Bruno Sofiato

Igualdade, ordem e *hashing*: uma
abordagem declarativa para a comparação
de objetos

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Ciências.

Bruno Sofiato

Igualdade, ordem e *hashing*: uma
abordagem declarativa para a comparação
de objetos

Versão Corrigida

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Ciências.

Área de concentração:
Sistemas Digitais

Orientador:
Prof. Dr. Fábio Levy Siqueira

São Paulo
2021

Autorizo a reprodução e divulgação total ou parcial deste trabalho, por qualquer meio convencional ou eletrônico, para fins de estudo e pesquisa, desde que citada a fonte.

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, 15 de agosto de 2021

Assinatura do autor:

Bruno Sofiato

Assinatura do orientador:

Fábio S. S.

Catálogo-na-publicação

Sofiato, Bruno

Igualdade, ordem e hashing: uma abordagem declarativa para a comparação de objetos / B. Sofiato -- versão corr. -- São Paulo, 2021. 96 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Igualdade; ordem; hashing. 2.Programação orientada a objetos. 3.Programação declarativa. 4.Linguagens de programação. 5.Qualidade de software. I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

Agradecimentos

Agradeço ao meu orientador Professor Doutor Fábio Levy Siqueira, que me incentivou a continuar com o trabalho mesmo nos momentos que eu me encontrava menos motivado e ao Professor Doutor Ricardo Luís de Azevedo da Rocha, que, apesar de não ser meu formalmente meu orientador por motivos burocráticos, me acolheu e ajudou como se eu fosse um dos seus orientandos.

Agradeço também ao Professor Doutor Paulo Sergio Muniz Silva, que me acompanhou durante meu crescimento no meio acadêmico e me ajudou no que foi necessário durante a minha pesquisa.

Agradeço a minha esposa – Rita de Cássia Tescke – por aguentar os momentos que eu estava mais ausente e por me incentivar nos momentos que estava sem motivação. Obrigado por ser minha companheira e estar ao meu lado. Te amo.

Agradeço aos meus pais – Maria Angélica Marques Sofiato e Antônio Carlos Sofiato – que me ensinaram a persistir e ter em mente que todo crescimento e progresso é gradual e que nós subimos uma escada degrau por degrau. Agradeço também ao meu irmão – Danilo Sofiato – que mesmo estando morando em outro estado, posso contar em qualquer situação.

Por fim, agradeço aos meus animais de estimação (alguns que não estão mais entre nós), que fazem qualquer problema que eu tenho parecer secundário com seu amor incondicional.

Resumo

Igualdade e ordem são conceitos relacionados que são usados diariamente. Como eles dependem do domínio, o programador tem que eventualmente codificá-los. Essa codificação, porém, não é fácil. Estudos apontam para ela como sendo uma conhecida fonte de defeitos em programas escritos em linguagens orientadas a objetos. Uma das causas é que tal implementação requer múltiplos métodos que devem ser consistentes entre si. Outra causa é mais insidiosa e é devida à como essas linguagens lidam com a passagem de mensagens. Diversas propostas almejam mitigar essas questões. Elas, porém, impõem restrições severas e não lidam com todos os casos. Não obstante, nenhuma lida com a ordem. Este trabalho propõe um método para comparação de objetos que é agnóstico à linguagem e trata os três aspectos distintos da comparação – igualdade, ordem e *hash*. A proposta é descrita utilizando-se semântica operacional estruturada no estilo de Plotkin. A sua corretude é provada formalmente. Também é descrito um protótipo que é avaliado baseado em três características de qualidade do modelo de qualidade de *software* da *ISO/IEC 25010* – adequação funcional, eficiência de desempenho e manutenibilidade. Resultados sugerem que a qualidade geral dos programas melhorou, apesar da queda de desempenho.

Palavras-chave: Igualdade; ordem; *hashing*. Programação orientada a objetos. Programação declarativa. Linguagens de programação. Qualidade de *software*.

Abstract

Equality and ordering are relatable concepts we use daily. As they depend on the domain at hand, the programmer must eventually code them. However, coding them is not easy. Several studies show that they are a known source of bugs in programs written in object-oriented languages. One of the causes is that it requires multiple methods that must be consistent with each other. Another cause is more insidious and is due to how these languages deal with message passing. Several proposals aim to mitigate those issues. They, however, enforce hard constraints and do not handle all the cases. Moreover, none of them deals with order. We propose a method of comparing objects that is agnostic of language and handles the three distinct aspects of comparison – equality, order, and hashing. We use Plotkin’s style operational semantics (SOS) to describe it. Its soundness is then formally proven. We also describe a prototype that is assessed based on three software quality characteristics of the *ISO/IEC 25010* software quality model – functional suitability, performance efficiency, and maintainability. Results hint that the overall quality of the software improved, despite the decline of performance.

Keywords: Equality; order; hashing. Object-oriented programming. Declarative programming. Programming languages. Software quality.

Sumário

1	Introdução	1
1.1	Motivação	1
1.2	Objetivo	2
1.3	Justificativa	3
1.4	Contribuições	4
1.5	Escopo	4
1.6	Método	5
1.7	Organização do Trabalho	6
2	A Igualdade sob o Ponto de Vista Matemático	7
2.1	Semântica da Igualdade	7
2.2	Relações de Equivalência	8
2.3	Relações de Ordem	9
2.4	Conclusões do Capítulo	10
3	Implementando a Comparação em Linguagens OO	11
3.1	Comparativo entre Linguagens	11
3.1.1	Estudo de Caso: Ponto e PontoColorido	12
3.1.2	Java	12
3.1.3	Ruby	15
3.2	A igualdade e as funções <i>hash</i>	16
3.3	Conclusões do Capítulo	17
4	Problemas na Implementação da Igualdade	18
4.1	A interdependência entre os métodos de comparação	18

4.2	Presença de Ciclos	19
4.2.1	Estudo de Caso: Fabricantes e Produtores	19
4.3	Mutabilidade e Coleções baseadas em Igualdade	21
4.4	Impacto da herança na simetria	21
4.4.1	Técnicas para Manutenção de Simetria	22
4.4.2	Simulação de despacho múltiplo	27
4.5	Conclusões do Capítulo	28
5	Trabalhos Correlatos	30
5.1	Análise de Deficiências	30
5.2	Fundamentos	31
5.3	Soluções Completas	32
5.3.1	Análise Geral das Soluções	34
5.4	Técnicas	37
5.4.1	Técnicas para manutenção da simetria	37
5.4.2	Técnicas para a simulação de despacho múltiplo	38
5.5	<i>Wizards</i> de IDEs	38
5.6	Outras contribuições	39
5.7	Conclusões do Capítulo	40
6	Proposta	42
6.1	Características Desejáveis	42
6.2	Descrição Informal	43
6.2.1	Registros de Comparação	43
6.2.2	Fase de Construção	44
6.2.3	Fase de Avaliação	46
6.2.4	Lidando com Ciclos	46
6.3	Descrição Formal	48
6.3.1	Notação e Formalismos Utilizados	48

6.3.2	Registros de Comparação	50
6.3.3	Fase de Construção	51
6.3.4	Fase de Avaliação	52
6.3.5	Lidando com Ciclos	55
6.4	Provas	57
6.4.1	Igualdade	57
6.4.2	Ordem	58
6.4.3	Código <i>hash</i>	59
6.5	Conclusões do Capítulo	60
7	Protótipo	62
7.1	Registros de Comparação	62
7.2	Fase de Construção	64
7.2.1	Denotando Semântica de Comparação	65
7.3	Fase de Avaliação	66
7.4	Lidando com Ciclos	67
7.5	Ligando os Objetos de Domínio ao Protótipo	68
7.6	Conclusões do Capítulo	70
8	Experimento	72
8.1	Avaliação em trabalhos correlatos	72
8.2	Método	73
8.2.1	<i>Corpus</i>	73
8.2.2	Passos para migração	74
8.3	Metodologia de Avaliação	75
8.3.1	Eficiência de Desempenho	78
8.3.2	Manutenibilidade	79
8.3.3	Adequação Funcional	81
8.4	Resultados Obtidos	82

8.4.1	Eficiência de Desempenho	84
8.4.2	Manutenibilidade	85
8.4.3	Adequação Funcional	85
8.5	Conclusões do Capítulo	87
9	Considerações Finais	89
9.1	Trabalhos Futuros	90
9.2	Artigos Publicados	90
	Referências Bibliográficas	92

Lista de Figuras

3.1	Diagrama de Classes – <i>Ponto</i> e <i>PontoColorido</i>	12
3.2	Código-fonte da classe <i>Ponto</i> (Java)	13
3.3	Código-fonte das classe <i>PontoColorido</i> (Java)	14
3.4	Código-fonte da classe <i>Ponto</i> (Ruby).	15
3.5	Código-fonte da classe <i>PontoColorido</i> (Ruby).	16
4.1	Diagrama de Classes – <i>Produto</i> e <i>Fabricante</i>	19
4.2	Código-fonte das classes <i>Fabricante</i> e <i>Produto</i>	20
4.3	Diagrama de Objetos – <i>Produto</i> e <i>Fabricante</i>	20
4.4	Diagrama de Objetos – Impacto da herança na simetria.	22
4.5	Código-fonte das classes <i>Ponto</i> (técnica de Odersky).	23
4.6	Código-fonte da classes <i>PontoColorido</i> (técnica de Odersky).	24
4.7	Método <i>Ponto.equals</i> (técnica de Subramaniam).	25
4.8	Diagrama de Classes – Hierarquia de Figuras Geométricas.	25
4.9	Diagrama de Objetos – Impacto da técnica de Subramaniam.	26
5.1	Alerta sobre falha na manutenção de contratos (<i>IntelliJ</i>).	39
6.1	Diagrama de Objetos – Tratamento de Ciclos.	47
6.2	Exemplo de Notação – Semântica Operacional.	49
6.3	Regras de Inferência – Construção de Registros de Comparação.	52
6.4	Regras de Inferência – Avaliação de Igualdade.	53
6.5	Regras de Inferência – Avaliação de Ordem.	54
6.6	Regras de Inferência – Cálculo <i>Hash</i>	55
6.7	Regras de Inferência – Tratamento de Ciclos.	56
7.1	Diagrama de Classe – Registros de Comparação.	63

7.2	Diagrama de Classe – Construção de Registros de Comparação. . .	64
7.3	Diagrama de Classe – Anotações.	65
7.4	Semântica de Comparação denotada através de anotações.	66
7.5	Estratégia de Comparação.	66
7.6	Trecho do código-fonte do aspecto <i>Comparable</i>	69
7.7	Fluxo durante a computação de uma operação de comparação. . .	70
7.8	Diagrama de Classes do Protótipo.	71
8.1	Taxonomia de características de qualidade da norma <i>ISO/IEC</i> <i>25010</i> (2010)	75
8.2	Trecho do arquivo <code>XMLStringDefault.java</code> do <i>xalan</i>	82
8.3	Resultados da execução do <i>microbenchmark</i>	84
8.4	Classe utilizada no <i>microbenchmark</i>	85
8.5	Trecho do arquivo <code>PDFColor.java</code> do <i>fop</i>	86
8.6	Trecho do arquivo <code>PDFResources.java</code> do <i>fop</i>	87

Lista de Tabelas

5.1	Comparativo entre as soluções completas.	35
6.1	Semântica de equivalência de <i>desconhecido</i>	45
6.2	Semântica de ordem natural de <i>desconhecido</i>	45
6.3	Sumário da Notação Utilizada.	48
8.1	<i>Corpus</i> da suíte de testes DaCapo.	74
8.2	Resultados do Experimento.	83

Lista de Abreviaturas

AOP	<i>Aspect Oriented Programming</i>
CLOS	<i>Common LISP Object System</i>
DRY	<i>Don't Repeat Yourself</i>
EJB	<i>Enterprise Java Beans</i>
FJ	<i>Featherweight Java</i>
IDE	<i>Integrated Development Environment</i>
JCF	<i>Java Collection Framework</i>
JDK	<i>Java Development Kit</i>
JVM	<i>Java Virtual Machine</i>
OO	<i>Orientado a Objetos</i>
PDF	<i>Portable Document Format</i>
PSL	<i>Princípio da Substituição de Liskov</i>
SBST	<i>Search-Based Software Testing</i>
SOAP	<i>Simple Object Access Protocol</i>
SVG	<i>Scalable Vector Graphics</i>
XML	<i>Extensible Markup Language</i>
XSL	<i>Extensible Stylesheet Language</i>
XSL-FO	<i>XSL Formatting Objects</i>
XSLT	<i>XSL Transformations</i>

1 Introdução

O conceito de igualdade – do latim *aequalis* (comparável, similar) – é usado intuitivamente no dia-a-dia quando deseja-se identificar semelhanças entre duas entidades.

Algumas vezes, no entanto, é necessário identificar não apenas “se”, mas também “como” duas entidades se diferenciam. Essa diferenciação é dada pelo conceito de ordem: dado dois objetos diferentes, intui-se que um deles seja considerado o maior. Frege (1879) formaliza a relação entre ordem e equivalência através da Lei da Tricotomia: dado dois elementos diferentes, um elemento deve ser considerado maior enquanto o outro menor.

Tarski (1994) argumenta que a semântica de comparação¹ depende do domínio sendo estudado. Independentemente da semântica, existem regras de boa-formação que devem ser observadas. No âmbito da matemática, a igualdade é regida pelas regras das chamadas relações de equivalência, enquanto a ordem pela chamada ordem estrita total.

1.1 Motivação

Segundo Tiobe (2021), seis das dez linguagens mais populares – Java, Python, C++, C#, Javascript e PHP – adotam o paradigma Orientado a Objetos (OO). No paradigma OO programas são compostos de objetos – representações de conceitos (abstratos e concretos) de um determinado domínio.

A implementação da semântica de comparação não é trivial. Normalmente, esse código é distribuído em operações, que devem apresentar resultados consistentes entre si. Diversos estudos (VAZIRI et al., 2007; FRASER; ARCURI, 2013; RUPAKHETI; HOU, 2008; RAYSIDE et al., 2009; SILVA et al., 2019) apontam para problemas nas implementações dessas operações. Cerca de 87,5% dos programas analisados por Vaziri et al. (2007) apresentaram inconsistências tanto nos resul-

¹Doravante, subentende-se neste trabalho como semântica de comparação a junção das semânticas de igualdade e ordem.

tados obtidos através de operações de comparação quanto violações das propriedades de boa-formação. Segundo Bloch (2018), esses cenários acarretam defeitos de difícil diagnóstico. Não obstante, Silva et al. (2019) pontuam que as violações mais comumente encontradas em programas que fazem uso do *Hibernate*² – arcabouço para mapeamento objeto-relacional que acabou sendo o padrão de fato adotado pela indústria – está ligada a forma com a qual a lógica de comparação é modelada.

Rayside et al. (2009) citam a fragilidade na presença de ciclos como um problema a ser observado em implementações da semântica de comparação. As comparações resultam em estouro de pilha, caso ciclos não sejam devidamente tratados. Apesar de serem considerados más-práticas (PARNAS, 1979), ciclos são comumente encontrados: um estudo realizado por Melton e Tempero (2007) encontrou ciclos em cerca de 45% de um *corpus* composto por 75 programas.

Outro problema é mais sutil e tem sua raiz nas entranhas das linguagens OO. Para que sejam considerada corretas, as relações de equivalência, têm que, entre outras coisas, serem simétricas. A manutenção da simetria é desafiadora quando existe uma relação de herança entre os operandos. As técnicas de mitigações existentes (ODERSKY, 2010; SUBRAMANIAM, 2020), porém, não são eficazes em todos os cenários: não pode-se usar essas técnicas quando objetos de classes diferentes podem ser considerados iguais. Segundo Bruce et al. (1995), somente as linguagens com despacho múltiplo conseguem manter a simetria em todos os cenários.

1.2 Objetivo

O objetivo desta pesquisa é descrever um mecanismo, agnóstico de linguagem, para denotação e posterior avaliação de semântica de comparação que trate, de maneira uniforme, de todos os aspectos da comparação. Esse mecanismo deverá incluir os aspectos de *hashing* e ordem – negligenciada pelas propostas existentes –, e também permitir a modelagem de cenários em que nessas propostas são impraticáveis.

A semântica será denotada declarativamente. Entre as vantagens dessa abordagem, pode-se citar a facilidade de uso (LLOYD, 1994) e a aderência ao princípio *Don't Repeat Yourself* (DRY, do inglês não repita a si mesmo) – princípio criado por Hunt e Thomas (2000) que dita que todo conhecimento deve ter uma

²<https://hibernate.org/>

representação única, inequívoca e autoritativa dentro de um sistema.

Como objetivos secundários, têm-se:

- A construção de um protótipo, na linguagem Java, que implemente a proposta descrita neste trabalho;
- A avaliação do protótipo (e indiretamente da proposta) sob a ótica de características de qualidade. Para isso será utilizado o modelo de qualidade da norma *ISO/IEC 25010* (2010).

1.3 Justificativa

Vaziri et al. (2007), Rayside et al. (2009) e Grech, Rathke e Fischer (2010) descrevem soluções para a definição da semântica de comparação entre objetos. No âmbito da indústria, duas soluções se destacam: AutoValue (BOURRILLION; MCMANUS, 2014) e o Projeto Lombok (ZWITSERLOOT; SPILKER, 2009).

Apesar de eficazes, essas soluções apresentam uma série de problemas. Algumas impõem restrições artificiais aos objetos a serem comparados (VAZIRI et al., 2007; GRECH; RATHKE; FISCHER, 2010; BOURRILLION; MCMANUS, 2014). Outras, como a de Rayside et al., não são uniformes no tocante a como a semântica de comparação deve ser definida, o que acaba dificultando o trabalho do programador (BAKER, 1993).

Existem duas deficiências que são preponderantes em todas as soluções existentes: nenhuma proposta permite a implementação da chamada comparação compatível com tipo (RUPAKHETI; HOU, 2010) – onde dois objetos podem ser instâncias de classes diferentes e ainda assim serem considerados iguais – de uma forma satisfatória³; e, mais importante, nenhuma dessas propostas trata da questão da ordem de objetos. Segundo Zermelo (1904), pode-se definir relações de ordem em todo e qualquer conjunto. Mesmo não fazendo sentido do ponto de vista do domínio, essa ordem “sintética” ainda pode ser útil quando esses objetos são usados em estruturas de dados como árvores binárias, entre outras.

Pode-se identificar comunalidades entre os mecanismos para definição da semântica de comparação em linguagens seguindo o paradigma OO. Deste modo, a definição agnóstica da solução se torna vantajosa. Neste caso, adoção em outra linguagem é uma questão de mapeamento entre os construtos agnósticos e

³A proposta descrita por Rayside et al. (2009) é a única das propostas existentes que permite a implementação da estratégia de comparação compatível com tipo. Para isso, porém, é necessária a escrita manual de código imperativo.

os construtos físicos, específicos de cada linguagem. Para evitar o enviesamento, algumas características da solução – modo de codificação da semântica de comparação e mecanismo de detecção de ciclos – são deixadas propositalmente em aberto (de todo modo, esses pontos são mais profundamente discutidos no âmbito do protótipo).

1.4 Contribuições

As principais contribuições deste trabalhos são:

- A descrição de um mecanismo agnóstico de linguagem para denotação e avaliação de semântica de comparação com as seguintes características:
 - tratamento dos três aspectos da comparação – igualdade, ordem, e *hashing* – de maneira uniforme;
 - denotação declarativa e uniforme da semântica de comparação;
 - tratamento transparente de ciclos;
 - suporte a objetos mutáveis;
 - suporte à comparação compatível com tipo.
- A prova formal da corretude do mecanismo descrito, sob a ótica das regras de boa-formação que regem as relações de equivalência e ordem estrita total;
- Um protótipo⁴, mapeando os construtos definidos de maneira agnóstica à linguagem em Java;
- Uma forma de avaliação de protótipos baseada na norma *ISO/IEC 25010* (2010) e na Suíte de Testes DaCapo (BLACKBURN et al., 2006).

1.5 Escopo

Não faz parte do escopo desta pesquisa a resolução da questão relativa à inserção de objetos mutáveis em coleções baseadas em igualdade. Segundo Nelson, Pearce e Noble (2010), coleções deste tipo fazem uso da semântica de igualdade de seus elementos (e.g. tabelas *hash*). Alterações no estado de igualdade de seus elementos após sua inserção deixam-nas em um estado inconsistente, em que buscas

⁴Disponibilizado como um programa de código-fonte aberto (disponível em <https://github.com/bsofiato/prototype-master-thesis>).

subsequentes aos elementos modificados são infrutíferas. Assume-se que esse é um problema de implementação das coleções, e não da semântica utilizada na comparação dos seus elementos. De todo modo, é discutido no Capítulo 9, no âmbito de trabalhos futuros, possíveis mecanismos para a mitigação do problema em questão.

É importante ressaltar que a solução descrita nesta pesquisa trata da igualdade sobre o âmbito de classes de equivalência⁵. Não é abordada a igualdade sobre o ponto de vista comportamental – onde dois objetos arbitrários são considerados iguais quando se comportam da mesma maneira. Logo, objetos comparados através da técnica aqui apresentada, podem, eventualmente, violar o Princípio da Substituição de Liskov (PSL) (LISKOV; WING, 1994). Formalmente, o Princípio da Substituição de Liskov dita que um comportamento atribuído a um tipo deve estar presente em todos os seus subtipos (na prática, objetos de um mesmo tipo devem ser intercambiáveis). Um dos motivos para o uso de uma abordagem não-comportamental para a igualdade é que inferências sobre o comportamento de objetos são notoriamente incomputáveis.

1.6 Método

Wegner (1976) identifica três paradigmas metodológicos relacionados à pesquisa na área de ciências da computação: racionalista, tecnocrata e científico. De acordo com Eden (2007), cada paradigma difere quanto a abordagem adotada para a obtenção de conhecimento. Esse trabalho adota o paradigma científico, onde conhecimentos obtidos *a priori* são corroborados por conhecimentos obtidos *a posteriori*, através de observação experimental.

A obtenção de conhecimentos *a priori* é realizada através da revisão da literatura. Essa revisão tem como objetivo responder quatro questões: como os aspectos da comparação se relacionam do ponto de vista matemático; quais as comunicações encontradas nas linguagens OO no tocante à implementação da lógica de comparação entre objetos; quais as principais dificuldades encontradas pelo programador ao implementar tal lógica; e, quais são as soluções propostas pela academia e a indústria para a resolução de tal problema. Esse conhecimento é então usado para a criação de uma proposta de solução para o problema em questão. Essa proposta é descrita de maneira informal e formal, e tem sua

⁵Uma classe de equivalência indica um sub-conjunto de elementos relacionados através de uma relação binária chamada de relação de equivalência. Esse conceito é explorado com mais detalhes no Capítulo 2.

corretude provada formalmente através de dedução formal.

A obtenção dos conhecimentos *a posteriori* para corroboração é realizada através de um experimento, onde os impactos da solução apresentada é avaliada sob a ótica de três características de qualidade definidas pela *ISO/IEC 25010* (2010) – Eficiência de Desempenho, Manutenibilidade, e Adequação Funcional. Para a realização desse experimento, é necessária que a proposta descrita no âmbito formal seja materializada em um protótipo. Esse protótipo é então utilizado na migração de um *corpus* de programas. O *corpus* utilizado pertence a Suíte de Testes DaCapo (BLACKBURN et al., 2006) (versão *9.12-bach*).

1.7 Organização do Trabalho

Além desta introdução, este trabalho compreende mais oito capítulos. O Capítulo 2 trata da igualdade sob o ponto de vista matemático e descreve as regras de boa formação das relações de equivalência e ordem. Já o Capítulo 3 aborda como a semântica de comparação é implementada em linguagens OO e discute as comunalidades entre elas. As dificuldades encontradas nessas implementações são então discutidas no Capítulo 4 enquanto o Capítulo 5 apresenta os trabalhos correlatos que visam mitigar essas dificuldades. Dado o contexto, uma proposta para a comparação de objetos é descrita no Capítulo 6. Essa descrição é dada do ponto de vista formal e informal. Também são apresentadas no Capítulo 6 provas formais quanto a corretude da solução apresentada.

Os Capítulos 7 e 8 são responsáveis pela materialização da proposta apresentada no Capítulo 6 e sua avaliação do ponto de vista de características de qualidade. O Capítulo 7 apresenta um protótipo da solução na linguagem Java e é dividido de maneira semelhante ao Capítulo 6, de modo a facilitar o mapeamento entre construtos correspondentes. Já o Capítulo 8 descreve um experimento onde o protótipo em questão é avaliado e os resultados dessa avaliação são discutidos.

O Capítulo 9 é o último deste trabalho. Nele, são descritas as considerações finais sobre os resultados obtidos, bem como percepções sobre possíveis trabalhos futuros.

2 A Igualdade sob o Ponto de Vista Matemático

Apesar de ser um conceito familiar, a igualdade é um conceito complexo e cheio de nuances. Outro conceito intimamente ligado à igualdade que também é familiar é o conceito de ordem. Este capítulo tem como objetivo formalizar ambos conceitos sob o ponto de vista matemático.

2.1 Semântica da Igualdade

O Princípio da Identidade dos Indiscerníveis (LEIBNIZ, 1976, cap. 35) rege o conceito de igualdade do ponto de vista da lógica clássica. Segundo esse princípio, duas entidades são consideradas iguais quando todas suas propriedades são idênticas (i.e. quando não se pode identificar qualquer diferença entre eles).

