terceiros e ao uso de *frameworks* de mercado. Outro aspecto, intimamente ligado ao conceito de igualdade e não abordado pelas soluções analisadas é o conceito do cálculo de ordem natural, através do qual é possível estabeler uma relação de ordem entre objetos.

Tendo este cenário em vista, foi proposto neste trabalho uma solução baseada na linguagem Java, tendo como objetivo a mitigação das principais deficiências encontradas nas soluções analisadas – cálculo de ordem natural e integração com bibliotecas e *frameworks* de mercado. De maneira análoga as soluções analisadas, também foram adotadas técnicas de programação orientadas a atributos (anotações), que apresenta vantagens como aumento de produtividade e facilidade de uso (GUERRA; FERNANDES, 2013).

Através da execução de um experimento informal, no qual uma biblioteca de código-fonte aberto e amplamente utilizada foi modificada para a utilização da solução proposta, pode-se concluir que a solução aqui apresentada é suficientemente expressiva para a definição da lógica de verificação da igualdade e do cálculo de ordem natural na maioria dos casos. A solução foi capaz de definir a semântica de comparação em praticamente todos os casos – de um universo composto de 116 classe passíveis de migração, foram migradas 113 (cerca de 97%). Vale-se notar que, conforme discorrido na seção 7.3.2, as razões para a impossibilidade de migração nestes casos não residem na falta de expressividade da solução aqui proposta, e sim em questões relativa ao protótipo (no caso das classes *MonthDateFormat* e *QuarterDateFormat*) e à falhas na modelagem do objeto da migração (na ocorrência relativa à classe *TickUnit*).

Ainda tendo como foco a expressividade, pode-se notar que, em relação à solução

proposta em (RAYSIDE et al., 2009), foi utilizada uma quantidade maior anotações na migração do JFreeChart para a solução aqui proposta. Essa divergência entre os números obtidos se dá em função da diferença na estratégia de definição de quais informações são importantes para a lógica de comparação entre os objetos. Nas soluções previamente apresentadas, por padrão um atributo faz parte do conjunto de informações a serem utilizadas na verificação da igualdade, devendo o desenvolvedor explicitamente indicar quando essas informações não devem ser utilizadas. Já na solução aqui proposta, o desenvolvedor deve indicar explicitamente quais informações devem ser utilizadas. Logo, apesar de em primeiro momento esses números indicarem que a solução proposta neste trabalho é menos expressiva, pode-se inferir um grau de expressividade maior à solução, já que, no último caso, fica explícito ao programador quais informações são utilizadas na verificação da igualdade.

Outra vantagem expressiva no uso da solução aqui proposta reside na corretude da lógica de verificação de igualdade. Apesar dos ganhos mensurados através da execução da suíte de testes de unidade serem inexpressivos (o aumento da taxa de sucesso entre a versão original e a migrada foi de apenas 0,09%), foi detectada uma diferença brutal entre a quantidade de problemas encontrados na versão original e na modificada. Enquanto através da inspeção do código-fonte original foram encontradas 164 violações, na versão migrada apenas 12 foram detectadas. Análises posteriores, indicaram que, destas 12 falhas, apenas duas eram legitimas.

Uma comunalidade encontrada nessas duas falhas restantes é que elas encontram-se nas classes *MonthDateFormat* e *QuarterDateFormat*, que conforme descrito, não puderam ser migradas para o uso da solução proposta em funções de deficiências na implementação do protótipo. O que indica que, segundo os critérios adotados pelo ferramental de análise estática, todas as implementações da lógica de verificação de igualdade geradas pela solução estão livres de erros.

No que diz respeito ao desempenho, o tempo de resposta da biblioteca modificada foi 6% maior que o da versão original. Também pode-se considerar esse problema como sendo relativo à implementação do protótipo, ao invés de um problema da solução em questão. É possível a diminuição desta diferença através, por exemplo, da geração, em tempo de execução ou de compilação, de métodos concretos para a verificação de igualdade, em vez do uso dos mecanismos de reflexão, de composição e de orientação a aspectos.

Por fim, vale destacar que apesar da adoção da solução apresentada neste trabalho apresentar vantagens, principalmente no que tange a corretude da lógica de verificação da igualdade (e do cálculo de ordem natural), ainda existem questões pendentes que devem ser estudadas: as soluções atuais (incluindo, a solução aqui apresentada) não abordam de maneira satisfatória o problema da utilização de objetos mutáveis em coleções com restrições de igualdade; e não são definidos mecanismos que tenham como foco a

verificação do cumprimento do contrato de consistência.

