BOHR - Uma Ferramenta para a Identificação de Átomos de Confusão em Códigos Java

Wendell Mendes
wendell.mendes@sti.ufc.br
Universidade Federal do Ceará
Fortaleza, Ceará, Brasil

Windson Viana
windson@virtual.ufc.br
Universidade Federal do Ceará
Fortaleza, Ceará, Brasil

Lincoln Rocha
lincoln@dc.ufc.br
Universidade Federal do Ceará
Fortaleza, Ceará, Brasil

RESUMO

A atividade de compreensão do código-fonte é fundamental no desenvolvimento e manutenção de software. Nesse contexto, Átomos de Confusão (AC) surgem como a menor porção de código capaz de causar confusão em desenvolvedores nesse processo. Neste artigo, é apresentado o BOHR - The Atoms of Confusion Hunter, uma ferramenta que tem como objetivos: (i) auxiliar a identificação de AC em sistemas Java; (ii) fornecer relatórios sobre a prevalência desses AC; e (iii) fornecer uma API para o desenvolvimento de novos buscadores customizados para a captura de novos AC e, também, melhorias em suas identificações. Nesta primeira versão, o BOHR é capaz de detectar 8 dos 13 tipos de AC apontados por Langhout e Aniche [12]. Uma avaliação preliminar do BOHR foi conduzida em três sistemas reais, de código aberto, largamente adotados pela comunidade de desenvolvimento de softwares Java (Picasso, greenDAO e Socket.IO-client Java). Os resultados iniciais mostraram que o BOHR conseguiu detectar com precisão os AC existentes nesses sistemas analisados, apontando de forma correta os *snippets* dos AC, assim como o seu tipo, nome da classe que era pertencente e o número da linha de código de sua ocorrência. No total, foram encontrados 105 AC, sendo 37 no Picasso, 25 no greenDAO e 43 no Socket.IO-client Java.

PALAVRAS-CHAVE

Compreensão de Código, Átomos de Confusão, Análise Estática de Código

1 INTRODUÇÃO

A compreensão de código é uma atividade na qual engenheiros de software buscam entender o funcionamento de um programa de computador tendo seu código-fonte como principal referência [2]. O aumento do conhecimento sobre o código ajuda na execução de atividades como correção de *bugs*, melhorias, reutilização e documentação [19]. A compreensão de código-fonte é fundamental no desenvolvimento de sistemas, tanto na criação de novas funcionalidades quanto na manutenção das funcionalidades existentes.

Em atividades de desenvolvimento é muito comum que desenvolvedores tenham que lidar com trechos de código que originalmente não foram escritos por eles mesmos. Esse processo cognitivo envolve identificar, compreender e analisar o código criado por terceiros. Nesse contexto, é importante observar que o entendimento humano pode ser distinto do entendimento da máquina em determinados trechos de códigos, causando conclusões errôneas sobre os resultados de suas execuções [9]. Dessa forma, para que sejam realizadas as atividades de desenvolvimento, é necessário que haja o correto entendimento do código a ser modificado.

Estudos anteriores mostram que a compreensão do código é a atividade mais dominante no processo de desenvolvimento, consumindo cerca de 58% do tempo total despendido [13, 21]. Trabalhos anteriores também mostram que diferentes formas de escrever o código afetam o processo de compreensão do programa, códigos confusos acabam causando impactos no processo de desenvolvimento. Dessa forma, revisores de código, frequentemente, não entendem a mudança que está sendo revisada nem o seu contexto [7]. Nesse sentido, determinadas estruturas de código são mais difíceis de entender do que outras [1], assim como algumas práticas de programação afetam a legibilidade do código [6] e, também, desenvolvedores navegam e fazem edições em um ritmo significativamente mais lento [18].

Gopstein et al. [9] identificaram padrões de código responsáveis por gerar confusão em desenvolvedores. Esses padrões foram chamados de **Átomos de Confusão** (AC). Em seu estudo, foram realizadas duas pesquisas com o objetivo de avaliar o impacto desses padrões confusos. Trechos de códigos escritos na linguagem *C* foram analisados, com e sem a presença de AC, comparando a corretude dos resultados de suas execuções. Assim, a pesquisa concluiu que códigos contendo AC dificultam o entendimento quando comparados com códigos com funções equivalentes que não incluem esses átomos. Dessa forma, o trabalho mostrou que a presença desses padrões pode impactar significativamente, não só na corretude, mas também no tempo e esforço para o entendimento de código. Os estudos de AC ainda são recentes. Os impactos causados por esses padrões ainda demandam mais análises no campo de compreensão de código. Nesse contexto, ferramentas para a detecção de AC em sistemas podem impulsionar novos estudos, permitir a detecção de problemas e a consequente melhoria de código de sistemas existentes.