Apesar de consistente, o Princípio da Identidade dos Indiscerníveis é, em muitos casos, rígido demais. Às vezes é necessária uma visão mais pragmática sobre a igualdade. A observação de Tarski (1994, p. 56 et seq., tradução do autor) corrobora essa afirmação.

“Deve se notar, que existiram (e possivelmente ainda existam) matemáticos que se opuseram ao ponto 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 têm o mesmo tamanho e forma, ou figurativamente, dizer que, caso uma das figuras fosse colocada em cima da outra, ela a cobriria perfeitamente.”

Pode-se relaxar o Princípio da Identidade dos Indiscerníveis assumindo-se que exista um subconjunto das propriedades de uma entidade que seja suficiente para determinar a sua igualdade. Um exemplo cotidiano desse cenário é encontrado em bancos de dados relacionais.

Um banco de dados relacional é composto de tabelas. Tabelas, por sua vez, são compostas de linhas e colunas. Uma linha em uma tabela é identificada unicamente por uma chave. Uma chave é composta por uma ou mais colunas (DATE, 1997). Neste caso, as colunas que compõem a chave são suficientes para identificar unicamente uma linha.

É importante ressaltar que as propriedades usadas para identificar unicamente uma entidade pode variar em função do domínio. Por exemplo, em um programa de controle de estacionamentos, o número da placa pode ser suficiente para identificar unicamente um veículo. Outro sistema pode usar um critério diferente, como o número do chassi.

No decorrer desta dissertação, chamamos esse subconjunto de propriedades que são suficientes para determinar a igualdade de uma entidade de **estado de igualdade**.

É importante ressaltar que a comparação por propriedades conforme apresentada, não é suficientemente expressiva para a modelagem de uma semântica de cunho comportamental: dois objetos são iguais quando se comportam da mesma maneira. Essa asserção porém é análoga à equivalência de programas, que é um problema notoriamente indecidível (HOPCROFT; MOTWANI; ULLMAN, 2006).

2.2 Relações de Equivalência

Independentemente da semântica utilizada, existem características que são esperadas no que tange a relação entre duas entidades consideradas iguais. Na matemática essa relação é chamada de **relação de equivalência**. Entidades relacionadas entre si com relações desse tipo pertencem a mesma **classe de equivalência**.

Segundo Peano (1889), uma relação de equivalência deve ser **reflexiva**, **simétrica**, e **transitiva** para que seja considerada bem-formada. A Definição 1 formaliza essas condições.

Definição 1. Uma relação de equivalência $x = y$ é bem-formada se e somente se for reflexiva, simétrica e transitiva. Logo as seguintes expressões em lógica de primeira ordem devem ser verdadeiras:

Reflexividade: $(\forall x)(x = x)$;

Simetria: $(\forall x, y)[(x = y) \Leftrightarrow (y = x)]$;

Transitividade: $(\forall x, y, z)[((x = y) \wedge (y = z)) \Rightarrow (x = z)]$.

2.3 Relações de Ordem

Saber que duas entidades são diferentes muitas vezes não é o bastante. Às vezes, é necessário qualificar como duas entidades diferem. Um exemplo dessa qualificação é a ordem lexicográfica (ou alfabética) entre as palavras. Frege descreve esse conceito de “qualidade” entre diferenças em *Begriffsschrift*¹ (FREGE, 1879) na chamada Lei da Tricotomia. Originalmente formulada com base nos números naturais, a Lei da Tricotomia dita que, caso x e y sejam diferentes, então x deve ser considerado maior ou menor que y . Uma relação tricotômica transitiva é denominada **ordem estrita total**².

Do mesmo modo que as relações de equivalência, uma relação de ordem estrita total deve ter algumas características para que seja considerada bem-formada. Toda relação deste tipo deve ser **anti-simétrica**, **transitiva**, e **tricotômica** para que seja considerada bem-formada (SCHMIDT; STRÖHLEIN, 2012). A Definição 2 formaliza estas condições.

Definição 2. Uma relação de ordem $x < y$ é bem-formada se e somente se for tricotômica, antissimétrica e transitiva. Logo as seguintes expressões em lógica de primeira ordem devem ser verdadeiras:

Tricotomia: $(\forall x, y)[(x \neq y) \Leftrightarrow ((x > y) \vee (x < y))]$;

Anti-simetria: $(\forall x, y)[(x < y) \Leftrightarrow (y > x)]$;

Transitividade: $(\forall x, y, z)[((x > y) \wedge (y > z)) \Rightarrow (x > z)]$.

¹Escrita de termos, em alemão (tradução do autor).

²Uma pequena digressão com relação à ordem estrita total se faz necessária. Segundo Zermelo (1904), pode-se definir uma relação de ordem entre os elementos de todo e qualquer conjunto. Essa ordem, porém, pode não ter sentido do ponto de vista semântico (e.g. dada uma cor, qual o significado semântico na afirmação que uma cor é maior que outra). De todo modo, esta ordem “sintética” ainda se mostra vantajosa, uma vez que alguns algoritmos (e.g. algoritmos de agrupamento) necessitam que os dados de entrada estejam ordenados.

2.4 Conclusões do Capítulo

Este capítulo teve como objetivo delimitar o aspecto matemático da comparação entre entidades, representado pelas relações de ordem (estrita total) e equivalência.

A semântica dessas relações dependem do domínio. Todavia, existem características que devem ser válidas, independentemente da semântica utilizada. Relações de equivalência devem ser: **reflexiva**, **simétrica**, e **transitiva**. Já as relações de ordem devem ser **anti-simétrica**, **transitiva**, e **tricotômica**.

As fundações matemáticas apresentadas por este capítulo facilitam o entendimento das dificuldades de implementação delineadas no próximo capítulo.

3 Implementando a Comparação em Linguagens OO

Conforme discutido, o que determina se duas entidades – objetos – são iguais é normalmente determinado pelo domínio. Deste modo, é invariavelmente necessário que o programador escreva código representando como esses objetos devem ser comparados.

O objetivo deste capítulo é discutir como tal lógica é implementada em linguagens de programação OO e como os conceitos apresentados no Capítulo 2 se traduzem. Para isso será discutida a forma na qual são definidas semânticas de comparação em duas linguagens OO: Java e Ruby.

É importante fazer uma pequena digressão sobre os trechos de código apresentado neste capítulo. Os trechos aqui apresentados representam a forma mais direta de implementação dos métodos de comparação possível. Seu objetivo principal não é apresentar a forma correta de implementação (aliás, as implementações apresentadas contém erros que são explorados na Seção 4.4), mas sim ressaltar as comunicações encontradas entre as diversas linguagens de programação OO no tocante à implementação da comparação entre objetos.

3.1 Comparativo entre Linguagens

Apesar dos fundamentos matemáticos comuns, cada linguagem de programação tem suas idiossincrasias no que tange como a semântica de comparação deve ser implementada. Deste modo é importante identificar os pontos comuns entre as linguagens. Delimitando as comunicações entre as linguagens, é possível delinear uma solução agnóstica para o tratamento da comparação entre objetos.

3.1.1 Estudo de Caso: Ponto e PontoColorido

Para ilustrar como a semântica de comparação se traduz em código-fonte, é utilizado um exemplo utilizado por muitos autores (BRUCE et al., 1995; BLOCH, 2018; SUBRAMANIAM, 2020).

Figura 3.1: Diagrama de Classes –*Ponto* e *PontoColorido*.

O exemplo em questão abrange a modelagem de pontos em um plano cartesiano. Pontos têm duas propriedades – x e y –, referentes às suas coordenadas no plano cartesiano. Dois tipos distintos de pontos são definidos: **regulares** e **coloridos**. Pontos coloridos estendem o conceito de ponto, permitindo que cada ponto tenha uma coloração diferente. O diagrama da Figura 3.1 apresenta as classes e o relacionamento entre elas.

Com relação a semântica de comparação, Pontos (regulares) são considerados iguais quando têm as mesmas coordenadas (i.e., ocupam o mesmo lugar no plano cartesiano). Pontos coloridos iguais também compartilham as mesmas coordenadas, eles, porém, também devem ser da mesma cor.

3.1.2 Java

A primeira linguagem a ser analisada é a linguagem Java (GOSLING et al., 2000). Desenvolvida por James Gosling na Sun Microsystems na década de 1990, Java tem como mote *Write once, run anywhere*¹ – uma referência ao ambiente de execução ser baseado em uma Máquina Virtual (JVM). Segundo Tiobe (2021), é uma das linguagem mais utilizadas comercialmente. Não obstante, suas escolhas com relação a forma com o qual objetos são comparados é compartilhada com outras linguagens igualmente populares, dentre as quais o C#.

Java é uma linguagem fortemente tipada, com checagem de tipo estática e multi-paradigma. Apesar de normalmente associada ao paradigma OO, suas últimas versões incluem construtos que dão suporte à programação funcional.

¹Escreva uma vez, rode em todo lugar (tradução livre).

```
1 public class Ponto implements Comparable {
2     private double x;
3     private double y;
4
5     @Override
6     public boolean equals(Object o) {
7         if (o instanceof Ponto) {
8             Ponto p = (Ponto) o;
9             return compare(p.x, x) == 0 && compare(p.y, y) == 0;
10        }
11        return false;
12    }
13
14    @Override
15    public int hashCode() {
16        return hash(x, y);
17    }
18
19    @Override
20    public int compareTo(Object o) {
21        if (o instanceof Ponto) {
22            Ponto p = (Ponto)(o);
23            if (compare(p.x, x) != 0) {
24                return compare(p.x, x);
25            } else if (compare(p.y, y) != 0) {
26                return compare(p.y, y);
27            } else {
28                return 0;
29            }
30        }
31        return 1;
32    }
33 }
```

Figura 3.2: Código-fonte da classe *Ponto* (Java)

As Figura 3.2 e 3.3 exibem, respectivamente, o código-fonte de possíveis implementações das classes *Ponto* e *PontoColorido*, escrito em Java. O código em questão implementa a semântica de comparação conforme descrito anteriormente. Tudo que não seja estritamente necessário para a comparação foi omitido do código-fonte apresentado.

Analisando o código-fonte da classe *Ponto*, pode-se identificar três métodos: *equals*, *compareTo* e *hashCode*. Cada um desses métodos lida com diferentes aspectos da comparação. A classe *PontoColorido* redefine os métodos em questão, de modo que a cor dos pontos coloridos seja levada em conta.

O método *equals* é o método mais fácil de se entender. Dada a chamada *x.equals(y)*, *true* é retornado caso $x = y$. Caso contrário, *false* é retornado. A verificação propriamente dita é feita através da comparação das coordenadas uma-a-uma.

A realização da interface *Comparable* indica à linguagem Java que instâncias

```

1 public class PontoColorido extends Ponto {
2     private Cor cor;
3
4     @Override
5     public boolean equals(Object o) {
6         if (o instanceof PontoColorido) {
7             if (super.equals(o)) {
8                 PontoColorido p = (PontoColorido) o;
9                 return equals(cor, p.cor);
10            }
11        }
12        return false;
13    }
14
15    @Override
16    public int hashCode() {
17        return hash(super.hashCode(), cor);
18    }
19
20    @Override
21    public int compareTo(Object o) {
22        if (o instanceof PontoColorido) {
23            if (super.compareTo(o) != 0) {
24                return super.compareTo(o);
25            } else {
26                PontoColorido p = (PontoColorido) o;
27                return compare(cor, p.cor, Comparator.naturalOrder());
28            }
29        }
30        return 1;
31    }
32 }

```

Figura 3.3: Código-fonte das classe *PontoColorido* (Java)

de *Ponto* são ordenáveis (por padrão, objetos não são ordenáveis na linguagem Java). A interface *Comparable* define um único método – *compareTo*² – que modela a Lei da Tricotomia. O método *compareTo* deve ser implementado de modo que as seguintes regras sejam válidas:

$$x.compareTo(y) = \begin{cases} 0 & \text{caso } x=y \\ < 0 & \text{caso } x < y \\ > 0 & \text{caso } x > y \end{cases}$$

O terceiro método relacionado à igualdade declarado pelas classes *Ponto* e *PontoColorido* é o método *hashCode*. O método *hashCode* implementa uma função *hash* de uso geral.

Em Java, a implementação de *hashCode* segue um contrato implícito, que o

²[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Comparable.html#compareTo\(T\)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Comparable.html#compareTo(T))

relaciona ao método *equals*. Segundo Bloch (2018), todo objeto considerado igual deve ter o mesmo código *hash* (códigos *hash* diferentes para objetos diferentes é desejável, mas não mandatório).

3.1.3 Ruby

A segunda linguagem analisada é a linguagem Ruby (FLANAGAN; MATSUMOTO, 2008), desenvolvida por Yukihiro Matsumoto. Contemporânea a linguagem Java, Ruby é uma linguagem multi-paradigma que segue a filosofia de Smalltalk, na qual todos valores são objetos. Seu sistema de tipos difere do adotado por Java: apesar de ambos serem fortemente tipadas, em Ruby a verificação se dá de forma dinâmica.

Apesar de não tão popular quanto outrora, Ruby tem características interessantes. A ordenação de objetos em Ruby usa o mecanismo de *Mixins* – agrupamentos de métodos e atributos em unidades que podem ser incluídas em classes arbitrárias. Essa abordagem³ foi proposta pelo *ISO C++20* (2018) como forma de facilitação da definição das operações de comparação.

```
1 class Ponto include Comparable
2
3   attr_accessor :x, :y
4
5   def initialize(x: nil, y: nil)
6     self.x = x
7     self.y = y
8   end
9
10  def hash
11    [x, y].hash
12  end
13
14  def <=>(op)
15    (x != op.x) ? x <=> op.x : y <=> op.y
16  end
17 end
```

Figura 3.4: Código-fonte da classe *Ponto* (Ruby).

As Figuras 3.4 e 3.5 apresentam o código-fonte de possíveis implementações das classes *Ponto* e *PontoColorido* em Ruby. As classes seguem o mesmo tipo de implementação utilizado em Java (Seção 3.1.1). Nota-se a presença de dois métodos – *hash* e *<=>* – cuja semântica é idêntica à semântica dos métodos *hashCode* e *compareTo* descrita no exemplo em Java.

³No caso de C++, é usada herança múltipla ao invés de *mixins*.

```
1 class PontoColorido < Ponto
2
3   attr_accessor :cor
4
5   def initialize(x: nil, y: nil, cor: nil)
6     super x: x, y: y
7     self.cor = cor
8   end
9
10  def hash
11    [super, cor].hash
12  end
13
14  def <=>(op)
15    (super == 0) ? cor <=> op.cor : super
16  end
17 end
```

Figura 3.5: Código-fonte da classe *PontoColorido* (Ruby).

Nota-se também a inclusão da *mixin Comparable*. A *mixin Comparable* é implementada seguindo o padrão de projetos *Template method* (GAMMA et al., 1995) e define diversos métodos⁴ a partir de `<=>`. Um desses métodos é o operador `==`, cuja semântica é idêntica ao método *equals* do exemplo em Java.

3.2 A igualdade e as funções *hash*

Tanto Ruby quanto Java definem métodos que têm relação direta com os conceitos matemáticos introduzidos no Capítulo 2. No caso da relação de equivalência (i.e., igualdade) são definidos os métodos `==` e *equals*. Já a ordem é definida pelos métodos `<=>` e *compareTo* que modelam a Lei da Tricotomia.

Notavelmente, existe um terceiro método que têm a mesma semântica em ambas a linguagem e que não tem nenhuma contra-partida teórica. Esse método é um artefato puramente de implementação e implementa uma função *hash* de uso geral. Em Ruby esse método é chamado de *hash* enquanto em Java seu nome é *hashCode*.

Segundo Cormen et al. (2009), uma função *hash* tem como objetivo mapeamento de chaves em um conjunto potencialmente menor de valores. Esses valores são então utilizados como indexadores em uma classe de coleções chamadas de coleções baseadas em igualdade por Nelson, Pearce e Noble (2010). Coleções percententes à essa categoria são muito eficientes. Entre essas coleções encontram-se

⁴Além de `==`, Ruby define outros dois métodos – `===` e *eql?* – com dois propósitos distintos. O primeiro é usado em blocos multi-condicionais (i.e., blocos *case/when* enquanto o segundo é usado na verificação de igualdade em chaves de *arrays* associativos (tabelas *hash*).

as Tabelas *hash*⁵.

Apesar de não ser um conceito formal matemático, a implementação de *hash-Code* também está sujeita a regras de boa-formação. Segundo Bloch (2018), é mandatório que dois objetos considerados iguais (i.e., `x.equals(y) == true`) tenham o mesmo código *hash*. Em contrapartida, é desejável (mas não mandatório) que objetos diferentes tenham códigos *hash* diferentes. A Definição 3 formaliza esse conceito.

Definição 3. *Uma função hash h é bem-formada se e somente se $(\forall x, y)[(x = y) \Rightarrow (h(x) = h(y))]$.*

Notavelmente, Bloch (2018) pontua que uma função *hash* que retorne um valor constante é uma função válida. Todavia, essa função é péssima, uma vez que degradam o desempenho das buscas em tabelas *hash* de $\mathcal{O}(1)$ para $\mathcal{O}(n)$, fazendo com que seu desempenho fique equivalente ao nível de estruturas como, por exemplo, listas ligadas.

3.3 Conclusões do Capítulo

Apesar de diferentes no ponto de vista de sintaxe, as formas que linguagens OO definem a semântica de comparação de objetos são muito semelhantes: a implementação da semântica de igualdade é dividida em métodos que devem ser consistentes entre si.

Entre esses métodos, encontra-se uma função que não tem um correspondente teórico – a função *hash*. A corretude e a eficiência dessa função é de suma importância, já que as chamadas coleções baseadas em igualdade a usa e tem seu desempenho diretamente relacionado a sua eficiência.

Tendo-se definido um *baseline* de como a comparação de objetos funciona em linguagens OO, pode-se partir para a análise das dificuldades encontradas durante a sua implementação.

⁵Também chamadas de tabelas de espalhamento.

4 Problemas na Implementação da Igualdade

A primeira vista os exemplos indicados no capítulo anterior não parecem ser especialmente difíceis de entender. Apesar da aparente simplicidade, diversos estudos (VAZIRI et al., 2007; RUPAKHETI; HOU, 2008; FRASER; ARCURI, 2013; SILVA et al., 2019). apontam para um alto grau de defeitos na implementação dos métodos que lidam com a comparação entre objetos.

Antes de iniciar a discussão, é importante ressaltar que as discussões apresentadas nesta seção fazem uso da terminologia (i.e., nome dos métodos) adotada na linguagem Java. A problemática aqui descrita, porém, também é pertinente a outras linguagens OO.

4.1 A interdependência entre os métodos de comparação

A primeira dificuldade vem do fato que existe um contrato informal entre esses três métodos – *equals*, *compareTo*, e *hashCode* – que obriga que seus resultados sejam consistentes entre si: dois objetos considerados iguais pelo método *equals* devem ter o mesmo código *hashCode*; e, serem também considerados iguais pelo método *compareTo* (i.e., `x.compareTo(y)` deve ser igual a zero).

Segundo Bloch (2018), falhas na implementação desse contrato resultam em falhas de difícil diagnóstico. Esta noção é corroborada por Vaziri et al. (2007), Fraser e Arcuri (2013) e Rayside et al. (2009), que identificam violações na consistências entre os métodos de comparação em diversos programas e as citam como razões para defeitos em programas.

Infelizmente, não é possível a criação de checagens automáticas que garantam que os três métodos cumpram esse contrato. Decidir se dois programas (ou métodos) são consistentes entre si de uma forma arbitrária é um problema

indecidível¹.

Algumas ferramentas adotam heurísticas para tentar alertar o programador de possíveis erros na implementação desses métodos. Findbugs (2021), por exemplo, avisa o programador caso o método *equals* seja definido sem que o método *hashCode* também o seja. De todo modo, como toda heurística, essas checagens estão sujeitas a falso-positivos e falsos-negativos.

4.2 Presença de Ciclos

Um ciclo é um fenômeno no qual uma referência arbitrária pode ser atingida, direta ou indiretamente, através da navegação por suas associações. As boas práticas de programação desencorajam ciclos, uma vez que ciclos tendem a impactar negativamente a modularidade, testabilidade e o acoplamento (PARNAS, 1979). Apesar disso, estudos mostram que ciclos são comumente encontrados no código-fonte de programas de diversos tamanhos (MELTON; TEMPERO, 2007).

4.2.1 Estudo de Caso: Fabricantes e Produtores

Conforme discorrido por Rayside et al. (2009), as operações de igualdade são particularmente vulneráveis a ciclos. Pode-se usar um sistema de inventário hipotético para ilustrar o porquê. Duas classes – *Produto* e *Fabricante* – compõem esse sistema. Um fabricante cria inúmeros produtos enquanto produtos são fabricados por um único fabricante. A Figura 4.1 representa a estrutura de classes desse sistema.

Figura 4.1: Diagrama de Classes – *Produto* e *Fabricante*.

A primeira vista, pode-se assumir que o critério para se determinar que um fabricante seja equivalente a outro inclua os produtos que são fabricados. De maneira semelhante, pode-se admitir que dois produtos só são iguais se forem fabricados pelo mesmo fabricante. Os trechos de código da Figura 4.2 representam o código-fonte da operação *equals* das classes *Fabricante* e *Produto*.

Infelizmente, o código-fonte na Figura 4.2 acarreta em falhas em tempo de

¹Pode-se reduzir esse problema ao problema de equivalência entre programas. Conforme discutido por Hopcroft, Motwani e Ullman (2006), a equivalência de programas é um problema indecidível.

```

1 public class Fabricante {
2     private String nome;
3     private List<Produto> produtos = new ArrayList<>();
4
5     @Override
6     public boolean equals(Object o) {
7         if (o instanceof Fabricante) {
8             Fabricante that = (Fabricante) o;
9             return Objects.equals(nome, that.nome) &&
10                Objects.equals(produtos, that.produtos);
11        } else {
12            return false;
13        }
14    }
15 }
16
17 public class Produto {
18     private String nome;
19     private Fabricante fabricante;
20
21     @Override
22     public boolean equals(Object o) {
23         if (o instanceof Produto) {
24             Produto produto = (Produto) o;
25             return Objects.equals(nome, that.nome) &&
26                Objects.equals(fabricante, produto.fabricante);
27        } else {
28            return false;
29        }
30    }
31 }

```

Figura 4.2: Código-fonte das classes *Fabricante* e *Produto*.

execução. Assumindo-se uma estrutura de objetos conforme disposta na Figura 4.3, pode-se entender o porquê. A chamada à `a.equals(b)` resulta na chamada `c.equals(d)`, uma vez que os objetos *c* e *d* são, respectivamente, produtos produzidos por *a* e *b*. A chamada `c.equals(d)` resulta na chamada `a.equals(b)`, dado que *a* e *b* são, respectivamente, os fabricantes de *c* e *d*. Logo, têm-se uma série de chamadas – `a.equals(b) ⊢ c.equals(d) ⊢ a.equals(b) ⊢ c.equals(d) ⊢ a.equals(b) ...` – que progridem indefinidamente. Esse ciclo só termina quando o espaço da pilha de chamadas da linguagem termina, o que resulta nos chamados estouros de pilha².

Figura 4.3: Diagrama de Objetos – *Produto* e *Fabricante*.

Conforme pontuado por Rayside et al. (2009), desenvolvedores normalmente

²No inglês, *stack overflow*.

veem duas opções nesse caso: ignorar os atributos que possam levar a ciclos; ou deixar com que as comparações possam estourar a pilha. Rayside et al., porém, pontuam que é possível a criação de implementação que tratem ciclos da forma correta.

4.3 Mutabilidade e Coleções baseadas em Igualdade

Objetos mutáveis são objetos que têm seu estado alterado após sua construção. Esses objetos impõem um desafio extra, cuja problemática não está ligada à lógica de comparação *per se*, e sim no uso desses objetos em uma categoria específica de coleção. Nelson, Pearce e Noble (2010) denomina essa categoria de coleções de coleções baseadas em igualdade³.

De modo geral, essas coleções necessitam que os seus itens não tenham seu estado de igualdade alterado após a sua inserção. Qualquer alteração após a inserção pode acarretar falhas nos processos de buscas e levar a defeitos de difícil diagnóstico. Segundo observações de Nelson, Pearce e Noble, cerca de 19% dos objetos inseridos em coleções desse tipo têm alterações em seu estado da igualdade após a sua inserção.

De modo a mitigar esse problema, é recomendado que objetos não tenham seu estado de igualdade alterado após sua construção (LISKOV; GUTTAG et al., 1986). Algumas das propostas existentes (VAZIRI et al., 2007; GRECH; RATHKE; FISCHER, 2010; BOURRILLION; MCMANUS, 2014), inclusive, adotam essa abordagem.

Impedir mutações, apesar de eficaz, é muito restritivo. As mudanças de estado fazem parte do paradigma OO e são largamente utilizadas. Impedir objetos mutáveis impactaria gravemente a proposta em questão. Uma abordagem que impeça a inserção de itens que alterem seu estado de igualdade em coleções baseadas em igualdade parece mais adequada. Todavia, a descrição de tal abordagem **não** faz parte do escopo deste trabalho (sendo, todavia, um possível trabalho futuro).

4.4 Impacto da herança na simetria

Existe, porém, um problema mais traiçoeiro que tem suas raízes em como linguagens OO escolhem qual método será executado durante uma chamada de método.

³*Equality-based collections*, no texto original

O método mais comum de resolução é o Despacho Simples. No despacho simples um único objeto – o objeto recebedor – é usado para definir qual método é executado. Outro método de resolução é o chamado Despacho Múltiplo. No despacho múltiplo, múltiplos objetos são analisados para se decidir qual método é executado. Praticamente todas as linguagens OO são baseadas em despacho simples.

