Definição de clusters para classificação do uso de anotações em código Java

Phyllipe Lima[1], **Eduardo Guerra**[1], **Paulo Meirelles**[2]

[1]Instituto Nacional de Pesquisas Espaciais (INPE)
Av. dos Astronautas, 1758, Jardim da Granja, 12227-010 – São José dos Campos – SP

[2]Instituto de Matemática e Estatística (IME) – Universidade de São Paulo (USP)
Rua do Matão, 1010, Cidade Universitária, 05508-090 – São Paulo – SP

phyllipe_slf@yahoo.com.br,guerraem@gmail.com,paulormm@ime.usp.br

Abstract. *Code annotation allows the introduction of custom metadata on programming elements directly on the source code. In Java, it is used extensively by enterprise applications and frameworks. Although it has become popular, there are few studies dedicated to the analysis and assessment of its usage. For instance, it is unknown the impact that annotations brings to software maintenance. This paper aims to classify object oriented classes based on its annotations usage, through a Kohonen Self Organizing Map. The input of the map are metrics values related to annotations. As a result, this approach allowed the identification of distinct classes.*

Resumo. *Anotações de código permitem introduzir metadados sobre elementos de programação diretamente no código fonte. Na linguagem Java, ela é utilizada extensivamente por aplicações e frameworks corporativos. Apesar da sua popularidade, existem poucos estudos dedicados à análise e avaliação do seu uso. Por exemplo, não se sabe qual o impacto de anotações na manutenção do código fonte. Este trabalho tem o objetivo de classificar classes de acordo com seu uso de anotações, utilizando um Mapa de Kohonen para visualizar os grupos. Para esse agrupamento, são utilizadas como entrada os valores de métricas relacionadas a anotações. Como resultado, a abordagem proposta permitiu identificar grupos bem distintos de classes, classificando-as dentro ou fora do uso comum de anotações dentro do universo de projetos analisados.*

1. Introdução

As anotações estão presentes em grande parte de APIs Java para aplicações corporativas [JSR 2007]. Como exemplo, a API EJB utiliza anotações para configurar transações e restrições de segurança. A API JPA utiliza anotações para o mapeamento objeto-relacional. Um estudo realizado em 2011 [Rocha and Valente 2011] verificou que, de 106 projetos da base Qualitas Corpus[1], 65 utilizam anotações de alguma forma. A partir desses dados, é possível perceber a relevância de estudos a respeito de anotações.

[1]`http://qualitascorpus.com`

Não existe um estudo que trata da evolução do código fonte do ponto de vista das anotações, ou seja, qual o impacto delas na manutenção do software, quais boas práticas em códigos anotados, entre outros. Uma das formas de avaliar a qualidade do software é a partir de extração de informações do código-fonte através de métricas. Existem métricas de software bem conhecidas como LOC (Linhas de código), NOM (Número de Métodos), CYCLO (Complexidade Ciclomática), LCOM (Falta de Coesão), CBO (Acoplamento entre Objetos) e muitas outras [Lanza and Marinescu 2006]. Entretanto, apesar de terem sido propostas métricas de software para anotações [Correia et al. 2010], ainda existe uma carência de trabalhos a respeito delas.

A partir das métricas a respeito de anotações, pode-se classificar as classes de acordo com seus valores. O objetivo deste trabalho é propor uma abordagem de classificação e visualização de grupos de classes de acordo com suas características no uso de anotações utilizando Mapa de Kohonen [Kohonen 1997]. Essa técnica permite visualizar grupos, ou *clusters*, de dados onde não se conhece previamente nenhum padrão de comportamento dos dados de entrada, permitindo identificar classes que possam ser consideradas *outliers*. Para este trabalho a entrada é constituída pelos valores das métricas de anotações.

Para este estudo, 25 projetos de software livre escritos em Java foram analisados, num total de 24.947 classes anotadas. Os dados foram coletados usando um *plugin* para Eclipse chamado *Annotation Sniffer*[2], desenvolvido por nós. Após a coleta dos dados, o Mapa de Kohonen foi utilizado para o agrupamento. Como resultado, identificou-se cerca de 70% das classes possuem um comportamento comum, e as restantes puderam ser classificadas em outros 3 grupos pequenos com comportamentos distintos.