Dentro deste contexto, este trabalho apresenta o BOHR - *The Atoms of Confusion Hunter*. Ele visa a detecção da presença de átomos de confusão em sistemas Java. Nesta primeira versão, o BOHR é capaz de detectar 8 dos 13 tipos de AC em Java, apontados como confusos por Langhout and Aniche [12]. Esses AC impedem uma melhor compreensão de programas ou se apresentam de forma mais difícil de entender quando comparados com suas versões de código traduzidas, sem a presença de átomos. Esses átomos são os seguintes: *Post-Increment/Decrement, Pre-Increment/Decrement, Conditional Operator, Arithmetic as Logic, Logic as Control Flow, Change of Literal Encoding, Omitted Curly Braces* e *Type Conversion*. Nesta primeira versão os casos contemplados do átomo *Type Conversion* são os casos apresentados em [12], que são as conversões entre tipos primitivos de *float* para *int* e de *int* para *byte*.

Essa ferramenta analisa repositórios de software buscando quantificar a ocorrência de AC e identificar a sua localização nas classes

do sistema em análise. O BOHR gera um relatório na forma de arquivo CSV (*Comma-Separated Values*), contendo o tipo do átomo encontrado e o seu trecho de código correspondente, assim como o nome da classe que o contém e a linha de código de sua ocorrência. A partir desse relatório é possível extrair dados que podem substanciar diferentes análises de impactos causados pela presença desses padrões como, por exemplo, a relação desses átomos com a presença de *bugs* ou com a presença de comentários em revisões de código.

Este trabalho apresenta as seguintes contribuições: (i) uma forma automática para identificação de AC em sistemas Java; (ii) geração de relatórios padronizados sobre a prevalência e a localização desses AC no código-fonte do sistema; e (iii) uma API para o desenvolvimento de novos buscadores customizados para a captura de novos padrões de código e de novos AC.

O restante do artigo está organizado da seguinte forma: na Seção 2, a fundamentação teórica e os trabalhos relacionados são apresentados. Em seguida, na Seção 3, a ferramenta proposta é descrita. A avaliação da ferramenta é apresentada na Seção 4. Por fim, na Seção 5, as conclusões e considerações finais são descritas.

2 ÁTOMOS DE CONFUSÃO

Como mencionado anteriormente, Gopstein et al. [9] introduziram o conceito de **Átomos de Confusão** (AC), no qual um átomo de confusão pode ser definido como a menor porção de código capaz de causar confusão em desenvolvedores, causando conclusões errôneas sobre seu comportamento. A hipótese é que a existência desses átomos prejudica o entendimento do código-fonte e pode ocasionar erros no sistema.

Para auxiliar pesquisadores interessados no estudo de AC, Castor [4] aprofundou o conceito, definindo um átomo de confusão como um padrão de código: (i) capaz de ser identificado precisamente; (ii) passível de causar confusão; (iii) substituível por um padrão de código de função equivalente com menos probabilidade de causar confusão; e (iv) indivisível.

Em [9], Gopstein et al. apontaram 15 átomos que, quando presentes no código, causam confusão de forma significativa. Em um trabalho complementar, Gopstein et al. [10] observou uma forte relação entre linhas de código contendo átomos de confusão e a ocorrência de *bugs*, mostrando que *commits* do tipo *bug-fix* removiam mais átomos quando comparados com outros tipos de *commits*. Da mesma forma, foi observado que essas linhas causavam mais confusão, pois costumavam ser mais comentadas que outras. Em um trabalho mais recente, Gopstein et al. [8] realizaram um estudo qualitativo onde foi observado que: estudos quantitativos podem estar subestimando a quantidade de mal-entendidos que ocorrem durante suas avaliações, uma vez que respostas corretas nas tarefas das avaliações não garantem que não houve confusão no processo de compreensão de código-fonte.