Conforme discutido por Bruce et al. (1995), o despacho simples impossibilita que a simetria seja mantida na presença de herança. Isto é problemático, uma vez que em linguagens com verificação de tipos estática, usa-se a herança⁴ para a implementação do polimorfismo de subtipo – um dos pilares do paradigma OO.

Figura 4.4: Diagrama de Objetos – Impacto da herança na simetria.

Pode-se ilustrar⁵ melhor a problemática com o seguinte exemplo: dado dois objetos – *a* e *b* –, conforme o digrama de objetos da Figura 4.4. A invocação de *a.equals(b)* dispara o método *Ponto.equals*. Considerando a implementação apresentada na Figura 3.2, o método *Ponto.equals* checa somente se os operandos têm as mesmas coordenadas. Deste modo *a.equals(b)* resulta em *true*. Por outro lado, *b.equals(a)* dispara o método *equals* em *PontoColorido* (Figura 3.3). Essa versão de *equals* também exige que ambos operandos tenham a mesma cor. O objeto *a*, porém, não é uma instância de *PontoColorido*. Logo *b.equals(a)* resulta em *false*.

Obter resultados diferentes para *a.equals(b)* e *b.equals(a)* é uma violação da simetria (outra conclusão válida é que a implementação da semântica de comparação exemplificada na Seção 3.1.2 está errada).

4.4.1 Técnicas para Manutenção de Simetria

Conforme explanado, a presença de herança no mínimo complica a implementação das operações de comparação entre objetos.

A questão que ainda permanece é se a manutenção da simetria é possível quando a herança é utilizada. Alguns autores, como Subramaniam (2020) são

⁴Para efeito de argumentação, assume-se que herança e realização de interfaces são equivalentes.

⁵O mesmo racional vale para a comparação através do método *compareTo*.

categoricos ao afirmar que a simetria pode ser mantida. Outros são taxativos ao afirmar que não. Bloch (2018, p. 42, tradução do autor), por exemplo, argumenta que:

“Simplesmente não existe uma maneira de estender uma classe instanciável adicionando um aspecto e manter o contrato de comparação, a não ser que aceite-se abandonar os benefícios da abstração baseada em objetos”

Nesta seção, são discutidas técnicas que têm como objetivo a manutenção de simetria quando a herança é utilizada.

4.4.1.1 Técnicas de Odersky (2010) de Subramaniam (2020)

```
1 public class Ponto implements Comparable {  
2     private double x;  
3     private double y;  
4  
5     protected boolean canEqual(Object o) {  
6         return o instanceof Ponto;  
7     }  
8  
9     @Override  
10    public boolean equals(Object o) {  
11        if (o instanceof Ponto) {  
12            Ponto p = (Ponto) o;  
13            return compare(p.x, x) == 0 &&  
14                compare(p.y, y) == 0 &&  
15                o.canEqual(this);  
16        }  
17        return false;  
18    }  
19  
20    @Override  
21    public int hashCode() {  
22        return hash(x, y);  
23    }  
24 }
```

Figura 4.5: Código-fonte das classes *Ponto* (técnica de Odersky).

Tanto Odersky (2010) quanto Subramaniam (2020) delineiam técnicas⁶ que têm como objetivo manter a simetria durante a comparação de objetos.

O trecho de código nas Figuras 4.5 e 4.6 exhibe o exemplo descrito na Seção 3.1.2, de modo a adotar a técnica para manutenção de simetria descrita por Odersky. Pode-se notar a inclusão de do método *canEqual* nas classes *Ponto* e *PontoColorido*.

⁶Essas técnicas são discutidas com maiores detalhes na Seção 5.4.1.

```
1 public class PontoColorido extends Ponto {
2     private Cor cor;
3
4     @Override
5     protected boolean canEqual(Object o) {
6         return o instanceof PontoColorido;
7     }
8
9     @Override
10    public boolean equals(Object o) {
11        if (o instanceof PontoColorido) {
12            if (super.equals(o)) {
13                PontoColorido p = (PontoColorido) o;
14                return equals(cor, p.cor);
15            }
16        }
17        return false;
18    }
19
20    @Override
21    public int hashCode() {
22        return hash(super.hashCode(), cor);
23    }
24 }
```

Figura 4.6: Código-fonte da classes *PontoColorido* (técnica de Odersky).

Seguindo o exemplo dado na Seção 4.4, a invocação de `a.equals(b)` dispara o método *Ponto.equals*. O método *Ponto.equals* checa se *a* e *b* têm as mesmas coordenadas e se o `b.canEqual(a)` retorna *true*. No caso específico, `b.canEqual(a)` retorna *false*, uma vez que *a* é uma instância de *Ponto* e o método *PontoColorido.canEqual* exige que *a* seja uma instância de *PontoColorido*. Logo, `a.equals(b)` resulta em *false*. Já no caso de `b.equals(a)`, o método *equals* em *PontoColorido* é invocado. Essa versão de *equals* exige que o parâmetro fornecido seja uma instância de *PontoColorido*. Como *a* não é uma instância de *PontoColorido*, `b.equals(a)` resulta em *false*.

Neste caso, ao contrário do descrito na Seção 4.4, tanto `a.equals(b)` e `b.equals(a)` resultam no mesmo valor – *false*. Logo, a simetria foi mantida.

Caso a técnica descrita por Subramaniam (2020) fosse adotada, teria-se um código semelhante. O trecho de código da Figura 4.7 mostra o corpo do método *equals* da classe *Ponto* após a adoção da técnica. Nenhum outro método além dele foi modificado (tendo como base o código do exemplo original descrito na Seção 3.1.2). Neste caso, a simetria também é mantida: como as classes concreta dos operandos são comparadas, tanto `a.equals(b)` quanto `b.equals(a)` resultam em *false*, uma vez que *Ponto* $\neq$ *PontoColorido*.

```

1 @Override
2 public boolean equals(Object o) {
3     if (getClass() != o.getClass()) {
4         return false;
5     } else {
6         Ponto p = (Ponto) o;
7         return compare(p.x, x) == 0 &&
8             compare(p.y, y) == 0;
9     }
10 }

```

Figura 4.7: Método *Ponto.equals* (técnica de Subramaniam).

Figura 4.8: Diagrama de Classes – Hierarquia de Figuras Geométricas.

4.4.1.2 Estudo de Caso: Formas Geométricas

Existem alguns cenários onde as técnicas apresentadas na Seção 4.4.1.1 impedem a modelagem. Para que possa-se explorar a questão com mais profundidade, dispõem-se de um segundo exemplo.

Dado um programa de desenho de formas geométricas, na qual pode-se desenhar, entre outras formas, quadrados, retângulos e círculos. No código-fonte desse programa, cada forma geométrica tem uma classe específica. Todas essas classes herdam de uma classe-mãe – *FormaGeometrica* –, conforme o diagrama de classes da Figura 4.8.

Um critério possível para determinar se duas formas geométricas são equivalentes é a congruência. Duas formas são congruentes quando têm a mesma forma e tamanho. Baseado nesse critério, pode-se concluir que um quadrado e um retângulo podem ser iguais (i.e., congruentes) em alguns cenários: um retângulo com a mesma altura e largura é congruente a um quadrado com de mesmo tamanho.

Imaginando-se que existam duas formas – *a* e *b* –, conforme de diagrama de objetos da Figura 4.9. Apesar de serem de instâncias de classes diferentes, os objetos *a* e *b* são claramente congruentes (ambos representam um quadrilátero

Figura 4.9: Diagrama de Objetos – Impacto da técnica de Subramaniam.

com quatro lados iguais). Adotando-se a técnica de Subramaniam (2020), porém, ambas formas seriam consideradas diferentes, uma vez que não pertencem a mesma classe. A propriedade de consistência seria mantida, porém a semântica de comparação não estaria condizente com a semântica esperada pelo domínio.

4.4.1.3 Classificação para abordagem para comparação de objetos de Rupakheti (2010)

Pode-se entender melhor quais os cenários as técnicas de Odersky (2010) e Subramaniam (2020) são eficientes através de uma taxonomia onde a abordagem de comparação é classificada com base no papel desempenhado pela classe dos operandos sendo comparados. Segundo Rupakheti (2010), pode-se classificar a abordagem para comparação entre objetos em três categorias:

Compatível com tipo: é “determinada através da comparação de um subconjunto comum de propriedades de dois tipos S e T , que podem ou não estar na mesma hierarquia de tipos” (RUPAKHETI, 2010, p. 6, tradução do autor) (i.e., para que dois operandos sejam considerados equivalentes, é necessário que seus estados de igualdade sejam iguais);

Incompatível com tipo: “requer que somente objetos do mesmo tipo⁷ podem ser considerados iguais, e a igualdade de um par de objetos de um mesmo tipo pode ser determinada pela comparação de um subtipo de propriedades de ambos objetos.” (RUPAKHETI, 2010, p. 6, tradução do autor) (i.e., para que dois operandos sejam considerados equivalentes, é necessário que também sejam instâncias da mesma classe concreta);

Híbrida: representa “uma [abordagem de] igualdade que mescla igualdade compatível com tipo e incompatível com tipo em uma mesma hierarquia” (RUPAKHETI, 2010, p. 8, tradução do autor)

As técnicas de Subramaniam e Odersky são eficazes quando os operandos em questão adotam a estratégia incompatível com tipo. Ambas técnicas, porém, fa-

⁷Rupakheti (2010) considera, neste contexto, tipos como classes concretas.

lham quando a estratégia compatível com tipo é adotada. Não obstante, existem diversos autores (BLOCH, 2018; ODESKY, 2010) que indicam que a igualdade compatível com tipo inerentemente viola o Princípio da Substituição de Liskov. Conforme discutido na Seção 1.5, objetos de um mesmo tipo devem ser intercambiáveis para que violações desse princípio não ocorram: uma instância de *Ponto*, por exemplo, deve ser substituível em todos os contextos por uma instância de *PontoColorido*, dado que ambos são do mesmo tipo.

Outro problema da comparação incompatível com tipo é mais sutil. Alguns *frameworks*, como o *Hibernate*, fazem modificações em classes em tempo de execução. Uma dessas modificações é a criação de *proxies* – segundo Gamma et al. (1995, p. 233, tradução do autor), “[um *proxy* tem como objetivo] prover um substituto de outro objeto de modo a controlar o acesso à ele [o outro objeto]”. *Proxies* substituem seus respectivos objetos de maneira transparente. Logo, a comparação entre eles também deve ser transparente. Como *proxies* são baseados em polimorfismo, a sua implementação faz uso de herança. Como as classes concretas dos objetos e seus *proxies* são diferentes, a comparação incompatível com tipo falha. Notavelmente, o ambiente de desenvolvimento IntelliJ avisa explicitamente o programador sobre esta questão ao gerar implementações automáticas para os métodos *equals* e *hashCode*, conforme descrito na Seção 5.5.

4.4.2 Simulação de despacho múltiplo

Conforme discutido por Bruce et al. (1995), é necessário o uso de técnicas de despacho múltiplo para que a simetria seja mantida de forma abrangente quando herança é utilizada. Apesar de existentes, as técnicas⁸ para a simulação de despacho múltiplo impactam negativamente a modularidade e a manutenibilidade de um programa (MUSCHEVICI et al., 2008). Notavelmente, essas deficiências se apresentam no nível do código-fonte e não, necessariamente, impactam o código-objeto. Deste modo, pode-se concluir que um arcabouço que isole o uso dessas técnicas do programador mitigaria as suas deficiências.

Técnicas de meta-programação são boas candidatas para a criação das fundações deste arcabouço. Segundo Flanagan e Matsumoto (2008), meta-programação é a habilidade de programas escreverem programas, o que permite, por exemplo, a geração de novos métodos em tempo de execução. Através de meta-programação, pode-se criar código para simulação de despacho múltiplo transparentemente, sem que o desenvolvedor lide com a sua inerente complexidade. A natureza dinâmica

⁸Descritas com mais profundidade na Seção 5.4.2.

da técnica mitiga a necessidade de se conhecer todos os possíveis operandos antecipadamente.

A meta-programação, porém, não é suportada numa gama grande de linguagens (Ruby é a linguagem mais popular com suporte à técnica). Pode-se, porém, usar técnicas de programação reflexiva (SOBEL; FRIEDMAN, 1996) (muito mais popular) para atingir o mesmo efeito. Ao invés de métodos e classes, criam-se objetos específicos – estratégias (GAMMA et al., 1995, p. 315) – que encapsulam a semântica de comparação. O protótipo descrito nessa dissertação (Capítulo 7) baseia-se nesta abordagem.

4.5 Conclusões do Capítulo

A primeira vista, a implementação dos métodos associados à semântica de comparação parecem simples. Essa implementação, no entanto, possui desafios. O primeiro desafio consiste no contrato implícito que determina que esses métodos sejam consistentes entre si. Falhas nesse contrato são comuns (VAZIRI et al., 2007; RAYSIDE et al., 2009; FRASER; ARCURI, 2013). A criação de um procedimento de verificação automática que valide essa consistência é impossível, uma vez que essa verificação é indecidível (HOPCROFT; MOTWANI; ULLMAN, 2006).

Ciclos representam outro desafio. Ciclos, apesar de serem considerados uma má-prática, são comumente encontrados (MELTON; TEMPERO, 2007). Implementar as operações de comparação sem levar em conta os ciclos, resultam em falhas em tempo de execução quando essas estruturas são encontradas. Ciclos podem ser tratados. Os programadores, porém, podem optar por simplesmente retirar os atributos que acarretam ciclos da comparação, o que resulta em uma modelagem incompleta da semântica de comparação.

Outro desafio encontra-se na mutabilidade. As chamadas coleções baseadas em igualdade requerem que seus itens não tenham o estado de igualdade alterado após sua inserção. Uma forma de mitigação é impedir que objetos comparáveis sejam mutáveis. Essa abordagem porém é muito restritiva e tem graves impactos à adoção de qualquer proposta. Uma forma de mitigação mais adequada seria impedir que objetos mutáveis sejam itens desse tipo particular de coleção. Tal alteração no arcabouço de coleções, porém, não faz parte do escopo deste trabalho.

A presença de herança agrava os problemas. Em linguagens baseadas em despacho simples, a herança impede a manutenção da simetria – uma propriedade mandatória das relações de equivalência. Existem técnicas que se propõem

a manter a simetria na presença da herança. Essas técnicas, porém, só são eficazes em um cenário específico, que Rupakheti (2010) denomina “incompatível com tipo”. Apenas linguagem que suportam despacho múltiplo podem manter a simetria em todos os cenários (BRUCE et al., 1995).

Técnicas de simulação de despacho múltiplo tendem a ser prejudiciais do ponto de vista de código-fonte, onde impactos a modularidade e a complexidade são observados (MUSCHEVICI et al., 2008). Esses impactos, porém, afetam somente o código-fonte (i.e., podem ser usadas em tempo de execução sem qualquer prejuízo ao código-objeto).

Estabelecido o problema, pode-se estudar os trabalhos relacionados para entender qual foi a abordagem adotada por eles, bem como suas deficiências e pontos em aberto.

5 Trabalhos Correlatos

Este trabalho está relacionado à uma série de outros trabalhos que investigam, direta ou indiretamente, a comparação de objetos em linguagens OO. Pode-se dividir esses trabalhos em cinco grandes grupos, em razão de seu foco: análise de deficiências; fundamentos; soluções completas; técnicas; e ferramentas. Outras contribuições que não se enquadram nessas categorias são apresentadas numa última seção deste capítulo.

5.1 Análise de Deficiências

Vários estudos apontam para deficiências na implementação da semântica de igualdade como uma origem significativa de defeitos. Vaziri et al. (2007) citam alguns problemas referentes à implementação da semântica de comparação em alguns programas¹. Entre esses problemas, encontram-se: violações de simetria; inconsistências entre os métodos *equals* e *hashCode*; comparação com tipos não comparáveis; e, falha no tratamento de exceções.

Rupakheti e Hou (2008) apresentam uma análise empírica sobre o estado das implementações da semântica de comparação em Java. A análise é composta de um *corpus* de quatro bases de código. Uma dessas bases é o próprio *Java Development Kit* (JDK) (versão 1.5). Curiosamente, o próprio JDK foi um dos programas com maior número de problemas relacionados à igualdade: segundo os autores, foram encontradas implementações suspeitas em cerca de 174 classes (de um total de 624).

Fraser e Arcuri (2013) descrevem um experimento onde um programa da classe dos *Search-Based Software Testing* (SBST) é avaliado. Essa avaliação se dá através de um *corpus* composto de 100 programas de código-fonte aberto escritos em Java e a verificação, entre outras, da consistência entre os métodos *equals* e *hashCode*. Violações na implementação desses métodos foram consistentemente encontradas.

¹Pertencentes ao *corpus* da suíte de testes DaCapo (BLACKBURN et al., 2006).

Mais recentemente, Silva et al. (2019) estudam se um *corpus* de programas de código-fonte aberto segue as boas-práticas definidas pelo *Hibernate*. Uma das violações mais comumente encontradas por esse estudo é diretamente ligada à como a lógica de comparação é modelada.

5.2 Fundamentos

Uma série de trabalhos acadêmicos trata de questões diretamente ou indiretamente ligadas à implementação da igualdade. As técnicas contidas nesses trabalhos não são usadas de maneira direta na proposta deste trabalho, porém servem de base para as argumentações apresentadas nesta dissertação.

CLOS provê aos programadores uma gama rica de predicados para comparação que devem ser utilizados em situações específicas. Baker (1993) propõem a inclusão do predicado EGAL à linguagem, unificando os predicados pré-existentes e evitando que o desenvolvedor tenha que escolher um deles em razão de uma situação específica (a diferença entre esses predicados é muitas vezes sutil). Apesar de não ser diretamente aplicável à linguagens OO (CLOS segue o paradigma funcional), Baker pontua a importância da uniformidade do ponto de vista de um desenvolvedor.

Liskov, Guttag et al. (1986) sugerem o uso da identidade – endereço de memória onde um objeto está armazenado – como critério único para comparação entre objetos. Apesar das vantagens como imutabilidade e mitigação de problemas nas regras de boa-formação², esta abordagem impede a implementação de qualquer noção de semântica, o que faz dessa técnica pouco indicada para objetos de domínio.

A implicação da herança em linguagens OO baseadas em despacho simples é explorada em detalhes por Bruce et al. (1995). O problema em questão é descrito de forma genérica, o exemplo dado é baseado na manutenção da simetria na comparação de objetos (inclusive o exemplo fornecido é o mesmo exemplo descrito na Seção 3.1.1). Bruce et al. descrevem algumas técnicas para a mitigação do problema. Porém, a conclusão mais importante de Bruce et al. (1995) no contexto desta dissertação é que a manutenção da simetria em linguagens OO baseadas em despacho simples é impossível na presença de herança, a não ser que técnicas como a simulação de despacho múltiplos (entre outras), sejam utilizadas.

²Endereços de memória são inteiros – elementos cujas relações de equivalência seguem as regras expostas pela Definição 1 (PEANO, 1889)

Rupakheti (2010) analisa o problema de igualdade do ponto de vista do sistema de tipos. A partir desta análise, são propostas três categorias de abordagens possíveis de comparação – compatível com tipo; incompatível com tipo; e híbrida –, descritas no Capítulo 4. Essa categorização é importante pois permite formalizar as deficiências de algumas propostas para comparação de objetos.

Bloch (2018) discute em detalhes as questões referentes à implementação dos métodos associados as operações de comparação. Além de boas práticas quanto a como esses métodos devem ser implementados em Java, é discutido também as deficiências das técnicas para manutenção da simetria quando a herança é presente. Bloch é categórico quanto a impossibilidade de se manter simetria na presença de herança (BLOCH, 2018, p. 42, tradução do autor):

“Então qual é a solução? Acontece que isso [manutenção da simetria na presença de herança] é um problema fundamental das relações de equivalência em linguagens OO. Não existe maneira de estender uma classe instanciável, adicionando algum valor e ao mesmo tempo preservar o contrato regendo *equals*.”

5.3 Soluções Completas

Existem soluções para a denotação (e cálculo) da igualdade tanto no âmbito da academia quanto da indústria.

A primeira solução é a proposta de Vaziri et al. (2007), que descreve uma extensão da linguagem Java. Essa extensão introduz um novo construto denominado tipo de relação (*relation type*, no texto original em inglês). Um tipo de relação se comporta de maneira parecida com classes: ambos são compostos de atributos e métodos. Atributos específicos podem ser marcados como chaves através da palavra-chave **key**. Esses atributos são usados em comparações (i.e., fazem parte do estado de igualdade do tipo de relação). Segundo análise empírica do autor, pode-se substituir classes por tipo de relações em cerca de 70% dos casos. Tipos de relação diferem também de classes no que tange seu ciclo de vida. No momento da sua criação, é verificado se existem tipos de relação com os mesmos valores de atributo-chave. Caso existam, a instância existente é reusada. Isso permite que a abordagem delimitada por Liskov, Guttag et al. (1986) (Seção 5.2) seja adotada e uma igualdade baseada em semântica ainda seja possível.

Rayside et al. (2009) apresentam a segunda solução analisada. Escrita em Java, essa solução permite a definição de semânticas de comparação de maneira

quasi-declarativa, através do uso de anotações – construtos da linguagem Java que permitem a inclusão de metadados arbitrários em uma classe. A solução de Rayside et al. baseia-se em Funções de Abstração: funções que mapeiam a representação concreta de uma estrutura de dados à uma representação abstrata (LISKOV; GUTTAG et al., 1986). Estruturas com comportamento semelhante têm a mesma representação abstrata (e.g. um vetor e uma lista ligada contendo os mesmos elementos têm o mesmo valor de abstração). O valor de abstração dos objetos é usado para compará-los: objetos com o mesmo valor de abstração são considerados equivalentes. Uma deficiência dessa solução é a falta de uniformidade na denotação da semântica de comparação. Quando não há uma relação de herança entre os operandos, pode-se usar anotações para determinar os atributos que compõem a função de abstração associada. Já quando existe a herança, o programador é obrigado a implementar a função manualmente, de maneira imperativa.

Grech, Rathke e Fischer (2010) apresentam a terceira solução apresentada. Assim como a solução de Rayside et al., essa solução é escrita em Java e baseia-se em uma anotação (nesse caso, porém, é permitido se denotar semântica através de anotações mesmo na presença de herança). Grech, Rathke e Fischer adotam uma abordagem baseada em *Aspect Oriented Programming* (AOP), através do uso da ferramenta Meta-AspectJ (ZOOK; HUANG; SMARAGDAKIS, 2004). Uma das técnicas introduzidas por Grech, Rathke e Fischer tange a melhoria de desempenho: a verificação de ciclos só é introduzida quando a presença de ciclos é sintaticamente possível; e, a ordem de verificação de um atributo é baseada no seu grau de diferenciação (i.e. atributos mais propensos a serem diferentes são verificados primeiro). Grech, Rathke e Fischer afirmam que ambas técnicas acarretaram uma sensível melhoria de desempenho em relação à Rayside et al. (2009).

O Projeto Lombok (ZWITSERLOOT; SPILKER, 2009) é um projeto Java de código-aberto que tem como objetivo aliviar o programador de ter que implementar *boilerplate* – o tratamento de igualdade é apenas uma das funcionalidades incluídas no projeto. O Projeto Lombok foi introduzido em 2009. No momento da escrita dessa dissertação, a última versão estável do Projeto Lombok era a versão 1.18.18, lançada em 28/01/2021. A denotação de semântica no projeto Lombok é semelhante ao modelo adotado por Rayside et al. e Grech, Rathke e Fischer: a denotação da semântica de comparação se dá de maneira declarativa, através de anotações. Existem, porém, algumas deficiências no tocante à manutenção da simetria: por adotar internamente a técnica descrita por Odersky

(2010) (descrita em detalhes na Seção 5.4), não é possível a denotação de semântica baseada na estratégia de comparação compatível com tipo.

A última solução analisada chama-se AutoValue (BOURRILLION; MCMANUS, 2014). Criado pela Google, essa solução tem como objetivo a automatização da geração de código dos chamados objetos-valor em Java. Segundo Evans (2004) “um objeto que representa um aspecto descritivo de um domínio sem uma identidade conceitual é um objeto valor” (i.e., objetos valores não têm o conceito de chaves, e são iguais quando têm as mesmas propriedades). Logo, entre os métodos necessários para se implementar um objeto-valor encontram-se os métodos *equals* e *hashCode*. Denota-se objetos-valor de maneira declarativa, através de uma anotação. Porém, o AutoValue impõe restrições importantes. A primeira tem relação à herança: AutoValue não permite que exista herança entre objetos valor. Segundo os autores, isso se dá pelo fato de “não existir uma maneira de fazê-lo [o tratamento da herança] corretamente”³. Outra questão tem relação ao estado de igualdade dos objetos-valores: de modo geral, todos os atributos são usados na comparação. É possível ignorar atributos, isso envolve, porém, a escrita de código não-declarativo.

5.3.1 Análise Geral das Soluções

Através da análise das soluções, pode-se chegar às seguintes conclusões, resumidas pela Tabela 5.1.