8.2 Trabalhos Futuros

No decorrer da elaboração deste trabalho, foi identificada uma série de caminhos possíveis para o aprimoramento da técnica de definição da lógica de verificação da igualdade através de uso de anotações. Entre esses possíveis caminhos, pode-se destacar os seguintes:

a) Uso de técnicas adaptativas tanto no algoritmo de decisão da incorporação da lógica de detecção e tratamento de ciclos quanto na implementação do algoritmo de verificação da igualdade (seções 6.2.2.1 e 6.2.4.4, respectivamente);

b) Incorporação de técnicas de análise estática e dinâmica do código-fonte para a identificação de problemas relativos à violação do subcontrato de consistência, tanto no uso da anotação *@Comparable* (seção 5.2.1.4) quanto no uso de estratégias de comparação customizadas (seção 5.2.2);

c) Proposição de mecanismos menos restritivos para a mitigação do problema relativo ao uso de elemento mutáveis em coleções com restrições de igualdade (seção 6.2.2.2).

Referências

Apache Commons Collections. Disponível em: ⟨http://commons.apache.org/proper/commons-collections/⟩.

Apache Lucene. Disponível em: ⟨http://lucene.apache.org/core/⟩.

AYEWAH, N. et al. Using static analysis to find bugs. *Software, IEEE*, IEEE, v. 25, n. 5, p. 22–29, 2008.

BLACKBURN, S. M. et al. The DaCapo benchmarks: Java benchmarking development and analysis. In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, 2006. p. 169–190.

BLOCH, J. How to design a good API and why it matters. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006. p. 506–507. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=1176622⟩.

BLOCH, J. *Effective Java (2Nd Edition) (The Java Series)*. 2. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN 0321356683, 9780321356680.

BRACHA, G.; COOK, W. Mixin-based inheritance. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 25, n. 10, p. 303–311, set. 1990. ISSN 0362-1340. Disponível em: ⟨http://doi.acm.org/10.1145/97946.97982⟩.

BRAIBANT, T.; JOURDAN, J.-H.; MONNIAUX, D. Implementing and reasoning about hash-consed data structures in coq. *arXiv preprint arXiv:1311.2959*, 2013.

BRUCE, K. *Foundations of Object-Oriented Languages: Types and Semantics*. Cambridge, Mass: The MIT Press, 2002. ISBN 9780262025232.

BRUCE, K. B. et al. *On Binary Methods*. [S.l.: s.n.], 1995.

CARDELLI, L.; MITCHELL, J. C. Operations on records. In: SPRINGER. *Mathematical foundations of programming semantics*. [S.l.], 1990. p. 22–52.

CARDELLI, L.; WEGNER, P. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, v. 17, n. 4, p. 471–523, 1985. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=6042⟩.

CEPA, V.; KLOPPENBURG, S. Representing explicit attributes in uml. In: *7th International Workshop on Aspect-Oriented Modeling (AOM)*. [s.n.], 2005. Disponível em: ⟨http://dawis2.icb.uni-due.de/events/AOM_MODELS2005/Cepa.pdf⟩.

Comparable (Java Platform SE 7). Disponível em: ⟨http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html⟩.

Eclipse. Disponível em: ⟨https://eclipse.org/⟩.

Eclipse Java Development Tools (JDT). Disponível em: ⟨https://eclipse.org/jdt/⟩.

EvoSuite | Automatic Test Suite Generation for Java. Disponível em: ⟨http://www.evosuite.org/⟩.

FindBugs™ - Find Bugs in Java Programs. Disponível em: ⟨http://findbugs.sourceforge.net/⟩.

FRASER, G.; ARCURI, A. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with EvoSuite. *Empirical Software Engineering*, p. 1–29, nov. 2013. ISSN 1382-3256, 1573-7616. Disponível em: ⟨http://link.springer.com/article/10.1007/s10664-013-9288-2⟩.

GAMMA, E. et al. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994. Disponível em: ⟨http://books.google.com.br/books?hl=pt-BR&lr=&id=6oHuKQe3TjQC&oi=fnd&pg=PT2&dq=design+patterns&ots=lOiMzZ9QKE&sig=sS-eZIA7i0QM4MAy1EDn1bjBi9w⟩.

GERSTING, J. L. *Mathematical Structures for Computer Science: A Modern Treatment of Discrete Mathematics*. W. H. Freeman, 2002. ISBN 0716743582. Disponível em: ⟨http://www.amazon.com/Mathematical-Structures-Computer-Science-Mathematics/dp/0716743582%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0716743582⟩.

GRECH, N.; RATHKE, J.; FISCHER, B. Generating correct and efficient equality and hashing methods using jequalitygen. 2010.