2. Métricas Propostas

A seguir são apresentadas quatro métricas de anotações utilizadas como base para a extração de dados[3]. Para melhor compreensão, a Figura 1 é usada na explicação de cada uma das métricas.

Anotações por Classe/*Annotations in Class* (AC): Esta métrica faz a contagem de quantas anotações estão presentes na classe. Anotações aninhadas também são consideradas. Na classe apresentada na Figura 1, o valor de AC é 10. Observe que a anotação aninhada *@Annotation1* também é considerada.

Anotações Únicas por Classe/*Unique Annotations in Class* (UAC): Esta métrica é semelhante a AC, mas ela mede a quantidade de anotações distintas na classe. Para duas anotações serem consideradas semelhantes, elas devem possuir o mesmo tipo de anotação e o mesmo valor em todos os seus atributos, incluindo atributos de anotações aninhadas. O objetivo desta métrica é registrar um número de configurações diferentes de anotações em uma mesma classe. Na Figura

[2]`http://github.com/phillima/AnnotationSniffer2.0`

[3]Estas e outras métricas são devidamente propostas e formalizadas em um artigo que está em segunda rodada de revisão pelo *Journal of Systems and Software – JSS*. O objetivo deste trabalho é classificar os projetos baseado nos valores das métricas, por isso é importante apresentar a definição das métricas nesta seção.

```
1   @Annotation1
2   @Annotation2
3   public class ClasseExemplo {...
4
5       @Annotation3 (value1 = 2, value2 = 4)
6       @Annotation4 (value1 = @Annotation1)
7       private int attributo1;
8
9       @Annotation4 (value1 = @Annotation1)
10      private double attributo2;
11
12      @Annotation1
13      @Annotation3(value1 = 2, value2 = 3)
14      public String metodo1(){...}
15
16      public String metodo2(){...}
17
18      public String metodo3(
19         @Annotation1 int x){...}
20
21  }
```

Figura 1. Código Exemplo

1, o valor de UAC é 5. Observe que as duas anotações denominadas *@Annotation3* possuem atributos com valores diferentes, assim são contadas separadamente.

Elementos por Classe/*Number of Elements per Class*(NEC): Esta métrica informa quantos elementos na classe podem ser anotados. Os elementos anotados são: atributos, métodos, a declaração da classe e parâmetros de métodos. Espera-se com essa métrica detectar relações com as demais métricas. Por exemplo, espera-se que as métricas AC e NEC possuam valores próximos, mas isso pode não ser verdade. Esta métrica permitirá verificar esse tipo de informação. No código exemplo da Figura 1 o valor de NEC é 7.

Elementos Anotados por Classe/*Number of Elements Annotated per Class*(NEAC): Esta métrica é semelhante a NEC, porém ela informa quantos elementos na classe estão de fato anotados. O uso combinado de NEC com NEAC informará a razão de elementos anotados na classe. Um alto número de NEC com NEAC baixo pode indicar um problema semelhante ao *bad smell* popularmente conhecido como *God Class* [Lanza and Marinescu 2006], mas em uma versão para anotações. Este *bad smell* se refere a classes que possuem muitas tarefas e geralmente são extensas. Nessa analogia com anotações, tem-se poucos elementos com várias configurações de metadados. Na Figura 1 o valor de NEAC é 5.

Classe	AC	UAC	NEC	NEAC
ClasseExemplo	11	5	7	5

Tabela 1. Descrição resumida das métricas.

Por fim, os valores das métricas para o código da Figura 1 são sumarizados na Tabela 1.

3. Coleta de Dados

Para a extração das métricas, 25 projetos de software livre escritos em Java foram selecionados. Posteriormente, essa amostra foi validada com base na abordagem de [Nagappan et al. 2013], onde o objetivo é determinar a semelhança e diversidade dentro dos projetos escolhidos. Para essa validação, foram definidas 3 dimensões, ou seja, propriedades dos projetos: Tipo (*framework* e *application*), Porcentagem de Classes Anotadas (PAC) e Número de Linhas de Código (LOC). Dos 25 projetos, obteve-se uma amostra de 24.947 classes anotadas, conforme apresentado na Tabela 2.

Tabela 2. Projetos selecionados.