Todas as soluções analisadas são declarativas e baseadas em Java. Somente Vaziri et al. (2007) definem uma outra linguagem – uma extensão de Java imbuída do conceito de tipo de relações. Todas as soluções analisadas, tirando Rayside et al. (2009) e AutoValue, são uniformes no tocante à denotação da semântica de comparação. A solução de Rayside et al. falha na questão da uniformidade em razão da geração automática de funções de abstração só ser possível quando a classe em questão não faz uso de herança. Neste caso, o programador tem que escrever a função de abstração manualmente (o que incorre nas mesmas dificuldades de implementação já discutidas no Capítulo 3). Já AutoValue não permite que seja usada uma abordagem declarativa para indicar quais atributos não pertencem ao estado de igualdade de um objeto-valor.

Com exceção de Lombok e AutoValue, todas as soluções apresentada tratam ciclos de maneira transparente. No caso de Vaziri et al. (2007), a comparação pelo

³Conforme descrito em <https://github.com/google/auto/blob/auto-value-1.8/value/userguide/howto.md#inherit>.

Trabalho relacionado	Denotação de semântica uniforme	Restrições à mutabilidade	Mantém a simetria	Trata ordem	Trata <i>hash</i>	Trata ciclos	Trata estratégia compatível com tipo
Vaziri et al. (2007)	Sim	Sim	Sim	Não	Sim	Sim	Não
Rayside et al. (2009)	Não ¹	Não	Sim	Não	Sim	Sim	Sim ⁵
Grech, Rathke e Fischer (2010)	Sim	Sim ²	Sim	Não	Sim	Sim	Não
Projeto Lombok (ZWITSERLOOT; SPILKER, 2009)	Sim	Não	Sim	Não	Sim	Não	Não
AutoValue (BOURRILLION; MCMANUS, 2014)	Sim	Sim ³	Não ⁴	Não	Sim	Não	Não

¹Quando herança é utilizada, o programador deve manualmente escrever uma função de abstração.

²Itens não podem mudar seu estado de igualdade após a primeira chamada ao método *hashCode*.

³Trata somente de objetos valor (EVANS, 2004), que por definição são imutáveis.

⁴Não suporta herança como um todo.

⁵É possível caso o programador implemente a lógica de comparação manualmente, encapsulada em uma função de abstração.

Tabela 5.1: Comparativo entre as soluções completas.

endereço de memória mitiga as referências circulares. Já Rayside et al. (2009) e Grech, Rathke e Fischer (2010) adotam a mesma técnica para tratamento de ciclos, baseada na linguagem Eiffel (MEYER, 1992). O tratamento em Rayside et al., porém, tem uma deficiência que impacta diretamente as funções *hash*: uma constante é retornada quando ciclos são detectados. Apesar de válida, essa abordagem degrada sensivelmente o desempenho, uma vez que gera funções *hash* com alto grau de colisão.

Somente os trabalhos de âmbito acadêmico têm métodos de avaliação bem-definidos. Do ponto de vista dos aspectos avaliados, todas as propostas adotaram métodos *ad-hoc*⁴ (i.e., não é usada nenhuma norma ou métrica pré-estabelecida na análise de impactos). Já do ponto de vista de *corpora*, Vaziri et al. baseiam-se no *corpus* da Suíte de Teste DaCapo (BLACKBURN et al., 2006) (*dacapo-2006-10*) enquanto Rayside et al. e Grech, Rathke e Fischer não usam *corpora* pré-definidos: o primeiro baseia-se em três programas – *JFreeChart* (v1.0.0), *Java Collection Framework* (JCF) e *Apache Commons Collections* (v3.2) – enquanto o segundo avalia os impactos apenas no *JFreeChart*.

Apenas Vaziri et al. (2007) e Grech, Rathke e Fischer (2010) impõem restrições quanto a mutabilidade dos objetos comparáveis. Os tipos de relação introduzidos por Vaziri et al. são totalmente imutáveis. Já Grech, Rathke e Fischer adotam uma abordagem mais leniente quanto a mutação: atributos pertencentes ao estado de igualdade de um objeto podem ser alterados até o método *hashCode* ser invocado. Após disso, alterações resultam em falhas em tempo de execução.

Nenhuma das soluções apresentadas pela academia ganhou tração na indústria. No campo das soluções da indústria, o projeto Lombok tem o grau de aceitação expressivo: seu repositório⁵ no Github tem, no momento da escrita desta dissertação, cerca de 704 mil usuários registrados.

Nenhuma das soluções aqui analisadas, incluindo o projeto Lombok, tem como escopo o tratamento da ordem. No que diz respeito à igualdade compatível com tipo, somente Rayside et al. (2009) provê um tipo limitado de suporte (no caso, o programador é obrigado a escrever uma função de abstração de forma imperativa).

⁴Mais detalhes sobre esses aspectos são descritos na Seção 8.1.

⁵<https://github.com/rzwitserloot/lombok>.

5.4 Técnicas

Nessa seção, são discutidas técnicas que podem ser usadas na mitigação dos problemas relatados no Capítulo 3. Ao contrário das propostas discutidas na Seção 5.3, as técnicas aqui apresentadas não têm como foco a resolução do problema da igualdade maneira holística, e sim facilitar na implementação de maneira pontual.

Pode-se dividir as técnicas abordadas em duas categorias: técnicas para manutenção da simetria; e, técnicas para simulação de despacho múltiplo.

5.4.1 Técnicas para manutenção da simetria

Tanto Odersky (2010) quanto Subramaniam (2020) delineiam técnicas (apresentadas na Seção 4.4.1.1) que têm como objetivo manter a simetria durante a comparação de objetos.

A técnica delineada por Odersky (2010) consiste na inclusão de um novo método – *canEqual* – cujo papel é verificar se os operandos são comparáveis (i.e. se existem casos onde instâncias de suas classes são consideradas equivalentes). Toda classe que estenda a semântica de comparação deve estender o método *canEqual* de modo consistente.

A técnica descrita por Subramaniam (2020) opera de maneira semelhante. Ao invés da inclusão do método *canEqual*, é incorporada no método *equals* uma checagem extra que obriga os operandos serem instâncias da mesma classe concreta para que sejam considerados iguais.

Ambos métodos têm deficiências. Em primeiro lugar, eles não são eficazes na manutenção da anti-simetria na verificação da ordem, uma vez que relação de anti-simetria implica uma relação bidirecional inversa entre os resultados (se $x < y$ então $y > x$). Outra desvantagem – específica à técnica de Odersky (2010) – reside no aumento da possibilidade da existência de inconsistências entre as operações de comparação, em razão da inclusão do método *canEqual*. Não obstante, as técnicas apresentadas também introduzem⁶ uma violação do Princípio da Substituição de Liskov.

A maior desvantagem dessas técnicas, porém, é que elas impossibilitam a modelagem de algumas semânticas de comparação válidas: é impossível, adotando ambas as técnicas, a modelar a estratégia de comparação compatível por tipo.

⁶Bloch (2018, p. 43) discute uma técnica análoga à técnica de Subramaniam (2020) e aponta para a violação do Princípio da Substituição de Liskov. Odersky (2010, p. 582) reconhece prontamente o mesmo problema em sua técnica.

Essa deficiência é pontuada por Odersky (2010, p. 582, tradução do autor):

“O problema em escrever uma implementação de *equals* que compara as classes em tempo de execução não é a violação do Princípio da Substituição de Liskov, mas que não existe uma maneira de se criar uma subclasse cujas instâncias são iguais a instâncias da sua superclasses.”

5.4.2 Técnicas para a simulação de despacho múltiplo

A segunda categoria de técnicas analisada compreende a simulação de despacho múltiplo em linguagens OO baseadas em despacho simples. Entre essas técnicas, encontra-se a técnica descrita por Ingalls (1986), onde o despacho múltiplo é transformada em uma série de chamadas usando despacho simples.

O padrão de projetos *Visitor* também pode ser utilizado para a simulação de despacho múltiplos em uma linguagem baseada em despacho simples. Segundo Gamma et al. (1995, p. 366, tradução do autor), “[um *Visitor*] representa uma operação a ser executada em elementos de uma estrutura de objetos”.

Apesar da simulação de despacho múltiplo em linguagens baseadas em despacho simples ser possível, as técnicas para sua execução têm deficiências importantes, conforme discutido por (MUSCHEVICI et al., 2008, p. 565, tradução do autor).

“Comparada à declaração direta de multi-métodos⁷, ambos idiomas para [simulação de] despacho duplo são tediosos de escrever e propensos a erros. Códigos que despacham baseado em três ou mais parâmetros são particularmente difíceis. A modularidade é comprometida, já que todas as classes devem ser modificadas quando uma nova é introduzida, seja pela escrita de novos métodos de despacho ou novos *ifs* encadeados”

Notavelmente, as deficiências discutidas nos métodos de simulação de despacho múltiplo afetam apenas o código-fonte. Do ponto de vista de código-objeto, as técnicas aqui discutidas são robustas.

5.5 Wizards de IDEs

Os *wizards* – padrão de interface onde uma atividade é guiada por uma sequência de telas cuja navegação é limitada ao passo anterior e subsequente (WINCKLER;

⁷De acordo com Muschevici et al. (2008), denomina-se métodos que usam despacho múltiplo de multi-métodos.

BERNHaupt; PONTICO, 2010) – nos ambientes de desenvolvimento de mercado também tentem mitigar os problemas relacionados à definição da semânticas de comparação. Foram analisados os *wizards* embarcados em três IDEs – *IntelliJ*⁸, *Netbeans*⁹ e *Eclipse*¹⁰ – populares da linguagem Java.

Os *wizards* embutidos nas IDEs operam de maneira semelhante: o programador escolhe os atributos a serem comparados e, ao final do processo, a IDE gera o código fonte correspondente para os métodos *equals* e *hashCode* (nenhum dos *wizards* gera código para o método *compareTo*).

Todos os *wizards* analisados são capazes de gerar código baseado na estratégia de comparação incompatível com tipo (RUPAKHETI, 2010). Os códigos gerados nos três casos são semelhantes, adotando a técnica de manutenção de simetria descrita por Subramaniam (2020). O *IntelliJ* e o *Eclipse* permitem também a geração de código baseada na comparação compatível com tipo. Nesse caso, o código gerado por ambas viola a propriedade de simetria. O *wizard* do *IntelliJ* explicitamente atenta para esse fato, conforme exposto na Figura 5.1.

Figura 5.1: Alerta sobre falha na manutenção de contratos (*IntelliJ*).

Um ponto curioso é que os *wizard* embutidos no *IntelliJ* e no *Netbeans* permitem que seja escolhido atributos diferentes para geração de código dos métodos *equals* e *hashCode*, o que poderia gerar implementações inconsistentes entre si. Essa situação não é permitida pelo *wizard* no *Eclipse*.

5.6 Outras contribuições

Nelson, Pearce e Noble (2010) estudam o impacto da implementação da lógica de comparação no uso de estruturas de dados denominadas coleções baseadas em igualdade. Esse tipo de coleção exige que o estado de igualdade de seus itens nunca mude após a sua inserção. Observações empíricas dos autores apontam

⁸Versão 2020.3 (community), disponível em <https://www.jetbrains.com/idea/>

⁹Versão 12.3, disponível em <https://netbeans.apache.org/>

¹⁰Versão 2020-09, disponível em <https://www.eclipse.org/ide/>

que 19% dos objetos incluídos em coleções mudam seu estado de igualdade após a sua inserção. Isso é uma potencial fonte de defeitos em tempo de execução, caso o programador mude uma implementação de coleção para uma coleção baseada em igualdade.

A *ISO C++20* (2018) introduz o operando $\leq$ à linguagem C++. A semântica do operador $\leq$ é a mesma do seu equivalente na linguagem Ruby, discutido na Seção 3.1.3: o operador $\leq$ – operador *spaceship* – modela a Lei da Tricotomia. Da mesma forma, é possível derivar as outras operações¹¹ de comparação a partir de $\leq$, mitigando assim possíveis inconsistências.

5.7 Conclusões do Capítulo

A implementação da semântica de igualdade é difícil. Diversos estudos (VAZIRI et al., 2007; FRASER; ARCURI, 2013; RUPAKHETI; HOU, 2008; RAYSIDE et al., 2009; SILVA et al., 2019) apontam para problemas na sua implementação em *corpora* compostos de programas de diversões níveis de qualidade e popularidade. Essas deficiências são encontradas em classes pertencentes ao JDK, inclusive.

A manutenção da simetria na presença de herança é uma questão perene. Técnicas foram desenvolvidas com o intuito de mitigar a questão. Essas técnicas, porém, não são eficazes quando deseja-se modelar uma semântica de comparação compatível com tipo. A simetria, de maneira mais abrangente, só pode ser mantidas em linguagens baseadas em despacho múltiplo. As técnicas de simulação de despacho múltiplo, infelizmente, tem impactos negativos na modularidade.

Tanto a academia quanto a indústria propuseram soluções para o tratamento de igualdade de maneira holística. De maneira geral, essas propostas adotam um paradigma declarativo, onde a semântica de comparação é informada através de metadados. É gerado código imperativo para os métodos relacionados à operações de comparação através desses metadados. Esses métodos são assim corretos por construção e a falta de consistência entre eles é mitigada.

Das soluções ditas completas, apenas o projeto Lombok ganhou alguma tração. Apesar de eficiente, o projeto Lombok é baseado em técnicas que notoriamente falham ao modelar semântica de comparação compatível com tipo. Não obstante, o projeto Lombok não trata o aspecto de ordem da comparação.

Curiosamente, a questão da ordenação entre objetos é ignorada. Somente

¹¹C++ define seis operadores relativos à operações de comparação: $==$, $!=$, $<$, $<=$, $>=$, e $>$.

C++20 trata desta questão ao introduzir o operador $\leq$. Todas as outras propostas, sejam no âmbito acadêmico ou da indústria, ignoram esse aspecto.

Após a análise dos problemas comumente encontrados na implementação da semântica de igualdade em linguagens OO (Capítulo 4) e os trabalhos correlatos apresentados aqui, pode-se delinear uma proposta de solução que tenha como objetivo sanar as deficiências encontradas nas propostas existentes. Essas deficiências são, principalmente: o suporte à ordenação dos objetos e a estratégia de comparação compatível com tipos.

6 Proposta

Tendo em vista os trabalhos descritos no Capítulo 5, pode-se notar que nenhuma das soluções apresentadas trata a questão da igualdade de maneira holística, uma vez que a questão da ordem não é abordada por nenhuma proposta. Outra deficiência recorrente é a falta de suporte para a chamada comparação compatível com tipo (RUPAKHETI, 2010).

Este capítulo compreende a descrição (informal e formal) da proposta para a computação das operações de comparação apresentada nesta dissertação. Ao contrário dos trabalhos relacionados, a proposta deste trabalho trata da ordem de maneira holística, incluindo os três aspectos da semântica de comparação: verificação de igualdade, ordem e código *hash*. Essa descrição se dá de maneira agnóstica à linguagem e à implementação. Isto permite que a solução seja utilizada em linguagens já existentes (como bibliotecas ou *frameworks*) ou em construtos de novas linguagens.

Antes da apresentação da descrição formal da solução, será apresentada uma descrição informal, cujo principal objetivo é dar ao leitor um entendimento prévio, o que facilita o entendimento das complexidades intrínsecas à descrição formal. Também são descritas algumas características desejáveis, baseadas nos trabalhos analisados no Capítulo 5.

6.1 Características Desejáveis

Conforme discutido anteriormente, as propostas existentes para o problema em questão têm como principal deficiência ignorar o aspecto de ordem da semântica de comparação. Apesar dessa deficiência, essas propostas têm características benéficas que podem ser aproveitadas pela proposta deste trabalho. Essas características são:

Denotação declarativa: conforme discutido na Seção 4.1, a semântica de comparação é distribuída em múltiplos métodos que devem ser consistentes

entre si. Uma forma de manter a consistência é gerar métodos corretos por construção através de metadados indicando a semântica de comparação desejada. Todas as soluções analisadas na Seção 5.3 seguem essa abordagem. A proposta aqui descrita também seguirá essa abordagem;

Uniformidade: a falta de uniformidade na forma de comparar valores dificulta o trabalho dos programadores (BAKER, 1993). Das soluções analisadas na Seção 5.3, apenas a de Rayside et al. (2009) não é uniforme na definição da semântica de comparação. A proposta descrita neste trabalho têm que ser uniforme;

Sem restrições à mutabilidade: conforme discutido, restrições na mutabilidade têm sua raiz em características intrínsecas a um tipo específico de coleções – coleções baseadas em igualdade (NELSON; PEARCE; NOBLE, 2010). Argumenta-se, porém, que tais restrições devam ser impostas pelas coleções a seus itens e não compulsórias à todo e qualquer objeto comparável. Desse modo, a proposta aqui apresentada não impõe nenhuma restrição de mutabilidade aos objetos comparáveis;

Expressividade algumas das soluções apresentadas (VAZIRI et al., 2007; ZOOK; HUANG; SMARAGDAKIS, 2004; BOURRILLION; MCMANUS, 2014) só conseguem manter a simetria na presença de herança quando uma estratégia específica de comparação é utilizada – estratégia incompatível com tipo (RUPAKHETI, 2010). A mitigação dessa deficiência é possível através do uso do despacho múltiplo ou de técnicas com o mesmo grau de dinamismo (BRUCE et al., 1995). A solução aqui apresentada deverá permitir a modelagem de comparações compatível com tipo.

6.2 Descrição Informal

A solução aqui apresentada divide o processo de computação das operações de comparação em duas fases. A primeira fase é a fase de construção dos registros de comparação – elementos que encapsulam o contexto onde uma operação é avaliada. Esses registros são avaliados na segunda fase, chamada fase de avaliação.

6.2.1 Registros de Comparação

Registros de comparação são a espinha dorsal da solução. Eles são baseados no cálculo de registros de Cardelli e Mitchell (1991) e provém o contexto no

qual as operações de comparação são computadas. Como contexto, entende-se os operandos a serem comparados.

Como qualquer registro, um registro de comparação é um conjunto de campos unicamente identificados que armazenam valores. Esses valores podem ser acessados através de operações de extração (e.g. a expressão $r.x$ resulta no valor associado ao campo x do registro de comparação r). No caso dos registros de comparação, porém, os valores associados aos campos seguem um padrão distinto. Dois componentes – $o1$ e $o2$ – constituem esses valores. Eles armazenam os valores associados, respectivamente, com o primeiro e o segundo operando.

Outra diferença importante é como os campos são percorridos. Registros simples não definem uma ordem específica para iteração. No caso de registros de comparação, porém, é obrigatório que a iteração seja feita respeitando a ordem de avaliação de seus campos.

6.2.2 Fase de Construção

Dois passos compõem o processo de construção de registro de comparação. Primeiramente, coleta-se os campos que fazem parte do estado de igualdade dos operandos¹. Subsequentemente, obtêm-se os valores associados a cada campo em ambos operandos. Por exemplo, os operandos $a : Ponto$ e $b : Ponto$ resultam em um registro de comparação contendo os campos x e y , que, por sua vez, são atributos de *Ponto*.

Um aspecto que vale uma digressão refere-se ao mecanismo para denotação de quais campos compõe o estado de igualdade de um objeto (i.e., a semântica de comparação de um objeto). Conforme discutido anteriormente (Seção 6.1), tais campos devem ser denotados de maneira declarativa. Diferentes linguagens, porém, têm diferentes métodos para inclusão de metadados. Java, por exemplo, usa anotações (tanto Rayside et al. (2009) quando Grech, Rathke e Fischer (2010) as usam em suas propostas). O conceito de anotações, porém, não existe em Ruby (normalmente usa técnicas de meta-programação que permitem mudar os programas em tempo de execução). Uma nova linguagem (como por exemplo a extensão de Java proposta por VAZIRI et al.), pode criar novas palavras chaves para transmitir a mesma informação. Desta forma, optou-se por não descrever o mecanismo de codificação desses metadados, de modo a não prejudicar o agnos-

¹O cálculo do código *hash* é uma exceção interessante, uma vez que o procedimento de construção de registros de comparação precisa de dois operandos (e a operação de cálculo de *hash* tem apenas um). Nesse caso, usa-se o único operando da operação de cálculo de *hash* como primeiro e segundo operando na invocação do procedimento de construção.

ticismo da solução deste trabalho.

Independentemente da forma de codificação, pode-se definir o que deve ser incluído. Para que se possa denotar semântica de comparação, é necessário informar:

- Quais campos pertencem ao estado de igualdade de um objeto;
- A ordem na qual esses campos são avaliados.

O cenário no qual existe uma relação de subtipo entre os operandos é bem mais interessante. Considerando os operandos $a : \text{Ponto}$ e $b : \text{PontoColorido}$, pode inferir que, além dos campos x e y definidos em Ponto ², o campo cor também deve aparecer no registro de comparação resultante. Isso, porém, leva a um impasse: não se pode obter o valor de cor em a uma vez que esse valor simplesmente não existe.

Esse impasse é resolvido através da introdução de um novo valor – *desconhecido* – que denota a ausência de valor. Sua semântica é similar a do valor *unknown* da Lógica K_3^S de Kleene (KLEENE et al., 1952): *desconhecido* é considerado o diferente de qualquer outro valor (inclusive de si mesmo) e é sempre o menor de dois valores. As Tabelas³ 6.1 e 6.2 sumarizam sua semântica de comparação.

$A = B$	A			
	$\top$	$\sqcup$	$\perp$	
$\top$	$\top$	$\perp$	$\perp$	
B $\sqcup$	$\perp$	$\perp$	$\perp$	
$\perp$	$\perp$	$\perp$	$\top$	

Tabela 6.1: Semântica de equivalência de *desconhecido*.

$A \Leftrightarrow B$	A			
	$\top$	$\sqcup$	$\perp$	
$\top$	0	1	1	
B $\sqcup$	-1	-1	-1	
$\perp$	-1	1	0	

Tabela 6.2: Semântica de ordem natural de *desconhecido*.

Voltando ao exemplo anterior, o registro resultante r tem três campos – x , y e cor . Nesse caso $r.cor.o_2$ armazena o valor de cor associado ao segundo operando da operação (no caso b), enquanto $r.cor.o_1$ armazena o valor *desconhecido*, uma vez que o campo cor não existem em a .

²A relação de herança entre as classes Ponto e PontoColorido implica que instâncias da última também tenha os campos definidos na primeira. Do mesmo modo, essa relação implica que os campos pertençam ao estado de igualdade de Ponto também pertençam ao estado de igualdade de PontoColorido .

³Considera-se, nessas tabelas, que tanto $A \in U$ quanto $B \in U$, sendo $\mathbb{U} = \{\top, \perp, \sqcup\}$. Nesse contexto, os operadores $=$ e $\Leftrightarrow$ são, respectivamente, funções $\mathbb{U} \rightarrow \mathbb{B}$ e $\mathbb{U} \rightarrow \mathbb{Z}$.

6.2.3 Fase de Avaliação

O próximo passo depois da contração dos registros é sua avaliação. Funções especializadas computam a avaliação. Cada uma das operações de comparação – igualdade, ordem e *hashing* – tem uma função específica.

A checagem da igualdade é feita ao iterar por cada um dos campos em um registro de comparação, verificando se seus valores (o_1 e o_2) são diferentes. Caso o sejam, o processo termina e os operandos são considerados diferentes. Os operandos são considerados iguais caso nenhuma discrepância seja encontrada.

A ordem segue um processo similar. Itera-se por todos os campos, checando se um dos valores é maior que o outro (se $o_1 < o_2$ ou $o_2 < o_1$). Neste caso, o operando associado ao maior valor é considerado maior e o processo para. Caso todos os campos sejam percorridos sem que sejam encontradas discrepâncias, os operandos são considerados iguais.

Como os outros algoritmos, o cálculo de *hash* também percorre os campos de um registro de comparação: cada campo tem o código *hash* de seu valor calculado e esse valor é agregado a um valor único (que no fim do processo é retornado como o código *hash* do registro de comparação como um todo).

Não faz parte do escopo desta dissertação discorrer sobre melhor forma de agregação de códigos *hashes*. De todo modo, as regras definidas na Seção 6.3.4 dependem que uma única regra seja observada: dada uma função de agregação de códigos *hash* $\oplus$, $(\mathbb{Z}, \oplus)$ deve formar um semi-grupo.

6.2.4 Lidando com Ciclos

Conforme discutido na Seção 4.2, ciclos ocorrem quando objetos em um grupo são capazes de se referenciar. Iterações nesses objetos podem encontrar laços fechados, o que termina por acarretar estouro de pilhas (ou em um *loop* infinito, se a linguagem em questão suportar recursão em cauda). Ciclos são considerados uma má-prática (PARNAS, 1979). Todavia, essas estruturas são comumente encontradas (MELTON; TEMPERO, 2007).

Um mecanismo de detecção de ciclos é uma escolha de implementação. Sendo assim, sua escolha afetaria o agnosticismo da solução. De qualquer modo, pode-se definir como a solução deve se comportar quando ciclos são encontrados sem perigo de enviesamento.

Para as operações de comparação, a solução deve simplesmente ignorar os

ciclos encontrados. No caso das operações de ordem e comparação, ignorar significa simplesmente não checar os elementos incluídos no ciclo. Já caso do cálculo de código *hash*, assume-se uma constante específica para qualquer estrutura que seja um ciclo.

Figura 6.1: Diagrama de Objetos – Tratamento de Ciclos.

Pode-se explicar melhor o racional dessa decisão através de um exemplo. Dado os objetos, *a* e *b*, estruturados conforme a Figura 6.1. Seguindo a semântica delineada na Seção 4.2, pode-se inferir que a comparação entre *a* e *b* resulta em um *loop* infinito, uma vez que a sequência de operações $(a = b) \vdash (c = d) \vdash (a = b) \vdash (c = d) \dots$ é executada indefinidamente. A comparação entre *a* e *e*, por sua vez, atinge a conclusão que $a \neq e$, uma vez que um dos objetos pertencentes à associação *produtosFabricados* em *e* difere de *a*: no caso, os objetos *d* e *f* têm nomes distintos (*foo* e *bar*, respectivamente) e por essa razão são considerados diferentes.