GUERRA, E.; FERNANDES, C. A qualitative and quantitative analysis on metadata-based frameworks usage. In: MURGANTE, B. et al. (Ed.). *Computational Science and Its Applications – ICCSA 2013*. Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, 7972). p. 375–390. ISBN 978-3-642-39642-7, 978-3-642-39643-4. Disponível em: ⟨http://link.springer.com/chapter/10.1007/978-3-642-39643-4_28⟩.

Hibernate. Disponível em: ⟨http://hibernate.org/⟩.

HOARE, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM*, ACM, v. 12, n. 10, p. 576–580, 1969.

INGALLS, D. H. H. A simple technique for handling multiple polymorphism. In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. New York, NY, USA: ACM, 1986. (OOPLSA '86), p. 347–349. ISBN 0-89791-204-7. Disponível em: ⟨http://doi.acm.org/10.1145/28697.28732⟩.

Inherited (Java Platform SE 6). Disponível em: ⟨http://docs.oracle.com/javase/6/docs/api/java/lang/annotation/Inherited.html⟩.

Java Archive Downloads - Java SE 1.4. Disponível em: ⟨http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javase14-419411.html⟩.

JavaBeans Spec. Disponível em: ⟨http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html⟩.

JCommon. Disponível em: ⟨http://www.jfree.org/jcommon/⟩.

JFreeChart. Disponível em: ⟨http://www.jfree.org/jfreechart/⟩.

JSR 175: A Metadata Facility for the JavaTM Programming Language. Disponível em: ⟨https://www.jcp.org/en/jsr/detail?id=175⟩.

KICZALES, G. *The art of the metaobject protocol*. [S.l.]: MIT press, 1991.

KICZALES, G. et al. *Aspect-oriented programming*. Springer, 1997. Disponível em: ⟨http://link.springer.com/chapter/10.1007/BFb0053381⟩.

KNUTH, D. E. *The Art of Computer Programming: Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, 1998. ISBN 0201896850. Disponível em: ⟨http://www.amazon.com/The-Art-Computer-Programming-Searching/dp/0201896850%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0201896850⟩.

LEIBNIZ, G. W. *Philosophical Papers and Letters: A Selection (Synthese Historical Library 2)*. D. Reidel/Springer, 1976. ISBN 902770693X. Disponível em: ⟨http://www.amazon.com/Philosophical-Papers-Letters-Selection-Historical/dp/902770693X%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D902770693X⟩.

LISCHNER, R. *Exploring C++ the programmer's introduction to C++*. Apress ; Springer-Verlag [distributor], 2009. ISBN 9781430203520 1430203528 9781430218951 1430218959. Disponível em: ⟨http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=276080⟩.

LISKOV, B.; GUTTAG, J. *Abstraction and specification in program development*. [S.l.]: MIT press, 1986.

LISKOV, B. H.; WING, J. M. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, v. 16, n. 6, p. 1811–1841, nov. 1994. ISSN 0164-0925. Disponível em: ⟨http://doi.acm.org/10.1145/197320.197383⟩.

MARTIN, R. C. Java and c++ a critical comparison. *Retrieved November*, v. 10, p. 2006, 1997. Disponível em: ⟨http://buiduychien.vnweblogs.com/gallery/4908/7.pdf⟩.

MARTIN, R. C. Design principles and design patterns. *Object Mentor*, v. 1, p. 34, 2000. Disponível em: ⟨http://scm0329.googlecode.com/svn-history/r78/trunk/book/Principles_and_Patterns.pdf⟩.

MARTIN, R. C. *Agile software development: principles, patterns, and practices*. [S.l.]: Prentice Hall PTR, 2003.

MELTON, H.; TEMPERO, E. An empirical study of cycles among classes in java. *Empirical Softw. Engg.*, v. 12, n. 4, p. 389–415, ago. 2007. ISSN 1382-3256. Disponível em: ⟨http://dx.doi.org/10.1007/s10664-006-9033-1⟩.

Meta-AspectJ Web Page. Disponível em: ⟨http://cgi.di.uoa.gr/~smaragd/maj/⟩.

MEYER, B. *Object-Oriented Software Construction*. 2 edition. ed. Upper Saddle River, N.J: Prentice Hall, 1997. ISBN 9780136291558.

MUSCHEVICI, R. *Multiple Dispatch in Practice: A Thesis Submitted to the Victoria University of Wellington in Fulfilment of the Requirements for the Degree of Master of Science in Computer Science*. Tese (Doutorado) — Victoria University of Wellington, 2009.

MUSCHEVICI, R. et al. Multiple dispatch in practice. In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. New York, NY, USA: ACM, 2008. (OOPSLA '08), p. 563–582. ISBN 978-1-60558-215-3. Disponível em: ⟨http://doi.acm.org/10.1145/1449764.1449808⟩.