Projeto	Repositório	Versão	Tipo	PAC	LOC
Agilefant	`github.com/Agilefant/agilefant`	3.5.4	Application	363 (70,6%)	43.539
ANTLR	`github.com/antlr/antlr4`	4.5.3	Application	334 (64,0%)	101.600
Apache Derby	`github.com/apache/derby`	10.12.1.1	Application	311 (10,6%)	689.869
Apache Isis	`github.com/apache/isis`	1.13	Framework	2.124 (63,7%)	163.665
Apache Tapestry	`github.com/apache/tapestry-5`	5.4.1	Framework	1.676 (54,7%)	156.450
Apache Tomcat	`github.com/apache/tomcat`	9.0.0	Framework	1.349 (66,4%)	300.819
ArgoUML	`argouml.tigris.org/source/browse/argouml/trunk/src`	0.34	Application	610 (31,7%)	195.670
Eclipse CheckStyle	`github.com/acanda/eclipse-cs`	6.2.0	Application	52 (21,8%)	28.123
Dependometer	`github.com/dheraclio/dependometer`	1.2.9	Application	44 (10,5%)	20.453
ElasticSearch	`github.com/elastic/elasticsearch`	5.0.0-rc1	Application	3.537 (69,4%)	615.637
Hibernate Commons	`github.com/hibernate/hibernate-commons-annotations`	4.0.5	Framework	20 (25,6%)	2.812
Hibernate Core	`github.com/hibernate/hibernate-orm`	5.2.0	Framework	6.140 (68,0%)	593.854
JChemPaint	`github.com/JChemPaint/jchempaint`	3.3-1210	Application	57 (18,8%)	27.371
Jenkins	`github.com/jenkinsci/jenkins`	2.25	Application	1.052 (76,7%)	124.576
JGit	`github.com/eclipse/jgit`	4.5.0	Framework	841 (64,8%)	173.681
JMock	`github.com/jmock-developers/jmock-library`	2.8.2	Framework	79 (35,0%)	9.580
JUnit	`github.com/junit-team/junit5`	5.0.0-M2	Framework	373 (85,0%)	25.935
Lombok	`github.com/rzwitserloot/lombok`	1.16.10	Framework	596 (71,0%)	50.324
Megamek	`github.com/MegaMek/megamek`	0.41.24	Application	597 (31,7%)	306.210
MetricMiner	`github.com/mauricioaniche/metricminer2`	2.6	Framework	53 (72,6%)	23.602
OpenCMS	`github.com/alkacon/opencms-core`	10.0.1	Application	1.707 (48,2%)	476.074
OVal	`github.com/sebthom/oval`	1.86	Framework	184 (54,4%)	17.381
Spring Integration	`github.com/spring-projects/spring-integration`	4.3.4	Framework	1.886 (75,0%)	208.750
VRaptor	`github.com/caelum/vraptor4`	4.2.0-RC4	Application	445 (80,8%)	542.030
VoltDB	`github.com/VoltDB/voltdb`	6.5.1	Framework	517 (44,7%)	26.660

Os valores das métricas foram extraídos para formar o vetor de dados de entrada para o Mapa de *Kohonen*. Para automatizar o processo de extração dos valores das métricas, um software livre chamado *Annotation Sniffer* (ASniffer) foi desenvolvido. O ASniffer é um *plugin* para o Eclipse[4]. Após a extração, um relatório é gerado através de um arquivo XML. Em suma, o *plugin* busca no *Workspace* do Eclipse todos os projetos Java abertos, e coleta as métricas para todos. É criado um arquivo XML para cada métrica para cada projeto encontrado. O *script* XMLtoCSV[5] transforma os dados presentes no arquivo XML para o formato CSV. Esse arquivo CSV é usado em uma etapa posterior.

4. Experimentos e Resultados

O Mapa de Kohonen foi criado por Teuvo Kohonen em 1982, sendo simples e com a capacidade de organizar os dados em grupos de acordo com suas relações. É uma rede neural do tipo competitiva com aprendizado não supervisionado [Kohonen 1997]. Os dados de entrada são agrupados de acordo com o padrão existente entre

[4]`http://www.eclipse.org`

[5]`http://github.com/phillima/XMLToCSV`

eles. A rede ajusta os pesos na medida em que novas amostras são apresentadas em sua entrada, caracterizando o processo de aprendizagem não supervisionado [Soares and Souza 2016]. Após esse processo, toda vez que uma nova amostra for colocada na entrada da rede, determinados neurônios serão ativados com mais intensidade. Essa região de ativação pode ser vista como o grupo onde a amostra de entrada pertence. Uma vez que desconhecemos a relação existente entre os valores das métricas, o Mapa é uma solução para identificar essas relações.