A linguagem Eiffel (MEYER, 1992) segue uma abordagem semelhante: os objetos são primeiramente considerados iguais e os seus campos são iterados em paralelo, em busca de evidências que provem o contrário. Como nenhuma nova evidência de disparidade entre os operandos pode ser obtida dentro de um ciclo, pode-se ignorar o ciclo como um todo.

Através desses dois casos, pode-se chegar a conclusão que ciclo só incorre em um *loop* infinito caso não existam discrepâncias nos elementos dentro desse ciclo. Essa conclusão vem do fato que, caso existissem tais discrepâncias (como por exemplo, no caso de $a = e$, onde um dos produtos fabricados é diferente), o processamento seria prontamente interrompido, dada as características da avaliação de curto-circuito⁴. Desse modo, conclui-se que, no que toca as operações de comparação, pode-se ignorar todo ciclo formado.

⁴Também chamada de Avaliação de McCarthy, a avaliação de curto-circuito especifica que o segundo argumento de uma expressão *booleana* só deve ser avaliado se o primeiro argumento não for suficiente para determinar o valor da expressão. Por exemplo dada a expressão `firstArgument() && secondArgument()` em Java: o método `secondArgument()` só é invocado se o resultado de `firstArgument()` for *true*. O racional é que, caso o resultado de `firstArgument()` seja *false*, o valor da expressão como um todo será sempre *false*, independentemente do valor retornado por `secondArgument()`. Logo `secondArgument()` não precisa ser invocado.

Construto	Descrição
$\diamond$	Denota um registro vazio.
$r = \langle x = v \rangle$	Cria um registro r com o valor v associado ao campo x .
$r_2 = \langle r \mid x = v \rangle$	Adiciona o valor v associado ao campo x do registro r , retornando um novo registro r_2 .
$r.x$	Obtém o valor associado ao campo x no registro r .
$r_2 = r \setminus x$	Remove o campo x do registro r , retornando um novo registro r_2 .
$C = A \setminus B$	Remove todos elementos contidos em B do conjunto A , obtendo ao final o conjunto C (e.g., $\{1, 2, 3\} \setminus \{1, 4\} = \{2, 3\}$).
$\bar{x}$	Abreviação de sequências do tipo $x_0 \dots x_n$.
$\text{class } C \triangleleft D \{ \bar{C} \bar{f}; K \bar{M} \}$	Define uma classe C que estende a classe D . C é composta dos campos $\bar{f}$, do construtor K e dos métodos $\bar{M}$.
$o : C$	Representa um objeto o que é instância de C
$fields(C)$	Obtém todos os campos definidos em C (e nas suas super-classes).
$es_fields(C)$	Obtém todos os campos que fazem parte do estado de igualdade de C (e de suas super-classes).
$has_cycle(o_1, o_2)$	Retorna $\top$ caso exista um ciclo referente ao processamento de o_1 e o_2 . Retorna $\perp$ caso contrário.
$\top$	Representa o valor booleano <i>verdadeiro</i> .
$\perp$	Representa o valor booleano <i>falso</i> .
$\sqcup$	Representa um valor <i>desconhecido</i> (conforme descrito na Seção 6.2.2).

Tabela 6.3: Sumário da Notação Utilizada.

6.3 Descrição Formal

A descrição formal apresentada segue a Semântica Operacional Estruturada. Introduzida por Plotkin (1981), a motivação para o desenvolvimento de tal técnica foi o desenvolvimento de um método simples e direto para a especificação da semântica de linguagens de programação. O uso da semântica operacional permite a descrição de comportamento de maneira formal, mantendo o agnosticismo de linguagem de programação.

6.3.1 Notação e Formalismos Utilizados

Antes de se discutir os detalhes intrínsecos à descrição formal da proposta. É importante se explicar a notação utilizada e introduzir os formalismos utilizados para tal formalização. A Tabela 6.3 sumariza a notação, bem como as principais construções utilizadas na formalização de cada um dos formalismos.

6.3.1.1 Semântica Operacional Estruturada

Uma especificação baseada em semântica operacional estruturada define um programa através de uma série de transições denominadas relações de transições. As transições válidas são expressas através de regras de inferência.

$$\begin{array}{c}
 \text{Nome} \quad \text{Premissa 1} \quad \text{Premissa 2} \quad \dots \quad \text{Premissa n} \\
 \hline
 \text{EXEMPLO} \quad \frac{a_1 \rightarrow \top \quad a_2 \rightarrow a_1 \quad \dots \quad a_n \rightarrow a_{n-1}}{a_n \rightarrow \top} \\
 \text{Conclusão}
 \end{array}$$

Figura 6.2: Exemplo de Notação – Semântica Operacional.

A Figura 6.2 exemplifica uma regra de inferência. A informação mais à esquerda é o nome da regra. Cada regra é identificada unicamente por seu nome (o nome da regra no exemplo é EXEMPLO). Na parte superior direita encontram-se as premissas. Uma regra pode possuir quantas premissas forem necessárias. Na parte inferior encontra-se a conclusão da regra. A conclusão de uma regra é válida apenas quando todas as premissas são verdadeiras.

6.3.1.2 Cálculo de Registro

Além da Semântica Operacional, também é utilizado⁵ na formalização da proposta deste trabalho o cálculo de registros, conforme descrito por Cardelli e Mitchell (1991). O cálculo de registro provê os construtos para construção e manipulação de registros. Registros são especialmente importantes por serem a base sob a qual um construto utilizado na comparação de objetos – os registros de comparação – são construídos.

Segundo Cardelli e Mitchell (1991, p. 3, tradução do autor), “um registro é uma coleção de campos, onde cada campo é um valor rotulado”. Cada rótulo é único dentro de um registro e assim, pode ser utilizado para identificar campos. Registros são descritos seguindo a seguinte notação: $\langle l_0 = v_0, l_1 = v_1, \dots, l_n = v_n \rangle$, onde l representa um rótulo e v um valor associado. Valores podem, por sua vez, ser registros. O registro $\diamond$ denota um registro vazio.

O cálculo de registro define três operações básicas: extensão, restrição e extração. A extensão permite que novos campos sejam adicionados a um registro e segue o modelo $r_2 = \langle r \mid l = v \rangle$, onde é criado um novo registro r_2 , composto de todos os campos contidos em r , acrescido do campo l armazenando o valor v .

⁵É importante notar que somente os conceitos do cálculo de registro estritamente utilizado pela proposta deste trabalho são explicados nessa seção.

A restrição remove campos arbitrários de um registro e segue a notação $r_2 = r \setminus l$ (no caso, é criado um novo registro r_2 , composto de todos os campos contidos em r , exceto o campo l). A extração, por sua vez, permite a obtenção de um valor – projeção – associado a um rótulo dentro de um registro. Já a extração segue a notação $r.l$ (no caso, se deseja extrair o valor de l do registro r).

6.3.1.3 Featherweight Java

O terceiro e último formalismo utilizado é o *Featherweight Java* (FJ). Segundo Igarashi, Pierce e Wadler (2001), o FJ formaliza um subconjunto da linguagem Java e tem como propósito ser um cálculo minimalista para a modelagem de sistema de tipos da linguagem Java.

Ser baseado em Java, pode, à primeira vista, fazer sua adoção questionável, uma vez que se almeja a definição de uma solução que seja agnóstica à linguagem. É importante ressaltar, porém, que o uso da FJ não afeta o agnosticismo da proposta, uma vez que os construtos utilizados são muitos similares que seus correspondentes em outras linguagens OO.

O primeiro construto definido pelo FJ utilizado é a classe. Uma classe em FJ é dada por uma expressão na forma de $class\ C \triangleleft D\ \{\ \overline{C}\ \overline{f};\ K\ \overline{M}\}$, onde C representa uma classe que estende a classe D e é composta dos campos $\overline{f}$, do construtor K e dos métodos $\overline{M}$. O segundo construto utilizado é representado pela função *fields*, que tem como papel a obtenção de todos os campos definidos por uma classe. Ou formalmente, dada uma classe $C \triangleleft D\ \{\ \overline{C}\ \overline{f};\ K\ \overline{M}\}$, $fields : C \mapsto ((\overline{C}\ \overline{f}) \cup (fields(D)))$.

6.3.2 Registros de Comparação

Tendo-se definido os formalismos e notação, pode-se partir para a formalização propriamente dita. A primeira formalização corresponde aos registros de comparação, apresentada na Definição 4, conforme informalmente discorrido na Seção 6.2.1.

Definição 4 (Registro de Comparação). *Um registro de comparação r é um registro $\langle f_0 = \langle o_1 = x.f_0, o_2 = y.f_0 \rangle, f_1 = \langle o_1 = x.f_1, o_2 = y.f_1 \rangle \dots f_n = \langle o_1 = x.f_n, o_2 = y.f_n \rangle \rangle$. Para cada campo $f_i = \langle o_1 = x.f_i, o_2 = y.f_i \rangle$, o_1 armazena o valor associado com o primeiro operando enquanto o_2 o valor associado com o segundo operando (x e y , respectivamente). A iteração dos campos $\overline{f}$ segue a ordem $(f_0, f_1 \dots f_n)$.*

Campos de registros de comparação podem armazenar o valor $\sqcup$. Conforme discutido na Seção 6.2.2, qualquer registro de comparação que tenha um dos componentes de seus campos é considerado diferente (no que tange a ordem, $\sqcup$ é sempre o menor valor). A Definição 5 formaliza a semântica de comparação de $\sqcup$.

Definição 5 (Semântica de comparação de $\sqcup$). *O símbolo $\sqcup$ denota um valor desconhecido. O valor desconhecido é considerado diferente de qualquer outro valor (inclusive de si mesmo). Logo, $(\forall x)(x \neq \sqcup)$. No que tange a ordem de dois valores, $\sqcup$ sempre será considerado o menor⁶ valor. Logo, $(\forall x)(x > \sqcup)$ e $(\forall x)(\sqcup < x)$ são ambos válidos.*

6.3.3 Fase de Construção

Conforme discutido anteriormente, a primeira fase do processo de avaliação de operações de comparação é a fase de construção de registros comparação, cuja função *build* é a porta de entrada. A função *build* recebe os dois operandos da operação a ser computada e retorna o registro de comparação correspondente. A regra CR-BUILD formaliza a função *build*. Como retratado na Figura 6.3, CR-BUILD usa duas funções: *es_fields* e *cpBuild*.

A função *es_fields* merece um destaque. Dado um operando, essa função retorna os campos que pertencem a seu estado de igualdade. É importante ressaltar, porém, que o modo de operação dessa função não é formalizado. Não obstante, nem o cálculo de registro e nem o FJ as definem. Conforme explicado anteriormente (Seção 6.2), esses campos são obtidos através de metadados que denotam a semântica de comparação. Para evitar que a proposta em questão tenha qualquer tipo de viés, se optou por não formalizar a estrutura desse meta-dado. Desse modo, a função *es_fields* deve ser tratada como um caixa preta: dada uma classe C , *es_fields* retorna os campos $\bar{f}$ que compõem o estado de igualdade de C , ordenados pela sua ordem de avaliação.

Já a função *cpBuild* constrói o registro de comparação de maneira recursiva. Três regras formalizam a função *cpBuild*: CR-EMPTY, CR-MULTIPLE e CR-SINGLE. A regra CR-EMPTY é o caso base da recursão e é disparada quando não existem mais campos a serem processados. A regra CR-MULTIPLE formaliza o passo recursivo

⁶À primeira vista, esta afirmação parece ser um descuido, uma vez que claramente viola a propriedade da assimetria. Porém, a existência de um campo f armazenando $\sqcup$ em o_1 e o_2 não é possível. Tal fato implica que f não existe em nenhum dos operandos usados para a construção do registro de comparação em questão. Em tal cenário, porém, f não faria parte do registro de comparação.

$$\begin{array}{l}
\text{CR-BUILD} \frac{op_1 : C \quad \text{class } C \triangleleft D \{ \overline{C} \, \overline{f}; K \, \overline{M} \} \quad op_2 : C' \quad \text{class } C' \triangleleft D' \{ \overline{C}' \, \overline{f}'; K' \, \overline{M}' \}}{build(op_1, op_2) \rightarrow cpBuild(op_1, op_2, es_fields(C) \cup es_fields(C'))} \\
\\
\text{CR-EMPTY} \frac{f \rightarrow \emptyset}{cpBuild(op_1, op_2, f) \rightarrow \langle \rangle} \\
\\
\text{CR-SINGLE} \frac{f \rightarrow \{f_0\}}{cpBuild(op_1, op_2, f) \rightarrow \langle o_1 = fetch(f_0, op_1), o_2 = fetch(f_0, op_2) \rangle} \\
\\
\text{CR-MUTIPLE} \frac{f \rightarrow \{f_0 \dots f_n\}}{cpBuild(op_1, op_2, f) \rightarrow \langle cpBuild(op_1, op_2, f \setminus \{f_n\}) \mid cpBuild(op_1, op_2, \{f_n\}) \rangle} \\
\\
\text{FETCH-EXISTS} \frac{o : C \quad \text{class } C \triangleleft D \{ \overline{C} \, \overline{f}; K \, \overline{M} \} \quad f \in \overline{f}}{fetch(f, o) \rightarrow o.f} \\
\\
\text{FETCH-DOES-NOT-EXIST} \frac{o : C \quad \text{class } C \triangleleft D \{ \overline{C} \, \overline{f}; K \, \overline{M} \} \quad f \notin \overline{f}}{fetch(f, o) \rightarrow \sqcup}
\end{array}$$

Figura 6.3: Regras de Inferência – Construção de Registros de Comparação.

da função, onde os campos são processados um a um.

A regra **CR-SINGLE** formaliza como cada campo é efetivamente processado. Seu processo consiste na obtenção dos valores associados à um campo f em ambos operandos (op_1 e op_2). A obtenção do campo f em questão é realizada pela função $fetch$.

Duas regras – **FETCH-EXISTS** e **FETCH-DOES-NOT-EXIST** – formalizam a função $fetch$. A regra **FETCH-EXISTS** é disparada quando a a hierarquia de classes do objeto o define o campo f . Nesse caso, o valor associado à f em o é retornado. A regra **FETCH-DOES-NOT-EXIST**, por outro lado, é disparada quando a hierarquia de o não define o campo f . Neste caso, $fetch$ retorna $\sqcup$.

6.3.4 Fase de Avaliação

A fase de avaliação é a segunda fase do processo de avaliação de operações de comparação. Cada operação tem um conjunto próprio de regras de reescrita.

6.3.4.1 Igualdade

A Figura 6.4 exibe as regras de reescrita referentes à igualdade. A sua função principal é a função cr_eq que, dado um registro de comparação r , avalia se os operandos usados para sua construção são equivalentes. A função cr_eq atua de forma recursiva nos campos de r , conforme formalizado pelas regras **EQ-MULTIPLE-EQ**, **EQ-MULTIPLE-NEQ** e **EQ-EMPTY**.

A existência de uma discrepância em um campo f_n tal que $r.f_n.o_1 \neq r.f_n.o_2$

$$\begin{array}{ll}
\text{EQ-SINGLE} \frac{r \rightarrow \langle f_0 = \langle o_1 = x_0, o_2 = y_0 \rangle \rangle}{cr_eq(r) \rightarrow eq(x_0, y_0)} & \text{EQ-KNOWN} \frac{x \neq \perp \wedge y \neq \perp}{eq(x, y) \rightarrow x = y} \\
\\
\text{EQ-MULTIPLE-EQ} \frac{r \rightarrow \langle f_0 = \langle o_1 = x_0, o_2 = y_0 \rangle \dots f_n = \langle o_1 = x_n, o_2 = y_n \rangle \rangle \quad cr_eq(r.f_n) \rightarrow \top}{cr_eq(r) \rightarrow cr_eq(r \setminus f_n)} & \\
\\
\text{EQ-MULTIPLE-NEQ} \frac{r \rightarrow \langle f_0 = \langle o_1 = x_0, o_2 = y_0 \rangle \dots f_n = \langle o_1 = x_n, o_2 = y_n \rangle \rangle \quad cr_eq(r.f_n) \rightarrow \perp}{cr_eq(r) \rightarrow \perp} & \\
\\
\text{EQ-UNKNOWN} \frac{x = \perp \vee y = \perp}{eq(x, y) \rightarrow \perp} & \text{EQ-EMPTY} \frac{r \rightarrow \Diamond}{cr_eq(r) \rightarrow \top}
\end{array}$$

Figura 6.4: Regras de Inferência – Avaliação de Igualdade.

dispara a regra EQ-MULTIPLE-NEQ, o que resulta na obtenção de $\perp$ (e no término do processo). As regras EQ-EMPTY e EQ-MULTIPLE-NEQ modelam, respectivamente, o caso base e o passo recursivo da função cr_eq .

Caso nenhuma discrepância seja encontrada, a regra EQ-SINGLE é disparada. O resultado obtido EQ-SINGLE depende se $f_0.o_1$ e $f_0.o_2$ são iguais. Essa comparação é realizada pela função eq . Duas regras – EQ-UNKNOWN e EQ-KNOWN – modelam a semântica de eq . Caso o_1 ou o_2 sejam $\perp$, a regra EQ-UNKNOWN é disparada e $\perp$ é obtido (de acordo com a Definição 5).

Caso contrário EQ-KNOWN é disparada. Caso os tipos incluídos em o_1 e o_2 sejam *primitivos*⁷, a sua equivalência é checada através de sua implementação nativa. Caso contrário, um novo registro de comparação é construído através de $f_0.o_1$ e $f_0.o_2$ e é subsequentemente avaliado. Essa verificação, independentemente da abordagem utilizada, é representada pelo operador $=$. Qualquer que seja a abordagem utilizada, $\top$ ou $\perp$ é retornado, dependendo de se $f_0.o_1$ ou $f_0.o_2$ são equivalentes.

6.3.4.2 Ordem

A função cr_cp avalia a ordem natural entre os operandos utilizados para construir um registro de comparação r . Conforme Figura 6.5, as duas principais regras que formalizam cr_cp são as regras CP-MULTIPLE-EQ e CP-MULTIPLE-NEQ que operam sob o mesmo racional que EQ-MULTIPLE-EQ e EQ-MULTIPLE-NEQ, respectivamente.

As regras CP-MULTIPLE-EQ e CP-MULTIPLE-NEQ dividem um registro de comparação r em dois. O segundo registro contém unicamente o último campo de r

⁷O termo *primitivo* aqui é empregado de uma maneira abrangente. Neste contexto, qualquer tipo que provenha implementações para operação de comparação é primitivo (a classe *String* na linguagem de programação Java seria considerada primitiva).

$$\begin{array}{ll}
\text{CP-SINGLE} \frac{r \rightarrow \langle f_0 = \langle o_1 = x, o_2 = y \rangle \rangle}{cr_cp(r) \rightarrow cp(x, y)} & \text{CP-KNOWN} \frac{x \neq \sqcup \quad y \neq \sqcup}{cp(x, y) \rightarrow x \Leftrightarrow y} \\
\\
\text{CP-UNKNOWN-LT} \frac{x = \sqcup}{cp(x, y) \rightarrow -1} & \text{CP-UNKNOWN-GT} \frac{y = \sqcup}{cp(x, y) \rightarrow 1} \\
\\
\text{CP-MULTIPLE-EQ} \frac{r \rightarrow \langle f_0 = \langle o_1 = x_0, o_2 = y_0 \rangle \dots f_n = \langle o_1 = x_n, o_2 = y_n \rangle \rangle \quad cr_cp(r.f_n) \rightarrow 0}{cr_cp(r) \rightarrow cr_cp(r \setminus f_n)} & \\
\\
\text{CP-MULTIPLE-NEQ} \frac{r \rightarrow \langle f_0 = \langle o_1 = x_0, o_2 = y_0 \rangle \dots f_n = \langle o_1 = x_n, o_2 = y_n \rangle \rangle \quad cr_cp(r.f_n) \neq 0}{cr_cp(r) \rightarrow cp(r.f_n)} & \\
\\
\text{CP-EMPTY} \frac{r \rightarrow \Diamond}{cp(r) \rightarrow 0} &
\end{array}$$

Figura 6.5: Regras de Inferência – Avaliação de Ordem.

enquanto o primeiro registro os campos restantes. A regra **CP-MULTIPLE-NEQ** é disparada quando existem discrepâncias no segundo registro. Neste caso, a ordem obtida através do segundo registro define a ordem de r . Caso contrário, a regra **CP-MULTIPLE-EQ** é disparada e o processo continua recursivamente, até que a regra **CP-SINGLE** seja disparada.

A regra **CP-SINGLE** lida com registros de comparação contendo um único campo f . Nesse caso, a comparação propriamente dita é realizada pela função cp , que por sua vez é formalizada pelas regras **CP-UNKNOWN-LT**, **CP-UNKNOWN-GT**, e **CP-KNOWN**.

A regra **CP-SINGLE** lida com registros de comparação contendo um único campo f . A comparação propriamente dita é realizada pela função cp , que por sua vez é formalizada pelas regras **CP-UNKNOWN-LT**, **CP-UNKNOWN-GT**, e **CP-KNOWN**. As regras **CP-UNKNOWN-LT** e **CP-UNKNOWN-GT** modelam a Definição 5 e são disparadas quando um dos componentes de f armazena $\sqcup$. No caso de **CP-UNKNOWN-LT**, o valor armazenado em o_1 é $\sqcup$ (já no caso de **CP-UNKNOWN-GT**, $\sqcup$ está armazenado em o_2).

A regra **CP-KNOWN** é disparada quando ambos componentes de f armazenam valores conhecidos. Neste caso, a ordem é obtida através do operador $\Leftrightarrow$, cuja implementação pode ser nativa (quando os valores sendo comparados são primitivos) ou baseada em registros de comparação.

Por fim, a regra **CP-EMPTY** lida com registros de comparação vazios. Registros vazios sempre são considerados equivalentes.

$$\begin{array}{l}
\text{HC-SINGLE} \frac{r \rightarrow \langle f_0 = \langle o_1 = x_0, o_2 = y_0 \rangle \rangle}{cr_hc(r) \rightarrow hc(x_0)} \quad \text{HC-EMPTY} \frac{r \rightarrow \langle \rangle}{cr_hc(r) \rightarrow 0} \\
\\
\text{HC-MULTIPLE} \frac{r \rightarrow \langle f_0 = \langle o_1 = x_0, o_2 = y_0 \rangle \dots f_n = \langle o_1 = x_n, o_2 = y_n \rangle \rangle}{cr_hc(r) \rightarrow cr_hc(r \setminus f_n) \oplus hc(r.f_n)}
\end{array}$$

Figura 6.6: Regras de Inferência – Cálculo *Hash*.

6.3.4.3 Código *hash*

As regras na Figura 6.6 lidam com o cálculo de códigos *hash*. A função *cr_hc* calcula o código hash de um registro de comparação *r*. Se o registro *r* tem mais que um campo, a regra HC-MULTIPLE é disparada.

A regra HC-MULTIPLE divide o registro de comparação *r* em dois sub-registros contendo o último campo e os campos restantes. O código *hash* de ambos sub-registros é obtido recursivamente e combinado através do operador $\oplus$ (conforme observado na Seção 6.2.3, $(\mathbb{Z}, \oplus)$ deve formar um semigrupo).

A regra HC-SINGLE é disparada quando é necessário a obtenção do código hash de um registro contendo apenas um campo. Nesse caso é utilizada a função *hc*. De maneira análoga a função *eq* discutida anteriormente, a função *hc* usa a implementação nativa de cálculo de *hashes* caso *o*₁ seja um tipo primitivo, ou, caso contrário, cria um novo registro de comparação e o avalia.

É importante ressaltar o porquê da regra HC-SINGLE só fazer uso da primeira componente do registro de comparação – *o*₁. Registros de comparação são compostos de duas componentes, que armazenam, respectivamente, os valores associados a cada operando de uma operação de comparação. O cálculo de código *hash*, porém, usa apenas um único operando. Deste modo, somente *o*₂ é considerado.

Por fim, a regra HC-EMPTY é usada para calcular o código *hash* de registros vazios e também serve de caso base para o processo recursivo de cálculo de código *hash*. Considera-se que o código *hash* de um registro vazio seja zero⁸.

6.3.5 Lidando com Ciclos

O último passo para a formalização é o tratamento de ciclos. Conforme descrito na Figura 6.7, são introduzidas seis novas regras para o tratamento de ciclos. Essas regras recém introduzidas especializam três regras de inferência descritas

⁸O número zero foi escolhido de maneira arbitrária. Qualquer outro número poderia ser usado.

$$\begin{array}{l}
\text{EQ-SINGLE-CYCLE-NOT-FOUND} \frac{r \rightarrow \langle f_0 = \langle o_1 = x, o_2 = y \rangle \rangle \quad \neg(\text{has_cycle}(x, y))}{cr_eq(r) \rightarrow eq(x, y)} \\
\text{EQ-SINGLE-CYCLE-FOUND} \frac{r \rightarrow \langle f_0 = \langle o_1 = x_0, o_2 = y_0 \rangle \rangle \quad \text{has_cycle}(x_0, y_0)}{cr_eq(r) \rightarrow \top} \\
\text{CP-SINGLE-CYCLE-NOT-FOUND} \frac{r \rightarrow \langle f_0 = \langle o_1 = x, o_2 = y \rangle \rangle \quad \neg(\text{has_cycle}(x, y))}{cr_cp(r) \rightarrow cp(x, y)} \\
\text{CP-SINGLE-CYCLE-FOUND} \frac{r \rightarrow \langle f_0 = \langle o_1 = x, o_2 = y \rangle \rangle \quad \text{has_cycle}(x, y)}{cr_cp(r) \rightarrow 0} \\
\text{HC-SINGLE-CYCLE-NOT-FOUND} \frac{r \rightarrow \langle f_0 = \langle o_1 = x, o_2 = y \rangle \rangle \quad \neg(\text{has_cycle}(x, y))}{cr_hc(r) \rightarrow hc(x)} \\
\text{HC-SINGLE-CYCLE-FOUND} \frac{r \rightarrow \langle f_0 = \langle o_1 = x, o_2 = y \rangle \rangle \quad \text{has_cycle}(x, y)}{cr_hc(r) \rightarrow 0}
\end{array}$$

Figura 6.7: Regras de Inferência – Tratamento de Ciclos.

na Seção 6.3.4.