NELSON, S.; PEARCE, D. J.; NOBLE, J. Understanding the impact of collection contracts on design. In: VITEK, J. (Ed.). *Objects, Models, Components, Patterns*. Springer Berlin Heidelberg, 2010, (Lecture Notes in Computer Science, 6141). p. 61–78. ISBN 978-3-642-13952-9, 978-3-642-13953-6. Disponível em: ⟨http://link.springer.com/chapter/10.1007/978-3-642-13953-6_4⟩.

NetBeans. Disponível em: ⟨https://netbeans.org/⟩.

Object (Java Platform SE 7). Disponível em: ⟨http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html⟩.

ODERSKY, M. *Programming in Scala*. 2nd ed. ed. [S.l.]: Artima, 2010. ISBN 9780981531649.

OMG. *OMG Unified Modeling Language OMG UML, Infrastructure, Version 2.4.1*. [s.n.], 2011. Disponível em: ⟨http://www.omg.org/spec/UML/2.4.1⟩.

PÅHLSSON, N. Aspect-oriented programming. *University of Kalmar*, 2002.

PEANO, G. *Arithmetices Principia Novo Methodo Exposita*. Bocca, 1889. Disponível em: ⟨http://eudml.org/doc/203509⟩.

PIERCE, B. C. *Types and programming languages*. [S.l.]: MIT press, 2002.

PIERCE, B. C.; TURNER, D. N. et al. Statically typed friendly functions via partially abstract types. 1993.

Project Coin. Disponível em: ⟨http://openjdk.java.net/projects/coin/⟩.

Randoop - Random test generation. Disponível em: ⟨https://code.google.com/p/randoop/⟩.

RAYSIDE, D. et al. Equality and hashing for (almost) free: Generating implementations from abstraction functions. In: *IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009*. [S.l.: s.n.], 2009. p. 342–352.

RUPAKHETI, C. R. *A Path-Based Approach for Analyzing Object Equality in Java*. Tese (Doutorado) — Clarkson University, 2010. Disponível em: ⟨http://www.rose-hulman.edu/~rupakhet/docs/eq-thesis.pdf⟩.

RUPAKHETI, C. R.; HOU, D. An empirical study of the design and implementation of object equality in java. In: *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*. New York, NY, USA: ACM, 2008. (CASCON '08), p. 9:111–9:125. Disponível em: ⟨http://doi.acm.org/10.1145/1463788.1463800⟩.

Shrike technical overview - WalaWiki. Disponível em: ⟨http://wala.sourceforge.net/wiki/index.php/Shrike_technical_overview⟩.

SNYDER, A. Encapsulation and inheritance in object-oriented programming languages. *ACM Sigplan Notices*, ACM, v. 21, n. 11, p. 38–45, 1986.

SourceForge - Download, Develop and Publish Free Open Source Software. Disponível em: ⟨http://sourceforge.net/⟩.

Spring. Disponível em: ⟨http://spring.io/⟩.

TARSKI, A. *Introduction to Logic and to the Methodology of the Deductive Sciences (Oxford Logic Guides)*. Oxford University Press, 1994. ISBN 019504472X. Disponível em: ⟨http://www.amazon.com/Introduction-Methodology-Deductive-Sciences-Oxford/dp/019504472X%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D019504472X⟩.

The AspectJ Project. Disponível em: ⟨http://www.eclipse.org/aspectj/⟩.

TUDOSE, C.; ODUBăşTEANU, C.; RADU, S. Java reflection performance analysis using different java development. In: DUMITRACHE, L. (Ed.). *Advances in Intelligent Control Systems and Computer Science*. Springer Berlin Heidelberg, 2013, (Advances in Intelligent Systems and Computing, v. 187). p. 439–452. ISBN 978-3-642-32547-2. Disponível em: ⟨http://dx.doi.org/10.1007/978-3-642-32548-9_31⟩.

VAZIRI, M. et al. Declarative object identity using relation types. In: ERNST, E. (Ed.). *ECOOP 2007 – Object-Oriented Programming*. Springer Berlin Heidelberg, 2007, (Lecture Notes in Computer Science, 4609). p. 54–78. ISBN 978-3-540-73588-5, 978-3-540-73589-2. Disponível em: ⟨http://link.springer.com/chapter/10.1007/978-3-540-73589-2_4⟩.

WADA, H.; SUZUKI, J. Modeling turnpike frontend system: A model-driven development framework leveraging UML metamodeling and attribute-oriented programming. In: *Model Driven Engineering Languages and Systems*. Springer, 2005. p. 584–600. Disponível em: ⟨http://link.springer.com/chapter/10.1007/11557432_44⟩.