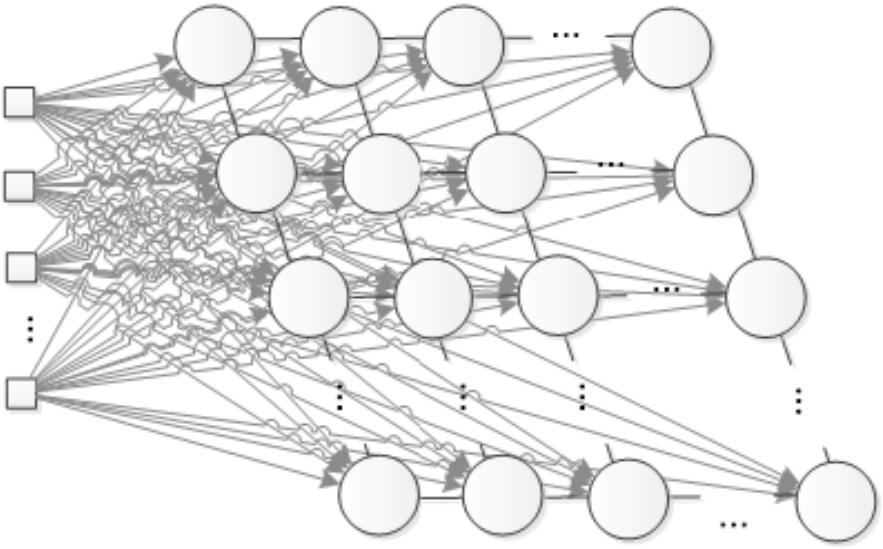

Figura 2. Exemplo de uma Rede Kohonen 2D.

A Figura 2 apresenta um exemplo de um Mapa de Kohonen 2D. Observe que cada entrada é conectada em cada um dos neurônios. A entrada do Mapa, é um vetor que contém o valor das métricas de cada classe, como temos quatro métricas cada vetor possui quatro posições. Durante o processo de treinamento, 24.947 vetores são apresentados para a rede, um vetor para cada classe. Cada vetor de entrada ativará todos os neurônios, mas com intensidade diferente. O neurônio vencedor, ou seja, aquele que obteve o maior valor para aquela entrada, irá atualizar o seu peso. O objetivo é que cada neurônio atualize o seu peso até que se comporte como um centróide para um grupo de dados. De forma resumida, os vetores que tiverem o mesmo neurônio como centróide podem ser considerados parte do mesmo *cluster*. Assim, podemos visualizar quais os valores das métricas presentes nos vetores que ativam neurônios semelhantes. A implementação do Mapa de Kohonen foi feita utilizando a linguagem R. Já existe a biblioteca *kohonen*[6] com uma implementação pronta. É necessário apenas informar o conjunto de dados de entrada, que neste trabalho constituiu 24.947 vetores com quatro posições cada.

O primeiro ponto para a implementação é definir o tamanho do mapa a ser utilizado, ou seja, quantos neurônios ele possuirá. É importante variar este número até se obter um mapa onde os neurônios contenham pesos o mais distante possível entre eles, assim será possível distinguir grupos de dados bem definidos. Neste trabalho, o mapa teve formatos do tipo 3x3 até 14x14. Ao se testar estes tamanhos

[6]`http://cran.r-project.org/web/packages/kohonen/kohonen.pdf`

de mapas, não houve variação significativa da disposição dos grupos. Assim, chegou-se no mapa final com 16 neurônios, ou seja, 4x4.

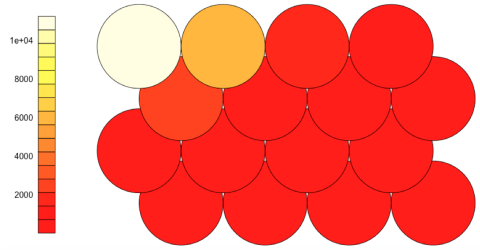

Figura 3. Mapa de Pontuação do Kohonen.