Todas as regras que tratam ciclos fazem uso da função *has_cycle* que recebe dois parâmetros e retorna $\top$ caso os parâmetros fornecidos já foram processados (i.e., caso um ciclo tenha sido encontrado), ou $\perp$, caso contrário. Da mesma maneira que a função *es_fields*, os detalhes de como esses ciclos são detectados é particular de cada implementação.

As regras **EQ-SINGLE-CYCLE-NOT-FOUND** e **EQ-SINGLE-CYCLE-FOUND** substituem a regra **EQ-SINGLE**. A regra **EQ-SINGLE-CYCLE-NOT-FOUND** é disparada se um ciclo não é encontrado, e se comporta de maneira idêntica à **EQ-SINGLE**. Já a regra **EQ-SINGLE-CYCLE-FOUND** é disparada quando um ciclo foi encontrado. Nesse caso, o valor de computação retornado é $\top$. Conforme explicado na Seção 6.2.4, o racional baseia-se na premissa que, caso existissem discrepâncias entre o_1 e o_2 , as mesma já teriam sido encontradas.

Da mesma maneira, **CP-SINGLE-CYCLE-NOT-FOUND** e **CP-SINGLE-CYCLE-FOUND**; e **HC-SINGLE-CYCLE-NOT-FOUND** e **HC-SINGLE-CYCLE-FOUND** substituem as regras **CP-SINGLE** e **HC-SINGLE**, respectivamente. Em ambos os casos, é retornado 0 caso um ciclo seja detectado.

O retorno de zero pela regra **HC-SINGLE-CYCLE-FOUND** tem impacto direto nas premissas apresentadas para tratamento de código *hashes*. Conforme descrito na Seção 6.3.4.3, é necessário que $(\mathbb{Z}, \oplus)$ forme um semi-grupo. O tratamento de ciclo apresentado, porém, necessita de regras mais rígidas. Para que o algoritmo apresentado funcione corretamente, é necessário que $(\mathbb{Z}, \oplus, 0)$ forme um monoide.

6.4 Provas

Após a descrição formal da proposta, é necessário provar a sua corretude. A corretude da proposta é provada com base nas Definições 1, 2 (Capítulo 2) e na Definição 3 (Capítulo 3), que formalizam, respectivamente, a igualdade, ordem, e funções *hash* bem-formadas.

As provas aqui dispostas são baseadas em duas premissas, que devem ser válidas para que as provas também o sejam.

Premissa 1: Os operandos não mudam durante a execução das comparação;

Premissa 2: Os valores associados aos componentes o_1 e o_2 dos campos dos registros de comparação, quando primitivos, têm procedimentos de comparação corretos.

6.4.1 Igualdade

Lema 1 (Simetria). *Dados dois objetos – $x : C$ e $y : C'$ –, $x = y \Leftrightarrow y = x$.*

Demonstração. Através da aplicação de CR-BUILD, EQ-MULTIPLE-EQ, EQ-SINGLE, e EQ-EMPTY, infere-se que $(x = y) \implies [(\forall f \in \bar{f})(x.f = y.f)]$, onde $\bar{f} = (es_fields(C) \cup es_fields(C'))$. Assumir-se que $y \neq x$ implica que a união de conjuntos não é comutativa; ou que $[(\exists f \in \bar{f})((x.f = y.f) \wedge (y.f \neq x.f))]$. O último caso implica que a comparação dos valores no campo f não é simétrica (o que vai contra à Premissa 2). $\square$

Lema 2 (Reflexividade). *Dado um objeto $o : C$, então $o = o$.*

Demonstração. A execução da regra CR-BUILD resulta em $r = \langle f_0 = \langle o_1 = fetch(f_0, o), o_2 = fetch(f_0, o) \rangle \dots f_n = \langle o_1 = fetch(f_n, o), o_1 = fetch(f_n, o) \rangle \rangle$. Assumindo que a função *eq* é reflexiva (Premissa 2) e que a função *fetch* retorna o mesmo valor dado os mesmos parâmetros (Premissa 1), conclui-se que $eq(fetch(f, o), fetch(f, o))$ retornará $\top$ para todo campo f . Aplicando-se a regra EQ-MULTIPLE-EQ repetidamente, a regra EQ-SINGLE acaba sendo disparada, retornando assim $\top$. $\square$

Lema 3 (Transitividade). *Dado três objetos – $x : C$, $y : C'$, e $z : C''$ –, então $(x = y \wedge y = z) \Leftrightarrow x = z$.*

Demonstração. Através da aplicação de CR-BUILD, EQ-MULTIPLE-EQ, EQ-SINGLE, e EQ-EMPTY, infere-se que:

- $(x = y) \implies [(\forall f \in \bar{f})(x.f = y.f)]$, onde $\bar{f} = (es_fields(C) \cup es_fields(C'))$;
- $(y = z) \implies [(\forall g \in \bar{g})(y.g = z.g)]$, onde $\bar{g} = (es_fields(C') \cup es_fields(C''))$;
- $(x = z) \implies [(\forall h \in \bar{h})(x.h = z.h)]$, onde $\bar{h} = (es_fields(C) \cup es_fields(C''))$.

Ter-se $(x = y) \wedge (y = z) \wedge (x \neq z)$ implica que existe um campo f onde $(x.f = y.f) \wedge (y.f = z.f) \wedge (x.f \neq z.f)$. Isso implica nas seguintes conclusões:

- A igualdade do campo f não é, por si só, transitiva. O que, segundo a Premissa 2, é contraditório;
- A expressão $(x \neq z)$ implica em $[(\exists f)(x.f \neq z.f)]$, tal que $(f \notin \bar{f}) \wedge (f \notin \bar{g}) \wedge (f \in \bar{h})$. De acordo com as definições de $\bar{f}$, $\bar{g}$, e $\bar{h}$ dadas anteriormente e a definição da união de conjunto disposta por Shen et al. (2002), têm se:

- $(f \notin \bar{f}) \implies [(f \notin es_fields(C)) \wedge (f \notin es_fields(C'))]$;
- $(f \notin \bar{g}) \implies [(f \notin es_fields(C')) \wedge (f \notin es_fields(C''))]$;
- $(f \in \bar{h}) \implies [(f \in es_fields(C)) \vee (f \in es_fields(C''))]$.

Todavia, essas afirmações implicam em $(\exists f)[(f \notin es_fields(C)) \wedge (f \notin es_fields(C'')) \wedge ((f \in es_fields(C)) \vee (f \in es_fields(C'')))]$, o que é contraditório.

□

6.4.2 Ordem

Lema 4 (Transitividade). *Dado três objetos – $x : C$, $y : C'$, e $z : C''$ –, então $(x > y \wedge y > z) \Leftrightarrow x > z$.*

Demonstração. Assumindo que $x > y$ e $y > z$, e analisando CP-MULTIPLE-NEQ, pode-se inferir que dois campos – f' e f'' – determinam que $x > y$ e $y > z$. Baseando-se nisso, pode-se dividir a prova em três cenários.

- Quando $f' = f''$, pode-se provar a transitividade de maneira trivial, uma vez que $=$ é uma operação por si só transitiva (Lema 3);

- Quando f' é avaliado antes de f'' , pode-se concluir que $y.f' = z.f'$ (caso contrário f' teria uma ordem de avaliação maior). Substituindo $y.f'$ em $x.f' > y.f'$, obtém-se $x.f' > z.f'$. Logo, $x > z$;
- Por fim, quando f' é avaliado depois de f'' , pode-se concluir que $x.f'' = y.f''$ (caso contrário, f' teria uma ordem de avaliação menor). A substituição de $y.f''$ em $y.f'' > z.f''$ resulta em $x.f'' > z.f''$. Logo, $x > z$.

□

Lema 5 (Assimetria). *Dado dois objetos – $x : C$ e $y : C'$ –, então $(x < y) \Leftrightarrow (y > x)$.*

Demonstração. A expressão $x < y$ ser verdadeira implica na existência de um campo f onde $x.f < y.f$. A expressão $y < x$ também ser válida implica que:

- A função cp não é simétrica, o que contradiz com a Premissa 2;
- Que a iteração dos campos obtidos por $(es_fields(C) \cup es_fields(C'))$ e $(es_fields(C') \cup es_fields(C))$ não é baseada na ordem de avaliação dos campos constituintes (caraterística fundamental dos registro de comparação, segundo a Definição 4);
- Que $(es_fields(C) \cup es_fields(C'))$ e $(es_fields(C') \cup es_fields(C))$ não contém os mesmos elementos. O que implica que a união de conjuntos não é comutativa.

Quaisquer das três implicações são contraditórias.

□

Lema 6 (Tricotomia). *Dado dois objetos – $x : C$ e $y : C'$ –, os objetos têm que ser $x = y$, $x < y$ ou $x > y$.*

Demonstração. Através da análise de CP-EMPTY, CP-UNKNOWN-LT, CP-UNKNOWN-GT, e CP-KNOWN, junto à premissa que $\Leftrightarrow$ é uma operação tricotômica (Premissa 2), pode-se concluir que a computação sempre termina e resulta em 0, -1 ou 1. Logo a comparação de objetos é tricotômica.

□

6.4.3 Código *hash*

Lema 7. *Dado dois objetos – $x : C$ e $y : C'$ –, $(x = y) \implies hc(x) = hc(y)$.*

Demonstração. Analisando as regras CR-BUILD, EQ-MULTIPLE-EQ, EQ-SINGLE e EQ-EMPTY, pode-se deduzir que $(x = y) \implies [(\forall f \in \bar{f})(eq(x.f, y.f))]$, sendo $\bar{f} = (es_fields(C) \cup es_fields(C'))$. Inspeccionando as regras HC-MULTIPLE e HC-SINGLE, concluí-se que $hc(x) \neq hc(y)$ implica que ou $(\exists f \in \bar{f})((x.f = y.f) \wedge (hc(x.f) \neq hc(y.f)))$ (que, segundo a Premissa 2, é um absurdo) ou o operador $\oplus$ não é associativo. Logo, $(\mathbb{Z}, \oplus)$ não é um semi-grupo, conforme requerido na Seção 6.2.3. Em ambos os casos, existe uma contradição. $\square$

6.5 Conclusões do Capítulo

Este capítulo teve como objetivo a descrição de uma proposta para cálculo das operações de comparação – verificação de igualdade, ordem e código *hash* – de maneira holística. Através da análise dos trabalhos correlatos, chegou-se a um grupo de características desejáveis para a solução:

- Denotação de semântica declarativa e uniforme;
- Sem restrições à mutabilidade;
- Expressivo suficiente para permitir as diversas estratégias de comparação definidas por Rupakheti (2010).

A proposta em questão foi descrita de maneira agnóstica de linguagem. Não se comprometer com uma linguagem específica permite que a mesma solução seja utilizada em outras linguagens e *frameworks*. Apesar das vantagens, a adoção dessa estratégia também trouxe alguns malefícios. De modo a não enviesar a solução descrita, algumas lacunas tiveram que ser deixadas em aberto. Essas lacunas foram: o modo com qual a semântica de declaração é codificada dentro das classes; e o modo com qual ciclos são detectados.

A descrição foi apresentada de modo informal e formal. A descrição informal têm como objetivo principal preparar o leitor para as complexidades da descrição formal. A descrição formal foi apresentada baseando-se em três formalismos: Cálculo de Registros (CARDELLI; MITCHELL, 1991), *FeatherWeight Java* (IGARASHI; PIERCE; WADLER, 2001), e Semântica Operacional (PLOTKIN, 1981).

Através da descrição formal da solução, foi apresentada provas validando sua corretude. Essas provas partem de um conjunto específico de premissas e têm como base as regras de boa-formação apresentadas nos Capítulos 2 e 3.

Todavia, é importante ressaltar que a corretude da solução é apenas um aspecto de qualidade a ser avaliado. Outros aspectos, porém, não podem ser avaliados a partir de uma descrição formal abstrata. Para a avaliação desses outros aspectos, se faz necessário um protótipo.

7 Protótipo

A proposta teórica apresentada no capítulo anterior será materializada através de um protótipo¹. Além de facilitar o entendimento da proposta, o protótipo permitirá a aferição empírica de algumas características de qualidade (Capítulo 8).

É importante ressaltar que o escopo do protótipo é o aspecto funcional da proposta (i.e., a habilidade de computar de maneira correta as operações de igualdade). As otimizações incorporadas são mínimas. De todo modo, são descritas durante as considerações finais – Capítulo 9 – algumas possibilidades de otimizações de desempenho que podem ser exploradas no futuro.

Apesar da proposta ser agnóstica de linguagem, é necessária a escolha de uma linguagem para a implementação do protótipo. Para esse intuito, escolheu-se a linguagem de programação Java. Entre os motivos da escolha de Java encontram-se: o seu rico ecossistema, que facilita a implementação do protótipo; a disponibilidade do código-fonte de tipos variados de programas que podem ser usados para a avaliação dos impactos da solução; e, a inexistência de construtos exóticos na linguagem (o que facilita uma eventual migração do protótipo para outra linguagem OO). Outro ponto digno de nota é que a maioria dos trabalhos analisados no Capítulo 5 focam em Java. É importante ressaltar, porém, que nada impede que outra linguagem seja usada para a implementação de um protótipo semelhante.

Este capítulo segue a mesma estrutura do Capítulo 6. Essa decisão tem como racional facilitar o entendimento de como os construtos do Capítulo 6 são mapeados no protótipo.

7.1 Registros de Comparação

Conforme discutido na Seção 6.2.1, registros de comparação armazenam o contexto no qual uma operação de comparação é avaliada. Os valores pertencentes

¹Disponível em <https://github.com/bsofiato/prototype-master-thesis>.

Figura 7.1: Diagrama de Classe – Registros de Comparação.

ao estado de igualdade de cada um dos operandos são armazenados nos campos dos registros de comparação. No protótipo, os registros de comparação são representados pela classe *ComparisonHandler*, conforme apresentado no diagrama da Figura 7.1.

Uma característica da linguagem Java faz com que a implementação de registros de comparação conforme descrito no Capítulo 6 seja difícil: em Java, é necessário que os atributos de uma classe sejam definidos *a priori*, não sendo possível adicionar ou remover atributos em tempo de execução. Implementar registro de comparação como definido requereria uma classe diferente para cada combinação possível de operandos.

Ao invés disso, optou-se por utilizar um campo – *fields* – que armazena uma *array* associativo, cuja chave é o campo original. Outra diferença é que, ao invés de serem armazenados valores concretos, são armazenadas estratégias de obtenção de valor – *ValueFetcher* –, específicas para cada um dos operandos. Essa abordagem permite que uma mesma instância de *ComparisonHandler* seja reutilizada em invocações futuras (i.e., quando se deseje comparar outro par de operandos que têm os mesmos tipos).

O uso da interface *Strategy* desacopla essa funcionalidade do resto do sistema, e permite que padrões de projetos como o Decorador (GAMMA et al., 1995) sejam utilizados para incorporar novos comportamento. A classe *CycleHandler*², cujo papel é encapsular a lógica de detecção de ciclos, é um exemplo disso.

Outro ponto importante refere-se ao valor *Desconhecido*. No protótipo, esse valor é representado pela classe *Unknown*. A classe *Unknown* implementa os métodos de comparação de maneira à representar a semântica de comparação formalizada na Definição 5. É importante ressaltar que a implementação segue

²Mais informações na Seção 7.4.

Figura 7.2: Diagrama de Classe – Construção de Registros de Comparação.

o padrão de projeto *Singleton* (GAMMA et al., 1995) (i.e., existe apenas uma instância de *Unknown*, que é reaproveitada).

7.2 Fase de Construção

De acordo com a Seção 6.2.2, o processo de criação de um registro de comparação é composto de dois passos. No primeiro passo são obtidos os campos pertencentes ao estado de igualdade dos operandos enquanto no segundo passo os valores associado com cada um dos campos são obtidos.

A classe *DefaultFactory* (Figura 7.2) encapsula o processo de comparação. *DefaultFactory* define um único método – *create* – que recebe os dois operandos da operação de comparação sendo computada e retorna um registro de comparação. Para isso, são utilizadas técnicas de programação reflexiva para a obtenção dos campos que compõem o estado de igualdade de um operando. Durante esse processo, os operandos são analisados em busca das anotações descritas na Seção 7.2.1.

Existe uma pequena diferença entre o protótipo e a descrição da proposta no Capítulo 6 no que tange o segundo passo do processo de construção de registros. Conforme discutido na Seção 7.1, o protótipo armazena instâncias de *ValueFetcher* nos registros de comparação ao invés de valores concretos. Deste modo, a segundo passo do processo de criação consiste na criação dessas instâncias.

Duas implementações distintas podem ser instanciadas, dependendo do campo *f* em questão existir no operando *x*. Caso o campo exista, é criada uma instância de *KnownValueFetcher*. Essa implementação usa programação reflexiva para obter o valor associado ao campo no operando fornecido. Caso o campo não exista em *o*, cria-se uma instância de *UnknownValueFetcher*. Instâncias deste tipo sempre retornam o valor `Unknown.UNKNOWN`, independentemente do operando

fornecido.

O protótipo fornece mais duas implementações de *Factory* – *CachingFactory* e *CycleAwareFactory*. Essas são construídas baseadas no padrão de projeto *Decorator* (GAMMA et al., 1995) e incorporam funcionalidades auxiliares à fábrica padrão. *CachingFactory* cacheia as estratégias criadas enquanto *CycleAwareFactory* adiciona tratamento e detecção de ciclos (Seção 7.4).

7.2.1 Denotando Semântica de Comparação

Conforme discutido na Seção 6.2.2, é necessário que seja possível que o programador defina quais propriedades devem ser levadas em conta quando objetos são comparados (i.e., quais propriedades fazem parte do seu estado de igualdade). Também é necessário que seja possível a definição da ordem de avaliação dessas propriedades.

A proposta não define como os metadados representando a semântica de comparação de um objeto devem ser codificados. Para o protótipo, foi escolhido o mecanismo de anotações para denotar a semântica de comparação. Disponíveis desde a versão 1.5, as anotações são o mecanismo padrão para a incorporação de metadados às classes na linguagem Java.

Figura 7.3: Diagrama de Classe – Anotações.

O protótipo disponibiliza duas anotações distintas – *@Comparable* e *@ComparableAttribute* – para a denotação de semântica de comparação (Figura 7.3). A anotação *@Comparable* marca as classes comparáveis. Seu atributo *useConcreteClass* indica se deve se usar uma estratégia de comparação compatível ou incompatível com tipos (*true* ou *false*, respectivamente).

Já a anotação *@ComparableAttribute* marca os atributos que devem ser levados em conta na comparação entre objetos (i.e., os atributos que fazem parte do estado de igualdade da classe em questão). Somente atributos anotados são levados em conta. O atributo *evaluationOrder* indica a ordem de avaliação do atributo (atributos com ordem de avaliação menor são avaliados primeiro). Caso uma ordem não seja fornecida, uma ordem sintética determinística é gerada.

A Figura 7.4 dispõe um trecho de código-fonte na qual a semântica de com-

paração das classes *Ponto* e *PontoColorido* é denotada através de anotações. É importante ressaltar que a semântica apresentada é idêntica a apresentada na Seção 3.1.1. Pode-se notar que tanto *Ponto* como *PontoColorido* é anotado com a anotação *@Comparable*. Não obstante, todas as propriedades que fazem parte de seu estado de igualdade são anotadas pela anotação *@ComparableAttribute*.

```

1 @Comparable
2 public class Ponto {
3     @ComparableAttribute
4     private double x;
5
6     @ComparableAttribute
7     private double y;
8 }
9
10 @Comparable
11 public class PontoColorido extends Ponto {
12     @ComparableAttribute
13     private Cor cor;
14 }

```

Figura 7.4: Semântica de Comparação denotada através de anotações.

7.3 Fase de Avaliação

Conforme descrito nas Seções 6.2.3 e 6.3.4, algoritmos ligeiramente diferentes são utilizados para computar o resultado de cada uma das operações de comparação. Cada um desses algoritmos é representado por um método definido na classe *ComparisonHandler*. Os métodos *equals*, *compareTo*, e *hashCode* implementam, respectivamente, os procedimentos para a verificação da igualdade, ordem e cálculo do código *hash*.

Figura 7.5: Estratégia de Comparação.

Pode-se notar no diagrama da Figura 7.5 que esses métodos recebem argumentos. No caso dos métodos *equals* e *compareTo*, esses argumentos são os operandos a serem comparados. Já o método *hashCode* tem apenas um único argumento, que por sua vez representa o objeto cujo código *hash* se deseja obter.

A presença de argumentos nos métodos indicados representa uma pequena diferença do modelo teórico (Seções 6.2 e 6.3). No modelo teórico, os algoritmos

processam os registros de comparação com os valores brutos armazenados. Já no protótipo, os métodos recebem os operandos a serem comparados.

Essa diferença é necessária em razão da otimização descrita na Seção 7.1, onde, de maneira à possibilitar seu reuso, fora decidido que instâncias de *ComparisonHandler* armazenam objetos especializados – *ValueFetcher* – cujo papel é obter valores correspondentes a um campos. Como esses objetos necessitam do operando para obtenção do valor, esses tiveram que ser incluídos como parâmetros na avaliação das operações de comparação.

Um outro detalhe importante tem a ver com como os códigos *hash* são calculados. A Seção 6.3.4.3 determina regras de corretude para o mecanismo de agregação do código *hash*: no caso, a $(\mathbb{Z}, \oplus)$, onde $\oplus$ representa a função agregadora, deve formar um semi-grupo. O tratamento de ciclos exposto na Seção 6.3.5 restringe ainda mais as regras de boa-formação de $\oplus$, requerendo que $(\mathbb{Z}, \oplus, 0)$ seja um monoide.

O protótipo utiliza a adição de inteiros (i.e., $(\mathbb{Z}, +, 0)$) como função agregadora de códigos *hash*. A razão do uso da adição foi a sua simplicidade para implementar e entender. Bloch (2018) delineia um algoritmo potencialmente melhor³ para cálculo de *hashes*: a cada propriedade, o código *hash* acumulado é multiplicado por 31 a cada propriedade incorporada. Este algoritmo, apesar de correto, não se encaixa nas regras de reescrita delineada na Figura 6.6. Por esse motivo, foi vetada sua adoção no âmbito do protótipo.

7.4 Lidando com Ciclos

A descrição da solução no Capítulo 6 descreve como ciclos devem ser tratados de modo que, mesmo na sua presença, resultados corretos sejam obtidos. A forma com o qual esses ciclos são detectados, porém, é deixada em aberto.

O protótipo detecta ciclos utilizando uma técnica parecida com como linguagens de programação manipulam sua pilha de chamadas. Os valores associados a um campo são armazenados em uma pilha imediatamente antes da sua avaliação. Esses mesmos valores são retirados imediatamente depois. A inclusão de um par de valores que estejam contidos na pilha indica a presença de um ciclo.

O protótipo incorpora uma pequena otimização, no processamento do ciclo em si. Ao invés de retornar um valor inócuo quando encontra um ciclo, conforme

³Nesse contexto, entende-se melhor como com menos chance de gerar colisões.

descrito no Capítulo 6, o protótipo pula o processamento como um todo (de forma análoga à avaliação de curto-circuito, presente em muitas linguagens de programação).

É importante ressaltar que, dado o fato da linguagem Java ser uma linguagem *multithreaded*⁴, podem ocorrer falso positivos caso um mesmo grupo operandos estivessem sendo comparados em *threads* distintas. Para garantir que cada *thread* tenha uma pilha exclusiva, foi utilizada a classe *ThreadLocal*⁵, pertencente à biblioteca padrão da linguagem Java.

7.5 Ligando os Objetos de Domínio ao Protótipo

Olhando para o exemplo da Figura 7.4, é possível notar que tanto os métodos associados às operações de comparação – *equals*, *compareTo* e *hashCode* – quanto a declaração da realização da interface *Comparable* – interface necessária para que a linguagem Java entenda que o objeto em questão é comparável – não estão presentes. É necessário, porém, que algo faça a ligação entre as classes do protótipo e as classes escritas pelo programador.

Uma abordagem seria obrigar o programador a escrever esse código. Apesar de simples, essa abordagem é problemática: um programador novato pode implementar o código de maneira errônea (ou até esquecer completamente de fazê-lo).

Ao invés disso, o protótipo usa uma outra abordagem, que não necessita que o programador escreva código.

O protótipo usa técnicas da programação baseada a aspectos (AOP). AOP (KICZALES et al., 1997) permite que aspectos distintos de um sistema que permeiam varias classes (e.g., *logging*, tratamento de erros, etc.) sejam encapsulados em módulos chamados aspectos. Esses aspectos são então "costurados"⁶ em partes específicas do código, permitindo que afetem partes distintas do código-fonte. A primeira vista, a criação de um aspecto que trate da comparação parece ser uma boa opção: ser comparável é uma característica que permeia várias classes. A linguagem Java, porém, não suporta AOP de maneira nativa.

Para que fosse possível o uso de aspectos em Java, foi necessária a utilização

⁴Linguagem de programações com essa característica permitem a execução simultânea de múltiplas linhas de execução.

⁵<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/ThreadLocal.html>.

⁶No inglês, *woven*.

do AspectJ⁷. O AspectJ é uma extensão que permite o uso de AOP na linguagem Java. Um compilador especial é disponibilizado com o AspectJ. Esse compilador, além de transformar o código-fonte Java em *bytecode* (como qualquer compilador Java), também “costura” aspectos nas classes sendo compiladas.

```
1 public final aspect ComparableAspect {  
2  
3     declare parents : (@Comparable *) implements AdvisedComparable;  
4  
5     public int AdvisedComparable.compareTo(final Object operand) {  
6         return factory.create(this, operand).compare(this, operand);  
7     }  
8  
9     public int AdvisedComparable.hashCode() {  
10        return factory.create(this, this).hashCode(this);  
11    }  
12  
13    public boolean AdvisedComparable.equals(final Object operand) {  
14        return factory.create(this, operand).equals(this, operand);  
15    }  
16 }
```

Figura 7.6: Trecho do código-fonte do aspecto *Comparable*.

A Figura 7.6 mostra um trecho⁸ de código do aspecto *ComparableAspect*. A construção do aspecto baseia-se numa funcionalidade do AspectJ chamada declaração inter-tipos. Declarações inter-tipos permitem que novas classes (e interfaces) sejam adicionadas a classes que realizem determinado padrão. Conforme descrito na linha 3, todas as classes anotadas com a anotação *@Comparable* devem realizar a interface *AdvisedComparable*.

As linhas 5, 9, e 13 mostram, por sua vez, as implementações de cada um dos métodos das classes que realizam a interface *AdvisedComparable* (i.e., das classes anotadas por *ComparableAspect*). Essas implementações operam de maneira similar: primeiramente, uma fábrica (instância de *Factory*) cria, baseada nos operandos fornecidos, uma estratégia⁹ de comparação (instância de *Strategy*); então, a operação corresponde à operação sendo computada é invocada, o resultado obtido sendo o resultado final da operação.

O diagrama de sequência da Figura 7.7 mostra o fluxo durante a computação de uma operação de comparação. Já o diagrama da Figura 7.8 apresenta uma visão geral do protótipo e de como as estruturas delineadas nas seções anteriores deste capítulo se relacionam.

⁷Disponível em <https://www.eclipse.org/aspectj/>.

⁸Para efeito de brevidade, algumas construções foram omitidas

⁹As instâncias criadas são decoradas com instâncias de *CycleHandler*, permitindo assim a detecção e tratamento de ciclos conforme descrito na Seção 7.4.

Figura 7.7: Fluxo durante a computação de uma operação de comparação.

7.6 Conclusões do Capítulo

Neste capítulo foi descrito em detalhes um protótipo na linguagem Java implementando à proposta descrita no Capítulo 6.

A estrutura desse protótipo foi pensada à facilitar o mapeamento dos construtos e procedimentos previamente descritos em conjuntos de classes com responsabilidades claras. Otimizações foram propositalmente deixadas de lado, no intuito de facilitar o entendimento.

Algumas lacunas deixadas em aberto propositalmente foram preenchidas pelo protótipo. Mais precisamente, foram detalhados o mecanismo de codificação da semântica de comparação e o mecanismo de detecção de ciclos. Esses mecanismos são particulares ao Java, mas podem, em teoria serem utilizado em outras linguagens similares.

Com o protótipo em mãos, pode-se partir para sua avaliação de acordo com características de qualidade que complementam a corretude previamente demonstrada no Capítulo 6.

Figura 7.8: Diagrama de Classes do Protótipo.

8 Experimento

A corretude da proposta não é suficiente para uma análise completa da adoção da solução: é necessário também analisar os impactos da proposta na prática. Para isso, são necessárias observações experimentais.

Para mensurar os impactos da solução, foi conduzido um experimento, no qual uma série de programas é modificada para uso do protótipo descrito no Capítulo 7. Esses programas então são avaliados através de requisitos não-funcionais, tendo como base os impactos positivos e negativos esperados pela adoção na solução.

Antes de se definir um método de avaliação para a proposta, é importante entender o mecanismo de avaliação adotado pelos trabalhos correlatos (Capítulo 5). Esse entendimento é feito sob dois pontos de vista: *corpora* e critérios para avaliação.

8.1 Avaliação em trabalhos correlatos

Os trabalhos correlatos analisados usam diferentes *corpora* em suas respectivas avaliações. Conforme discutido no Capítulo 5, Vaziri et al. (2007) usam a suíte de testes DaCapo, porém, numa versão antiga (*dacapo-2006-10*).

Já Rayside et al. (2009) e Grech, Rathke e Fischer (2010) têm *corpora* próprios. O primeiro usa três *benchmarks* distintos: um sub-conjunto do JDK que lida com o arcabouço de coleções da linguagem Java; a *Apache Commons Collections* (v3.2); e o *JFreeChart* (v1.0.0), um programa de código-fonte aberto para plotagem de gráficos. Já o segundo utiliza somente um programa – *JFreeChart* – na sua avaliação. É interessante ressaltar que o *JFreeChart* fazia parte do *corpus* da suíte DaCapo, porém foi removido na versão *9.12-bach*.

Do ponto de vista do critério para avaliação, os trabalhos relacionados usam abordagens *ad-hoc*. Vaziri et al. (2007) verificam a quantidade de classes alteradas. A corretude dos programas alterados é validada através de mecanismos do próprio DaCapo. Também são sumarizados os defeitos encontrados, porém não

é feito nenhum tipo de análise sobre o impacto da solução sobre esses defeitos.

Rayside et al. (2009) avaliam a sua solução sobre a ótica de quatro aspectos distintos: *expressividade*, baseada no número de métodos removidos; *corretude*, baseada no número de violações encontradas através da geração automática¹ de testes de unidade; *facilidade de uso*, baseada na diferença entre métodos removidos e anotações incluídas; e *desempenho*, baseado no tempo necessário para a execução dos testes gerados para a aferição da corretude.

A solução de Grech, Rathke e Fischer (2010) é avaliada sob dois aspectos: *corretude* e *desempenho*. Sob o aspecto de corretude, foi adotado ferramental de análise estática. Já o desempenho é verificado através da execução da suíte de testes de unidade.

8.2 Método

O método adotado nesse experimento segue um raciocínio semelhante ao adotado na avaliação dos trabalhos correlatos, onde os impactos são mensurados através da comparação entre duas versões de código-fonte de uma série de programas de código aberto. A primeira versão – *baseline* – corresponde ao código-fonte do programa sem modificação. A segunda versão – versão migrada – corresponde ao código-fonte do programa após os passos de migração descritos na Seção 8.2.2.

8.2.1 Corpus

Assim como Vaziri et al. (2007), o experimento aqui descrito também utiliza o *corpus* da suíte de testes de desempenho DaCapo (BLACKBURN et al., 2006). Foi utilizada a versão *9.12-bach* da suíte, mais recente que a versão utilizada por Vaziri et al. (*dacapo-2006-10*). Essas versões têm *corpora* diferentes – apenas o Apache Lucene está incluso em ambas as versões. A Tabela 8.1 dispõe os programas alterados durante os experimentos. Entre as razões para a adoção da suíte de testes DaCapo, encontram-se:

- Seu *corpus* foi escolhido de maneira a representar os diferentes perfis de carga de trabalho que são encontrados no mundo real;
- O código-fonte dos programas pertencentes ao seu *corpus* estão disponíveis;

¹Randoop (PACHECO et al., 2007) foi utilizado para a criação automática da suíte de testes.

<i>Benchmark</i>	Programa	Descrição
<i>avrora</i>	Avrora	Simulador do micro-controladore AVR.
<i>batik</i>	Apache Batik	Converte arquivos XML em SVG.
<i>eclipse</i>	Eclipse	IDE para desenvolvimento em Java.
<i>fop</i>	Apache FOP	Converte arquivos XSL-FO para PDF.
<i>h2</i>	H2	Banco de dados relacional.
<i>jython</i>	Jython	Implementação de python para a JVM.
<i>lucene</i>	Lucene	Motor de buscas textuais indexando os trabalhos de Shakespeare e a Bíblia do Rei James.
<i>lusearch</i>	Lucene	Motor de buscas textuais buscando por palavras-chaves nos trabalhos de Shakespeare e na Bíblia do Rei James.
<i>pmd</i>	PMD	Analizador estático de código.
<i>sunflow</i>	Sunflow	Renderizador de imagens.
<i>tomcat</i>	Apache Tomcat	Servidor WEB provendo aplicações Java.
<i>tradebeans</i>	Apache Geronimo	<i>Daytrader</i> com acesso via EJB.
<i>tradesoap</i>	Apache Geronimo	<i>Daytrader</i> com acesso via SOAP.
<i>xalan</i>	Apache Xalan	Processador XSLT.

Tabela 8.1: *Corpus* da suíte de testes DaCapo.

- A suíte de teste embute um validador, que verifica se as bases de código alteradas ainda produzem os mesmos resultados.

8.2.2 Passos para migração

A migração dos programas pertencentes ao *corpus* do DaCapo foi realizada através do seguinte método:

1. Altera-se o processo de construção (*build*) do programa em questão, de modo a que o aspecto de comparação seja introduzido;
2. Busca-se uma classe que declare métodos associados às operações de comparação (*equals*, *compareTo*, e *hashCode*);
3. Substituí-se os métodos por anotações correspondentes;
4. Executa-se a suíte de testes, de modo a garantir que os resultados obtidos após a execução do passo 3 sejam idênticos à versão de *baseline*, através do mecanismo de validação incluído no DaCapo;
5. Volta-se ao passo 2, até que não existam mais classes declarando operações de comparação.

É importante ressaltar que não foram realizadas quaisquer modificações além das delimitadas acima. Não foram feitas refatorações ou correção de defeitos que por ventura foram encontrados durante a migração.

8.3 Metodologia de Avaliação

Após a definição do método para migração, é necessária a definição da maneira com a qual os impactos da proposta será avaliada. Ao contrário dos trabalhos correlatos, que utilizam métodos *ad-hoc* de avaliação, o experimento descrito nesta dissertação adota os aspectos definidos na norma *ISO/IEC 25010* (2010) para a avaliação.

A norma *ISO/IEC 25010* define um modelo de avaliação de qualidade. Segundo a norma, entende-se qualidade como “o grau com qual o sistema satisfaz as necessidades implícitas e especificadas dos seus vários *stakeholders*, desta maneira provendo valor” (ISO/IEC, 2010, p. 2, tradução do autor).

Figura 8.1: Taxonomia de características de qualidade da norma *ISO/IEC 25010* (2010)

O modelo de qualidade definido pela norma *ISO/IEC 25010* é composto por características de qualidade, as quais são divididas em sub-características relacionadas, conforme representado pela Figura 8.1. As oito características definidas são:

Adequação Funcional: representa o “grau com o qual um produto ou sistema provê funções que atingem as necessidades implícitas e especificadas, quando usado sobre condições estabelecidas” (ISO/IEC, 2010, p. 10, tradução do autor);

Eficiência de Desempenho: representa o “desempenho relativo à quantidade de recursos usados sob condições estabelecidas” (ISO/IEC, 2010, p. 11, tradução do autor);

Compatibilidade: representa o “grau com o que um produto, sistema ou componente pode trocar informação com outros produtos, sistemas ou componentes, e/ou desempenhar suas funções enquanto partilhando o mesmo ambiente de *hardware* ou *software*” (ISO/IEC, 2010, p. 11, tradução do autor);

Usabilidade: representa o “grau com o que um produto ou sistema pode ser usado por usuários especificados para atingir metas especificadas em um contexto de uso especificado com eficácia, eficiência e satisfação” (ISO/IEC, 2010, p. 12, tradução do autor);

Confiabilidade: representa o “grau com o qual um sistema, produto ou componente executa funções especificadas sob condições especificadas por um período de tempo especificado” (ISO/IEC, 2010, p. 13, tradução do autor);

Segurança: representa o “grau com o qual um produto ou sistema protege informações e dados de modo que pessoas ou outros produtos e sistemas tenham nível de acesso a dados apropriados ao seus tipos e níveis de autorização” (ISO/IEC, 2010, p. 13, tradução do autor);

Manutenibilidade: representa o “grau de eficácia e eficiência com o qual um produto pode ser modificado de modo a melhorá-lo, corrigi-lo ou adaptá-lo à mudanças de ambiente ou requisitos” (ISO/IEC, 2010, p. 14, tradução do autor);

Portabilidade: representa o “grau de eficácia e eficiência com o qual um sistema, produto ou componente pode ser transferido de um *hardware*, *software* ou outro ambiente operacional ou de uso para outro” (ISO/IEC, 2010, p. 15, tradução do autor).

No tocante às características de qualidade definidas pela norma *ISO/IEC 25010*, não se espera impactos significativos na compatibilidade, portabilidade, usabilidade, segurança, e confiabilidade. A compatibilidade e a portabilidade são comumente afetadas por mudanças no nível de plataforma e de protocolos – nenhum dos dois aspectos é alterado com a adoção do protótipo. Já para que a usabilidade sofra impactos relevantes, seria necessário que houvessem alterações em como os usuários interagem com o sistema. Por fim, impactos indiretos na segurança e na confiabilidade são possíveis, porém, dependem de como os métodos de comparação são usados dentro dos mecanismos existentes de controle de acesso e tratamento de falhas. Como esses impactos não são diretos, optou-se por não mensurá-los.

Por outro lado, a eficiência de desempenho é uma característica cuja avaliação é importante. Isso se dá em razão do maior grau de dinamismo necessário para a implementação da solução. O protótipo usa técnicas de programação reflexiva, que são notoriamente mais lentas que as técnicas tradicionais em tempo de compilação (SHAMS; EDWARDS, 2013). Não obstante, o protótipo não incorpora otimizações extensas, desse modo, espera-se que os programas migrados tenham uma eficiência **menor** que a o *baseline*. Não obstante, com o advento dos dispositivos móveis, esse aspecto ganhou importância renovada, uma vez que melhorias em desempenho se traduzem em menor consumo de energia (BLACKBURN; GRID, 2008).

Outra característica que espera-se impactos é a manutenibilidade. Isso se dá em razão da remoção de código imperativo potencialmente complexo que é intrínseca à solução apresentada. O custo de manutenção representa grande parte do custo total de posse (TCO, do inglês *Total Cost of Ownership*) de um sistema. Grubb e Takang (2003) argumentam que esses custos variam entre 49% e 75% do montante gasto para o desenvolvimento inicial do sistema. Logo, um programa mais manutenível é mais barato de se manter. Espera-se que, após a migração, os programas sejam menos complexos, uma vez que o uso do paradigma declarativo (i.e., o uso das anotações definidas no protótipo) resulta em programas menos complexos (e assim mais manuteníveis), conforme discutido por Lloyd (1994).

A última característica a ser avaliada é a adequação funcional. Segundo Krasner (2020), a baixa qualidade em programas são responsáveis pelo prejuízo² de US\$ 2,08 trilhões nos anos de 2020, somente nos Estados Unidos. Não obstante, Krasner estima que o valor gasto nos Estados Unidos na busca e correção de defeitos no mesmo período beira os US\$ 607 bilhões. Esta característica é diretamente impactada pelos problemas discutidos no Capítulo 4 – problemas esses que a solução descrita neste trabalho almeja resolver –, logo, espera-se que seja a característica com o impacto mais positivo.

É importante ressaltar que as três características em questão serão avaliadas apenas sob o aspecto das suas sub-características de qualidade consideradas relevantes. Essa avaliação se dá quantitativamente através de métricas, onde um *delta* (Δ) representando os impactos entre as versões migradas e de *baseline* é calculado.

²Esse montante inclui custos com falhas operacionais (inclusive segurança), projetos mal-sucedidos, e débitos técnicos.

$$\Delta = \frac{M}{B} \quad (8.1)$$

A Fórmula 8.1 exibe a forma de cálculo de Δ , onde M representa uma métrica arbitrária na versão migrada e B a mesma métrica na versão de *baseline*. Isso permite normalizar os resultados obtidos, de modo que o tamanho dos programas em questão não impactem o grau de melhora (ou piora) dos mesmos (e.g. um programa cujo tempo de execução dobrou terá um delta de 2, não importando o tempo absoluto decorrido).

8.3.1 Eficiência de Desempenho

Conforme relatado, a eficiência de desempenho tem relação com quão eficientemente um produto utiliza os recursos computacionais para realizar suas tarefas. Duas sub-características da eficiência de desempenho são avaliadas: o comportamento temporal e a utilização de recursos. A terceira sub-característica – capacidade – não se aplica, uma vez que aborda questões de escalabilidade, que não são alteradas pela solução.

8.3.1.1 Comportamento Temporal

O comportamento temporal é “o grau com o qual o tempo de resposta, de processamento, e vazão de um produto ou sistema, enquanto desempenhando suas funções, cumpre os requisitos” (ISO/IEC, 2010, p. 11, tradução do autor).

No experimento essa sub característica é medida diretamente, através do tempo de execução de cada uma dos testes da suítes de testes DaCapo. Esses testes são executados usando o maior conjunto de dados disponível³. A fim de minimizar ruídos, cada teste é executado 10 vezes⁴, somente sendo considerados resultados cuja variância seja menor que 3%⁵.

8.3.1.2 Utilização de Recursos

Segundo a norma *ISO/IEC 25010*, utilização de recursos é “o grau com o qual a quantidade e tipos de recursos utilizados por um produto ou sistema, enquanto desempenhando suas funções, cumpre os requisitos” (ISO/IEC, 2010, p. 11, tradução do autor).

³Habilitado através do parâmetro de execução `-s large`.

⁴Configurado através do parâmetro de execução `--n 10`.

⁵Configurado através do parâmetro de execução `--variance 3.0`.

Neste contexto, recursos computacionais podem ser de vários tipos (banda de rede, disco e memória, entre outros). O experimento, porém, avalia a utilização de recursos com base em um único recurso computacional específico, utilizado por todos os programas analisados: a memória *heap* da JVM. Seu uso é avaliado indiretamente, através da métrica aqui chamada de Pressão de Memória.

A pressão de memória⁶ representa a quantidade de memória processada pelo coletor de lixo. Quanto maior a pressão, maior a utilização do subsistema de gerenciamento de memória da JVM.

$$MP = \frac{R_c}{R_b} \quad (8.2)$$

A Fórmula 8.2 indica como a pressão de memória é calculada⁷. R_c representa a quantidade de memória coletada no código migrado enquanto R_b a quantidade coletada no *baseline*. A quantidade de *bytes* coletada é obtida através da interpretação dos arquivos de *log* de atividade do coletor de lixo, gerado pela JVM⁸ durante os testes.

8.3.2 Manutenibilidade

Outra característica de qualidade avaliada no experimento é a manutenibilidade. Conforme discutido, manutenibilidade tem relação com o quão simples é modificar ou adaptar um programa. Programas com maior grau de manutenibilidade tendem a ter uma vida útil maior e apresentam um custo de desenvolvimento menor.

Em razão do perfil das modificações feitas, apenas o aspecto de modificabilidade é aferido. As mudanças descritas na Seção 8.2.2 não tem profundidade suficiente para realizar mudanças na modularidade e reusabilidade. As outras sub-características – analisabilidade e testabilidade – podem, na teoria, apresentar impactos. Esses aspectos, porém, não fazem parte do objetivo desta pesquisa. Todavia, os impactos da solução sob a ótica dessas sub-características representa um trabalho futuro viável.

⁶A pressão de memória utilizada no experimento é diferente da métrica homônima encontrada no sistemas operacional *Linux*. Neste caso, pressão de memória representa o trabalho que o sistema operacional tem ao gerenciar e mover blocos de memória para satisfazer as requisições de múltiplos usuários (LINUXMM, 2017).

⁷Curiosamente, a pressão de memória como apresentada age de maneira análoga aos Δ s apresentados para as outras métricas: os valores são relativos, e, valor menores ou maiores que um representam, respectivamente, uma melhora ou piora na utilização de recursos.

⁸Os *flags* habilitados durante a execução dos testes: `-verbosegc -Xloggc:gc.log -XX:+PrintGCDateStamps -XX:+PrintGCDetails`.

8.3.2.1 Modificabilidade

Segundo a norma *ISO/IEC 25010*, a modificabilidade é definida como o “grau com o qual um produto ou sistema pode ser eficazmente e eficientemente modificado sem a introdução de defeitos ou degradação da qualidade atual do produto” (ISO/IEC, 2010, p. 15, tradução do autor).

Banker et al. (1993) e Schroeder (1999) estabelecem uma relação entre complexidade e dificuldades na modificação de programas. Seguindo essa ideia, impactos na modificabilidade são aferidos indiretamente, através de impactos na complexidade dos programas em questão. São utilizadas duas métricas de complexidade, baseadas em dois aspectos distintos relacionados à complexidade: dificuldade de entendimento e tamanho. Essas métricas serão coletadas pela ferramenta Java-NCSS (2021).

Neste experimento, a complexidade é medida através de uma métrica *ad-hoc*, que será aqui chamada de Complexidade Ciclométrica Global (CCG). A CCG é calculada através da modificação da métrica *weighted methods per class*⁹ (WGC) Chidamber e Kemerer (1994) para uso da complexidade ciclométrica (CC) de McCabe (1976) como medida de complexidade.

A WMC é uma métrica que tem como objetivo medir a complexidade no âmbito de uma classe. Informalmente, a complexidade de uma classe é a soma-tória da complexidade de seus métodos. A Definição 6, conforme disposta em Chidamber e Kemerer (1994), formaliza a métrica WMC.

Definição 6. *Considerando uma classe C composta de métodos $m_1, \dots, m_n$, sendo $c_1, \dots, c_n$ as suas respectivas complexidades. Então:*

$$WMC = \sum_{i=1}^n c_i$$

Notavelmente, o conceito de complexidade usado na Definição 6 é deliberadamente vago. Segundo Chidamber e Kemerer (1994), o racional por de trás dessa decisão é permitir um uso mais abrangente da métrica. No contexto deste experimento, determina-se que complexidade de um método até definida através da Fórmula 8.3, considerando-se que c_m seja a complexidade de um método arbitrário m e a função *cc_mccabe* uma função que retorna a complexidade ciclométrica de um método, conforme definido por McCabe (1976).

⁹Métodos ponderados por classe, em tradução livre.

$$c_m = \begin{cases} cc_mccabe(m) & \text{se } cc_mccabe(m) \geq 1 \\ 0 & \text{se } cc_mccabe(m) < 1 \end{cases} \quad (8.3)$$

É importante pontuar o porquê do uso da complexidade conforme definida pela Função 8.3 em detrimento à definição clássica de CC – cc_mccabe –, conforme McCabe (1976). Isto se dá em razão de características idiomáticas da linguagem Java: nesta linguagem, é comum o uso de métodos chamados *accessors* e *mutators* (também chamados de *getters* e *setters*). A função dessas classes de métodos é permitir o acesso aos atributos privados de um objeto. Esses métodos não têm função na lógica de negócio e sua complexidade é baixa ($cc_mccabe = 1$), porém sua presença mascarariam o impacto da solução.

O último passo para o cálculo da CCG é agregar o WMC de todas as classes que compõem o programa em questão. Essa composição se dá através do cálculo da mediana estatística dos WMC de cada uma das classes que o compõe, conforme disposto na Definição 7.

Definição 7. *Considerando uma programa P composto de classes $c_1, \dots, c_n$ e wmc sendo uma função que retorna o valor de WMC de uma classe (seguindo a Definição 6 e a Fórmula 8.3). Então:*

$$GCC = mean(wmc(c_1), wmc(c_2), \dots, wmc(c_n))$$

A segunda métrica utilizada é o número de linhas de código de um programa e tem como foco o aspecto de tamanho de um programa. Um menor número de linhas indica um programa mais simples, enquanto um número maior indica um programa mais complexo. Para essa contagem, não são nem consideradas linhas referentes a comentários nem linhas em branco.

É importante ressaltar, que ambas bases de comparação – *baseline* e programa migrado – são escritas na mesma linguagem e seguem os mesmos idiomas de programação: a única diferença sendo o modo com qual a semântica de comparação entre objetos é definida.

8.3.3 Adequação Funcional

Conforme discutido anteriormente, a adequação funcional representa o grau com o qual um produto satisfaz as necessidades estabelecidas. Das três sub-características da adequação funcional – completude, adequação, e corretude funcional –, apenas a última é analisada. Isto se dá em razão da natureza das alterações do

experimento, em que não são alteradas funcionalidades.

8.3.3.1 Corretude Funcional

Segundo a norma *ISO/IEC 25010*, a corretude funcional é “o grau com o qual um produto ou sistema fornece resultados corretos com o grau de precisão necessário” (ISO/IEC, 2010, p. 11, tradução do autor).

A aferição da corretude funcional se através dá contagem do número de problemas encontrados na implementação dos métodos associados à comparação de objetos: *equals*, *hashCode*, e *compareTo*. Esses problemas são encontrados através do uso de uma ferramenta de análise estática – Findbugs (2021) – em ambas bases de código. Um número menor de problemas encontrados indica um código mais correto.