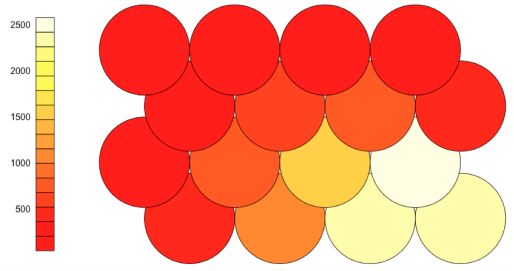

Figura 4. Mapa de Distância entre os neurônios.

Com o mapa pronto, diversas informações podem ser observadas. A primeira é a pontuação, ou seja, quantas amostras, ou vetores, ativaram um neurônio específico. A Figura 3 apresenta essa informação. Os neurônios presentes no canto esquerdo superior (neurônios mais claros) concentram grande parte dos dados, mais de 70%. Em um primeiro momento essa informação isolada não traz muita informação da quantidade de grupos existentes.

O mapa ilustrado na Figura 4 mostra a distância entre os neurônios. As cores mais claras sugerem neurônios bastante distante dos vizinhos, e cores mais vermelhas sugerem neurônios mais próximos. Nota-se que no canto esquerdo superior do mapa tem-se um *cluster* bastante homogêneo, e essa mesma parte do mapa possui a maior parte dos dados, de acordo com a Figura 3. No canto direito inferior tem-se neurônios mais heterogêneos, o que pode significar a presença de dois ou mais *clusters* nessa região. E observando a Figura 3 nota-se que no canto direito inferior há uma quantidade menor de dados. Com isso já é possível detectar que no geral haverá poucos grupos, onde um grupo contém uma grande quantidade de dados e os demais grupos são menores.

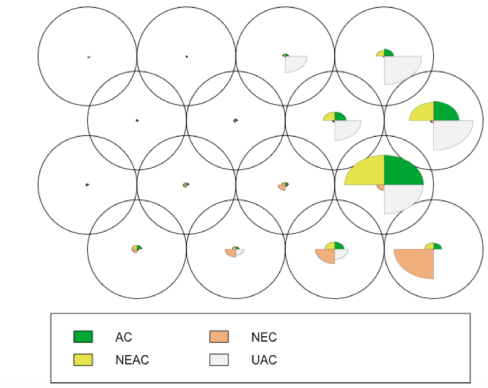

Figura 5. Mapa de Distribuição das Variáveis.

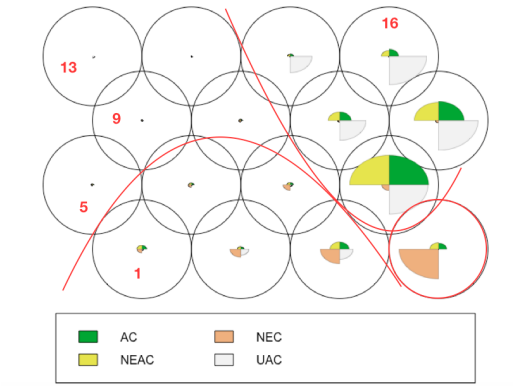

Figura 6. Clusterização dos dados.

O vetor de entrada é constituído pelos valores das métricas, e o mapa apresentado na Figura 5 mostra como as métricas estão distribuídas pelo mapa. No canto direito inferior há forte presença de elementos (NEC) e valores razoáveis de

AC e NEAC. No canto esquerdo superior, o grupo de dados dominante, a presença das variáveis é bastante pequena, sugerindo que as grandes quantidades de classes analisadas possuem poucos elementos e poucas anotações. No canto superior direito tem-se uma presença maior de UAC, ou seja, são classes com poucas anotações repetidas.

Cluster	Neurônios	AC	NEAC	NEC	UAC	Frequência
1	5,9,10,13 e 14	<43	<43	<120	<20	Classes Muito Frequentes
2	4	<40	<40	>1500	<2	Classes Raras
3	1,2,3,6 e 7	<140	<140	200 a 1000	<50	Classes Pouco Frequentes
4	8, 11, 12, 15 e 16	3 a 170	3 a 170	60 a 200	<59	Classes Pouco Frequentes

Tabela 3. Clusters obtidos e suas características.

Usando o mapa de pontuação da Figura 3, é possível saber exatamente quantas amostras (neste caso vetores contendo valores das métricas de "classes Java") estão em cada neurônio e também suas características. Assim, obteve-se um mapa com quatro *clusters* diferentes usando a análise empírica e verificando as características dos dados presentes em cada neurônio com auxílio da Figura 5. A Figura 6 ilustra esse mapa de rótulos. Essa Figura contém também a numeração dos neurônios. Por fim, a Tabela 3 apresenta as características das amostras pertencentes a cada *cluster*, além de mostrar quais neurônios compõem o *cluster*.

5. Conclusão

Existem alguns trabalhos na literatura descrevendo o uso de anotações na solução de problemas de diversos tipos de domínios. Na engenharia de software, algumas soluções aplicam anotações na implementação de padrões de projeto [Meffert 2006], ou para permitir a refatoração arquitetural [Krahn and Rumpe 2006]. Entretanto, poucos trabalhos avaliaram o uso de anotações propriamente ditas, focando em princípios de *design* para modelos de metadados ou executando estudos para avaliar como as anotações são utilizadas em projetos reais. Em outras palavras, os trabalhos que abordam códigos anotados poderiam prover uma avaliação mais completa se considerassem o uso de métricas para anotações baseadas nas próprias anotações e elementos anotados.

Este trabalho se diferencia por considerar métricas de software para anotações. Com isso, analisamos projetos de software livre escritos em Java para identificarmos como as anotações são utilizadas. Após as coletas de quatro métricas específicas para anotações, utilizamos o Mapa de Kohonen para auxiliar na identificação de *clusters*. Foi possível identificar quatro *clusters*. O *Cluster* 1, "classes frequentes", são responsáveis por mais de 70% dos dados. Assim, conclui-se que, no geral, as classes são pequenas, e a maioria das anotações estão presentes em pelo menos 50% dos elementos. Além disso, essas anotações são em sua maioria repetidas. Nota-se a existência de algumas classes "pouco frequentes", com muitos elementos (> 1500) e pouquíssimos anotados. A princípio, não se pode concluir a existência ou não de algum problema de configuração nessas classes, pois elas podem ter seguido um critério de desenvolvimento pertencente aquele projeto específico.

Como trabalho futuro, pode-se utilizar outras técnicas de visualização como coordenadas paralelas, dendrograma, e outras. Pode-se também flexibilizar o próprio

Mapa de Kohonen. Assim, torna-se possível comparar e analisar diversas técnicas com objetivo de extrair o resultado que melhor reflete o uso de anotações em projetos reais. Por fim, estudos futuros podem combinar os valores das métricas e o agrupamento das classes Java para se definir boas práticas e identificar problemas de configuração ou *design* (*bad smells*).

5.1. Ameaças a validade

Como não existe ferramenta semelhante ao ASniffer, sua acurácia foi medida manualmente, o que pode trazer impactos na precisão das métricas extraídas. As métricas utilizado fazem parte de um novo conjunto, portanto o seu uso está restrito às análises feitas pelos autores deste trabalho. Não foi utilizado outra técnica de visualização, o que está previsto como trabalho futuro.

5.2. Agradecimento

Este trabalho teve apoio do CNPq (445562/2014-5) e FAPESP (2015/16487-1).

Referências

[Correia et al. 2010] Correia, D. A., Guerra, E. M., Fernandes, C. T., and Silveira, F. (2010). Quality improvement in annotated code. *CLEI Electronic Journal*, pages 1–7.

[JSR 2007] JSR (2007). JSR 220: Enterprise javabeans 3.0.

[Kohonen 1997] Kohonen, T. (1997). *Self-Organizing Maps*. Springer, 2nd edition.

[Krahn and Rumpe 2006] Krahn, H. and Rumpe, B. (2006). Towards enabling architectural refactorings through source code annotations. *Lecture Notes in Informatics*, P-82:203–212.

[Lanza and Marinescu 2006] Lanza, M. and Marinescu, R. (2006). *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer.

[Meffert 2006] Meffert, K. (2006). Supporting design patterns with annotations. In *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, pages 8 pp.–445.

[Nagappan et al. 2013] Nagappan, M., Zimmermann, T., and Bird, C. (2013). Diversity in software engineering research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 466–476, New York, NY, USA. ACM.

[Rocha and Valente 2011] Rocha, H. and Valente, H. (2011). How annotations are used in java: An empirical study. In *23rd International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 426–431.

[Soares and Souza 2016] Soares, F. M. and Souza, A. M. (2016). *Neural Networks Programming with Java*. Packt, 1st edition.