É importante ressaltar dois aspectos associados ao uso do FindBugs: como qualquer ferramenta de análise estática, o Findbugs faz uso de heurísticas, o que, por sua vez, é passível de falsos positivos; são levadas em conta apenas as violações diretamente ligadas a problemas na implementação das operações de comparação¹⁰.

8.4 Resultados Obtidos

Houveram alguns contra-tempos durante o processo de migração dos programas pertencentes ao *corpus* do experimento. O tamanho do código-fonte do Eclipse tornou sua migração proibitiva; e, os testes baseados no Apache Geronimo – *tradebeans* and *tradesoap* – eram incompatíveis com as versões mais recentes da linguagem Java.

```
201 public boolean equals(Object anObject)
202 {
203     return m_str.equals(anObject);
204 }
```

Figura 8.2: Techo do arquivo XMLStringDefault.java do *xalan*.

Não obstante, algumas das decisões de implementação do *xalan* prejudicaram a adoção da proposta. O trecho de código representado pela Figura 8.2 exemplifica uma dessas decisões: a implementação de *equals* delega a comparação

¹⁰Foram levados em conta somente os defeitos Co, Eq, HE, FE, NP, e RV (NP e RV somente nos métodos *equals*, *compareTo*, e *hashCode*). A descrição detalhada dos defeitos está disponível em <http://findbugs.sourceforge.net/bugDescriptions.html>.

Programa	$RTime_B$	$RTime_M$	$RTime_\Delta$	MP	FV_B	FV_M	FV_Δ	CCG_B	CCG_M	CCG_Δ	$KLOC_B$	$KLOC_M$	$KLOC_\Delta$
<i>avro</i>	10976.0	24376.0	2.2208	1.0284	139	2	0.0144	13434	13227	0.9846	28505	28249	0.9910
<i>batik</i>	2149.0	2651.0	1.2336	1.0896	43	5	0.1163	32862	32738	0.9962	78565	78404	0.9980
<i>fop</i>	336.0	601.0	1.7887	1.4996	24	2	0.0833	16450	16055	0.9760	41272	40813	0.9889
<i>h2</i>	25012.0	28330.0	1.1327	1.0272	2	0	0.0000	26840	26727	0.9958	67817	67689	0.9981
<i>jython</i>	9791.0	11158.0	1.1396	0.9999	5	2	0.4000	72371	72316	0.9992	132099	132024	0.9994
<i>luindex</i>	944.0	1440.0	1.5254	1.0044	10	3	0.3000	11687	11388	0.9744	33954	33608	0.9898
<i>lusearch</i>	1378.0	1723.0	1.2504	1.0060	10	3	0.3000	11687	11388	0.9744	33954	33608	0.9898
<i>pmd</i>	2130.0	2124.0	0.9972	1.6018	11	7	0.6364	14251	14215	0.9975	27519	27477	0.9985
<i>sunflow</i>	4491.0	4204.0	0.9361	0.9989	4	0	0.0000	3863	3858	0.9987	12159	12153	0.9995
<i>tomcat</i>	5039.0	7436.0	1.4757	1.0347	9	2	0.2222	36874	36668	0.9944	83088	82826	0.9968
<i>xalan</i>	5009.0	7542.0	1.5057	1.0640	37	24	0.6486	26361	26263	0.9963	57314	57259	0.9990
média	–	–	1.3824	1.1231	–	–	0.2421	–	–	0.9913	–	–	0.9959

$RTime_B$, $RTime_M$, e $RTime_\Delta$ são os tempos de execução do *baseline* e da versão migrada de cada programa (e sua variação). MP é a pressão de memória. FV_B , FV_M , e FV_Δ representam o número de violações encontradas. CCG_B , CCG_M , e CCG_Δ mostram a complexidade ciclomática geral, enquanto $KLOC_B$, $KLOC_M$, e $KLOC_\Delta$ o número de linhas de código (excluindo comentários e linhas em branco). Resultados em **negrito** indicam os resultados mais relevantes para a análise.

Tabela 8.2: Resultados do Experimento.

para o atributo *m_str*, que por sua vez é uma instância de *String*. Para que a simetria fosse mantida, seria necessário alterar a classe *String*. A classe *String*, porém, é parte integrante da linguagem Java, o que faz com que sua alteração seja impossível. Decisões como esta permeiam o código-fonte do *xalan*.

Nesses casos específicos, a correção requereria mudanças que estão fora do escopo delimitado na Seção 8.2.2. Ainda assim foi possível realizar a migração do *xalan* para uso da proposta, porém com um escopo reduzido (i.e., apenas um sub-conjunto de classes foram migradas).

Tirando esses casos descritos, foi possível realizar a migração de todos os outros programas. A Tabela 8.2 exibe os resultados obtidos durante o experimento. A suíte de testes foi executada em um Intel I7-4790 com 32 GB RAM.

8.4.1 Eficiência de Desempenho

Pela falta de otimizações no protótipo, o desempenho dos programas alterados foi sensivelmente pior. Em média, a suíte de testes levou 38% mais tempo para ser executada ($RTime_{\Delta}$). Dos programas alterados, o *avro* sofreu os maiores impactos: seus testes demoraram cerca do dobro do tempo para serem executados. Também foi observada uma maior pressão de memória (*MP*). Neste caso, porém, a piora foi menos aguda.

Figura 8.3: Resultados da execução do *microbenchmark*.

No intuito de se investigar as causas da perda de desempenho apresentada, foi criado um mini-experimento, no qual instâncias de uma classe (Figura 8.4) composta por cinco atributos são comparadas em um laço (linhas de 20 à 24). Através da monitoração da execução desse mini-experimento pela ferramenta VisualVM¹¹, obteve-se obter os resultados dispostos na Figura 8.3, onde é evidenciado que cerca de 53% do tempo decorrido durante a execução é gasto nas buscas por atributos

¹¹<https://visualvm.github.io/>.

em operandos. Conforme descrito na Seção 7.1, essa busca de valores se dá através de técnicas de programação reflexiva, técnicas essas que notoriamente têm um desempenho inferior.

```

1 @Comparable
2 public class Bean {
3
4     @ComparableAttribute
5     private Long x = System.currentTimeMillis() << 1;
6
7     @ComparableAttribute
8     private Long y = System.currentTimeMillis() << 2;
9
10    @ComparableAttribute
11    private Long z = System.currentTimeMillis() << 3;
12
13    @ComparableAttribute
14    private Long w = System.currentTimeMillis() << 4;
15
16    @ComparableAttribute
17    private Long k = System.currentTimeMillis() << 5;
18
19    public static void testMicrobench() throws Exception {
20        for (int i=0; i<5000000; i++) {
21            Bean b1 = new Bean();
22            Bean b2 = new Bean();
23            b1.equals(b2);
24        }
25    }
26 }

```

Figura 8.4: Classe utilizada no *microbenchmark*.

8.4.2 Manutenibilidade

No que tange a manutenibilidade, houve uma melhora marginal. Foi observada uma queda de em média 1% tanto na complexidade ciclomática quanto no número de linhas de código (CCG_{Δ} e $KLOC_{\Delta}$, respectivamente). O *fop* e o *lucene* foram os programas que tiveram mais ganhos, ambos com uma redução de quase 3% de sua complexidade ciclomática.

8.4.3 Adequação Funcional

Como esperado, a adoção da proposta corrigiu cerca de 75% das violações encontradas pelo Findbugs – uma melhoria considerável da corretude. Tanto o *avro* e o *xalan* foram pontos fora da curva. O *avro* exibiu um número especialmente alto de violações (FV_B), que foram quase que completamente corrigidas (FV_M).

```
36 public class PDFColor extends PDFPathPaint {
37     // could be 3.0 as well.
38     private static double blackFactor = 2.0;
39     private double red = 1.0;
40     private double green = 1.0;
41     private double blue = 1.0;
42
43     private double cyan = 1.0;
44     private double magenta = 1.0;
45     private double yellow = 1.0;
46     private double black = 1.0;
47
48     ...
49
550 public boolean equals(Object obj) {
551     if (!(obj instanceof PDFColor)) {
552         return false;
553     }
554     PDFColor color = (PDFColor) obj;
555
556     if (color.red == this.red
557         && color.green == this.green
558         && color.blue == this.blue) {
559         return true;
560     }
561     return false;
562 }
563 }
```

Figura 8.5: Trecho do arquivo PDFColor.java do *fop*.

Os exemplos das Figuras 8.5 e 8.6 exemplificam defeitos corrigidos com o uso da proposta deste trabalho. No exemplo da Figura 8.5, são feitas nas linhas 556, 557 e 558 comparações de valores em ponto flutuante. A documentação do Findbugs (2021, tradução do autor) faz a seguinte consideração quanto ao defeito:

“Esta operação compara dois valores de ponto flutuante. Dado que cálculos baseados em valores de ponto flutuante podem necessitar de arredondamento, seus resultados podem não ser exatos. Para valores que devem ser precisos (como valores monetários), considere usar tipos de precisão arbitrária como o `BigDecimal`. Para valores que não precisem ser precisos, considere a comparação baseada em faixas (e.g.: `if (Math.abs(x - y) < .000000)`). Consultar a Sessão 4.2.4 da especificação da linguagem Java.”

Já o exemplo da Figura 8.6 mostra um defeito um pouco mais sutil (e possivelmente mais grave). O método `addGState`, definido na linha 129, recebe uma instância de `PDFGState` e insere no conjunto armazenado no atributo `gstates`. Conforme indicado na linha 66, o conjunto armazenado em `gstates` é uma instância de `HashSet` – um tipo de coleção que faz uso do método `hashCode` de seus itens. Esse método, porém, não é definido pela classe `PDFGState` (que apenas

```
41 public class PDFResources extends PDFObject {  
    ...  
66     protected Set gstates = new HashSet();  
    ...  
129     public void addGState(PDFGState gs) {  
130         this.gstates.add(gs);  
131     }  
    ...  
287 }
```

Figura 8.6: Trecho do arquivo PDFResources.java do *fop*.

define o método *equals*). Segundo a documentação do Findbugs (2021, tradução do autor):

“Uma classe define o método *equals* mas não o método *hashCode* e logo não cumpre o requisito que objetos iguais devem ter códigos *hash* iguais. Uma instância desta classe é usada em uma estrutura de dados que faz uso de códigos *hash*, fazendo com que a correção desse defeito seja de suma importância.”

8.5 Conclusões do Capítulo

Dado os resultados obtidos pelo experimento, pode-se chegar as seguintes conclusões sob o ponto de vista da eficiência de desempenho.

- A queda de desempenho foi maior que o esperado e pode prejudicar a adoção da proposta apresentada pela indústria;
- Técnicas dinâmicas, como por exemplo a programação reflexiva, ainda têm um impacto grande no desempenho de programas.

Já sob o ponto de vista da manutenibilidade, chegou-se as seguintes conclusões:

- Dada a melhora mínima na complexidade (e por consequência da manutenibilidade), é plausível que a lógica de comparação não tem um impacto tão significativo na complexidade total de um programa. Isto é de fato esperado, uma vez que a lógica de negócio tende a ser mais complexa que código *boilerplate*;

- Os programas analisados tinham implementações mais simples que o necessário das operações de comparação. Essa possibilidade é corroborada pelo aumento significativo observado na corretude.

Por fim, sob o ponto da adequação funcional, chegou-se as seguintes conclusões:

- A proposta apresentada é suficientemente expressiva para a representação da semântica de comparação na maioria dos casos (*xalan* sendo uma exceção, onde a manutenção da simetria envolveria a alteração de uma classe interna da linguagem);
- Após a adoção da solução, os programas alterados tiveram uma diminuição importante nas violações encontradas pela ferramenta Findbugs. Em alguns casos (*avro*, *fop*, e *sunflow*), essa diminuição chegou a cerca de 99%.

É importante ressaltar que a solução dos problemas de desempenho encontrados não fazem parte do escopo deste trabalho. Todavia, são discutidas no Capítulo 9 possíveis formas de mitigação, no âmbito de trabalhos futuros.

9 Considerações Finais

Neste trabalho, é descrita uma proposta para comparação de objetos. Ao contrário dos trabalhos existentes (VAZIRI et al., 2007; RAYSIDE et al., 2009; GRECH; RATHKE; FISCHER, 2010; ZWITSERLOOT; SPILKER, 2009; BOURRILLION; MCMA-NUS, 2014), essa proposta trata de três aspectos da comparação – igualdade, ordem, e *hashing* –, e permite a comparação baseada nas estratégias compatível e incompatível com tipo, conforme definidas por Rupakheti (2010). A proposta em questão é descrita de forma agnóstica à linguagem e tem sua correteza formalmente provada. Baseado nessa proposta, foi desenvolvido um protótipo em Java. Dez programas de código-fonte aberto, pertencentes ao *corpus* da suíte de testes DaCapo (BLACKBURN et al., 2006), são alterados para a adoção do protótipo. Os impactos da adoção são então avaliados sob a ótica de três características de qualidade – Eficiência de Desempenho, Manutenibilidade, e Adequação Funcional –, definidas pela norma *ISO/IEC 25010* (2010).

Foi possível a representação de quase todas as semânticas de comparação encontradas (a única exceção foi encontrada no *Apache Xalan* e requereria alterações à classe *String* pertencente ao JDK). Observou-se vantagens expressivas no uso da solução no tocante à Adequação Funcional – cerca de 75% dos defeitos encontrados foram corrigidos – e ganhos modestos na Manutenibilidade – 1%, mensurados indiretamente através da redução de complexidade. Em contrapartida, quedas expressivas na Eficiência de Desempenho foram encontradas. Os resultados mostraram que a execução de cada programa alterado levou cerca de 38% mais tempo após as alterações. Essa queda de desempenho impede a adoção generalizada da proposta.

Em resumo, conclui-se que um mecanismo de comparação de objetos que une os três aspectos da comparação é factível. A solução apresentada foi eficaz na mitigação dos problemas comumente encontrados na implementação da lógica de comparação, ao mesmo tempo resultando em um código mais manutenível e correto. A queda de desempenho apresentada, porém deve ser endereçada antes que a solução seja adotada em larga escala.

9.1 Trabalhos Futuros

Durante esta pesquisa foram encontrados alguns tópicos que são interessantes se explorar no futuro.

- Estudar uma solução que impede a inclusão de objetos mutáveis em coleções baseadas em igualdade. Uma biblioteca de coleções alternativa combinada a um sistema de tipo capaz de identificar objetos imutáveis parece viável, a primeira vista;
- Estudos preliminares indicaram o uso de técnicas de programação reflexiva como a principal causa para a queda de desempenho. Logo, um possível trabalho futuro é a substituição de tais técnicas por técnicas como a geração de *byte-code* em tempo de execução;
- Ainda no tocante ao desempenho, uma outra possível otimização é a adaptação das técnicas de reordenação das verificações nas operações de comparação descrita por Grech, Rathke e Fischer (2010). Deve-se levar em conta, porém, que o cálculo da ordem não é comutativo.

9.2 Artigos Publicados

Foram publicados dois artigos contemplando a proposta deste trabalho:

- SOFIATO, B.; SIQUEIRA, F. L.; ROCHA, R. L. de A. Towards a declarative approach to object comparison. In: *Proceedings of the 35th Italian Conference on Computational Logic - CILC 2020*. Rende, Italy: CEUR Workshop Proceedings, 2020. p. 297–311;
- SOFIATO, B.; SIQUEIRA, F. L.; ROCHA, R. L. de A. Towards an order-aware object comparison. In: *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity*. New York, NY, USA: Association for Computing Machinery, 2020. (SBLP '20), p. 64–71. ISBN 9781450389433. Disponível em: <<https://doi.org/10.1145/3427081.3427090>>.

O primeiro artigo trata da descrição da proposta de forma agnóstica à linguagem, sua formalização e subsequente prova. A formalização descrita no artigo foi acrescida de regras de reescrita relativas ao tratamento de ciclos na dissertação (algo apontado como trabalho futuro no artigo).

Já o segundo artigo publicado tem um ponto de vista menos teórico e mais prático. Seu foco é na descrição informal da proposta e na descrição do protótipo discutido nesta dissertação. O avaliação do protótipo (e a análise dos resultados obtidos) também é debatida por esse artigo.

Referências Bibliográficas

- BAKER, H. G. Equal rights for functional objects or, the more things change, the more they are the same. *ACM SIGPLAN OOPS Messenger*, ACM, v. 4, n. 4, p. 2–27, 1993.
- BANKER, R. D. et al. Software complexity and maintenance costs. *Communications of the ACM*, Association for Computing Machinery, Inc., v. 36, n. 11, p. 81–95, 1993.
- BLACKBURN, M.; GRID, G. Five ways to reduce data center server power consumption. *The Green Grid*, v. 42, p. 12, 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 Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, 2006. p. 169–190.
- BLOCH, J. *Effective Java*. Third edition. Boston, MA, USA: Addison-Wesley, 2018. ISBN 9780134685991.
- BOURRILLION, K.; MCMANUS, E. *google/auto*. 2014. Disponível em: <<https://github.com/google/auto/tree/master/value>>.
- BRUCE, K. et al. On binary methods. *Theory and Practice of Object Systems*, Wiley Online Library, v. 1, n. 3, p. 221–242, 1995.
- CARDELLI, L.; MITCHELL, J. C. Operations on records. *Mathematical structures in computer science*, Cambridge University Press, v. 1, n. 1, p. 3–48, 1991.
- CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, IEEE, v. 20, n. 6, p. 476–493, 1994.
- CORMEN, T. H. et al. *Introduction to algorithms*. Cambridge, MA, USA: MIT press, 2009.
- DATE, C. J. *A guide to the SQL standard: a user's guide to the standard database language SQL*. Boston, MA, USA: Addison-Wesley Professional, 1997.
- EDEN, A. H. Three paradigms of computer science. *Minds and machines*, Springer, v. 17, n. 2, p. 135–167, 2007.
- EVANS, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA, USA: Addison-Wesley, 2004.
- FINDBUGS. 2021. <http://findbugs.sourceforge.net/>. [Online; accessed 09-April-2021].

- FLANAGAN, D.; MATSUMOTO, Y. *The Ruby Programming Language: Everything You Need to Know*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2008.
- 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.
- FREGE, G. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Halle: L. Nebert, 1879.
- GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.
- GOSLING, J. et al. *The Java language specification, Java SE 8 Edition*. Boston, MA, USA: Addison-Wesley Professional, 2000.
- GRECH, N.; RATHKE, J.; FISCHER, B. Generating correct and efficient equality and hashing methods using jequalitygen. In: *Workshop on Generative Technologies*. Paphos, Cyprus: Elsevier, 2010.
- GRUBB, P.; TAKANG, A. A. *Software maintenance: concepts and practice*. Singapore: World Scientific, 2003.
- HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. Automata theory, languages, and computation. *International Edition*, 2006.
- HUNT, A.; THOMAS, D. *The pragmatic programmer: from journeyman to master*. Boston, MA, USA: Addison-Wesley Professional, 2000.
- IGARASHI, A.; PIERCE, B. C.; WADLER, P. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM, v. 23, n. 3, p. 396–450, 2001.
- INGALLS, D. H. A simple technique for handling multiple polymorphism. In: *ACM SIGPLAN Notices*. New York, NY, USA: Association for Computing Machinery, 1986. v. 21, n. 11, p. 347–349.
- ISO. *ISO/IEC 14882:N4778 Information technology – Programming languages – C++*. Geneva, Switzerland: International Organization for Standardization, 2018.
- ISO/IEC. *ISO/IEC 25010 System and software quality models*. [S.l.], 2010.
- Java-NCSS. 2021. <http://www.kclee.de/clemens/java/javancss/>. [Online; accessed 09-April-2021].
- KICZALES, G. et al. Aspect-oriented programming. *ECOOP'97—Object-oriented programming*, Springer, p. 220–242, 1997.
- KLEENE, S. C. et al. *Introduction to metamathematics*. New York: van Nostrand, 1952.
- KRASNER, H. The cost of poor quality software in the us: A 2020 report. *Consortium for IT Software Quality, Tech. Rep*, 2020.

- LEIBNIZ, G. W. *Philosophical Papers and Letters: A Selection (Synthese Historical Library 2)*. Dordrecht, Germany: D. Reidel/Springer, 1976. ISBN 902770693X.
- LINUXMM. Dec 2017. Disponível em: <<https://linux-mm.org/LinuxMM>>.
- LISKOV, B.; GUTTAG, J. et al. *Abstraction and specification in program development*. Cambridge, MA, USA: MIT press, 1986.
- LISKOV, B. H.; WING, J. M. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM, v. 16, n. 6, p. 1811–1841, 1994.
- LLOYD, J. Practical advantages of declarative programming. In: *Proceedings of Joint Conference on Declarative Programming*. Peniscola, Spain: Universidad Politécnica de Valencia, 1994. p. 3–17.
- MCCABE, T. J. A complexity measure. *IEEE Transactions on software Engineering*, IEEE, n. 4, p. 308–320, 1976.
- 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.
- MEYER, B. *Eiffel: the language*. Hoboken, NJ, USA: Prentice-Hall, Inc., 1992.
- MUSCHEVICI, R. et al. Multiple dispatch in practice. In: *ACM. Acm sigplan notices*. New York, NY, USA: ACM, 2008. v. 43, n. 10, p. 563–582.
- NELSON, S.; PEARCE, D.; NOBLE, J. Understanding the impact of collection contracts on design. *Objects, Models, Components, Patterns*, Springer, p. 61–78, 2010.
- ODERSKY, M. *Programming in Scala*. 2nd ed. ed. Walnut Creek, CA, USA: Artima, 2010. ISBN 9780981531649.
- PACHECO, C. et al. Feedback-directed random test generation. In: *Proceedings of the 29th international conference on Software Engineering*. Minneapolis, MI, USA: IEEE Computer Society, 2007. p. 75–84.
- PARNAS, D. L. Designing software for ease of extension and contraction. *IEEE transactions on software engineering*, IEEE, n. 2, p. 128–138, 1979.
- PEANO, G. *Arithmetices Principia Novo Methodo Exposita*. Bocca, 1889. Disponível em: <<http://eudml.org/doc/203509>>.
- PLOTKIN, G. D. Structural operational semantics. *Aarhus University, Denmark*, 1981.
- RAYSIDE, D. et al. Equality and hashing for (almost) free: Generating implementations from abstraction functions. In: *Proceedings of the 31st International Conference on Software Engineering*. Vancouver, British Columbia, Canada: ACM, 2009. p. 342–352.
- RUPAKHETI, C. R. *A Path-Based Approach for Analyzing Object Equality in Java*. Tese (Doutorado) — Clarkson University, 2010.

- 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*. Ontario, Canada: Association for Computing Machinery, 2008. p. 9.
- RUPAKHETI, C. R.; HOU, D. An abstraction-oriented, path-based approach for analyzing object equality in java. In: *17th Working Conference on Reverse Engineering (WCRE 2010)*. Los Alamitos, CA, USA: IEEE Computer Society, 2010. p. 205–214.
- SCHMIDT, G.; STRÖHLEIN, T. *Relations and graphs: discrete mathematics for computer scientists*. Berlin, Germany: Springer Science & Business Media, 2012.
- SCHROEDER, M. A practical guide to object-oriented metrics. *IT professional*, IEEE, v. 1, n. 6, p. 30–36, 1999.
- SHAMS, Z.; EDWARDS, S. H. Reflection support: Java reflection made easy. *The Open Software Engineering Journal*, v. 7, n. 1, 2013.
- SHEN, S. et al. *Basic set theory*. Providence, RI, USA: American Mathematical Soc., 2002.
- SILVA, T. M. et al. Automated design tests to check hibernate design recommendations. In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. Salvador, Brazil: Association for Computing Machinery, 2019. p. 94–103.
- SOBEL, J. M.; FRIEDMAN, D. P. An introduction to reflection-oriented programming. In: *Proceedings of reflection*. [S.l.: s.n.], 1996. v. 96.
- SUBRAMANIAM, V. *Effective implementation of equals method*. 2020. Disponível em: <<https://agiledeveloper.com/articles/equals062002.htm>>.
- TARSKI, A. *Introduction to Logic and to the Methodology of the Deductive Sciences (Oxford Logic Guides)*. Oxford, United Kingdom: Oxford University Press, 1994. ISBN 019504472X.
- TIOBE. 2021. <https://www.tiobe.com/tiobe-index/>. [Online; accessed 15-March-2021].
- VAZIRI, M. et al. Declarative object identity using relation types. In: *ECOOP 2007 – Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. v. 7, p. 54–78.
- WEGNER, P. Research paradigms in computer science. In: IEEE COMPUTER SOCIETY PRESS. *Proceedings of the 2nd international Conference on Software Engineering*. San Francisco, CA, USA, 1976. p. 322–330.
- WINCKLER, M.; BERNHAUPT, R.; PONTICO, F. Challenges for the development of user interface pattern languages: A case study on the e-government domain. *IADIS International Journal on WWW/Internet*, v. 8, n. 2, p. 59–84, 2010.
- ZERMELO, E. Beweis, daß jede menge wohlgeordnet werden kann. *Mathematische Annalen*, Springer, v. 59, n. 4, p. 514–516, 1904.

ZOOK, D.; HUANG, S. S.; SMARAGDAKIS, Y. Generating aspectj programs with meta-aspectj. In: *Generative Programming and Component Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. v. 4, p. 1–18.

ZWITSERLOOT, R.; SPILKER, R. *Project lombok*. 2009. [Online; accessed 09-Mar-2021]. Disponível em: <<https://projectlombok.org/>>.