Com base no estudo de Gopstein et al. [9], Langhout and Aniche [12] generalizaram o conhecimento sobre átomos de confusão para o contexto da linguagem de programação Java, uma linguagem tradicional das mais utilizadas no mercado de software [3, 14]. Esse estudo analisou e traduziu os 19 átomos de confusão definidos por Gopstein et al. [9], chegando a uma lista de 14 átomos reproduzíveis em Java. A Tabela 1 foi baseada nesse trabalho. Ela apresenta a lista dos 14 átomos e suas respectivas traduções. O estudo avaliou as percepções e os impactos desses padrões confusos em desenvolvedores novatos e apontou que em 7 dos 14 átomos estudados os desenvolvedores são 4.6 até 56 vezes mais propensos a cometerem erros de entendimento. E quando confrontados com as duas versões de código, com e sem átomos de confusão, os participantes relataram que a versão contendo átomos aparece como mais confusa e/ou menos legível em 10 dos 14 átomos pesquisados. Dessa forma, o estudo mostra que esses átomos podem causar confusão entre os desenvolvedores novatos [12].

Em [5], de Oliveira et al. avaliaram as interpretações de código com e sem AC usando um rastreador ocular. De uma perspectiva agregada, foi observado um aumento de 43,02% no tempo e 36,8% nas transições de olhar em trechos de código com AC. Também foi observado que as regiões que receberam mais atenção do olho foram as regiões que continham átomos.

Esses trabalhos relacionados apresentados e suas descobertas reforçam que os átomos de confusão não só atrapalham a compreensão de código, mas também o desempenho dos programadores em suas atividades de desenvolvimento. Portanto, os desenvolvedores devem evitar escrever código com eles. Ressalta-se que na revisão da literatura realizada durante o desenvolvimento deste trabalho, não foi encontrada nenhuma ferramenta capaz de realizar a busca e a identificação automática de AC em códigos Java.

3 A FERRAMENTA BOHR

A ferramenta proposta neste trabalho é denominada BOHR - *The Atoms of Confusion Hunter* que tem como objetivo obter a prevalência de AC em sistemas Java, varrendo e analisando os arquivos *.java* contidos em repositórios de software em busca de AC presentes em seu código-fonte. Nesta primeira versão, o BOHR é capaz de detectar 8 dos 13 tipos de AC apontados por Langhout e Aniche [12] que são os seguintes: *Post-Increment/Decrement, Pre-Increment/Decrement, Conditional Operator, Arithmetic as Logic, Logic as Control Flow, Change of Literal Encoding, Omitted Curly Braces* e *Type Conversion*. Em relação à cobertura do AC Type Conversion, atualmente ele contempla apenas os casos apresentados no trabalho [11], isto é, conversões entre tipos primitivos de *float* para *int* e de *int* para *byte*. Devido às suas complexidades, as implementações das detecções dos AC *Infix Operator Precedence* e *Repurposed Variables* também ainda não são contempladas, pretende-se contemplar a identificação desses AC em uma próxima versão da ferramenta.

Os AC *Remove Indentation, Indentation, Dead, Unreachable, Repeated,* também apontados como confusos em [12], não são cobertos pelo BOHR. Como exposto em [11], entende-se que esses átomos podem ser mais facilmente evitados. Por exemplo, os AC *Remove Indentation* e *Indentation* podem ser evitados com a utilização de formatadores de códigos automáticos presentes na maioria dos editores de código. Enquanto os AC *Dead, Unreachable, Repeated* são detectáveis por inspetores de códigos estáticos presentes em IDE's, que indicam mensagens de advertência informando sobre este comportamento indesejado. Portanto, decidiu-se não contemplar a detecção desses átomos de confusão nesta primeira versão do buscador.

Tabela 1: Átomos de Confusão em Java adaptados de [12]

Nome do Átomo	Snippet com Átomo de Confusão	Snippet sem Átomo de Confusão
Post-Increment/Decrement	`a = b++;`	`a = b;` `b += 1;`
Pre-Increment/Decrement	`a = ++b;`	`b += 1;` `a = b;`
Conditional Operator	`b = a == 3 ? 2 : 1;`	`if(a == 3){b = 2;}` `else{b = 1;}`
Arithmetic as Logic	`(a - 3) * (b - 4) != 0`	`a != 3 && b != 4`
Logic as Control Flow	`a == ++a > 0 || ++b > 0`	`if(!(a + 1 > 0)) {b += 1;}` `a += 1`
Change of Literal Encoding	`a = 013;`	`a = Integer.parseInt("13", 8);`
Omitted Curly Braces	`if(a) f1(); f2();`	`if(a){ f1(); } f2();`
Type Conversion	`a = (int) 1.99f;`	`a = (int) Math.floor(1.99f);`

3.1 Visão Geral

O BOHR realiza a inspeção de sistemas Java a partir do caminho do diretório dos arquivos do repositório do código. A partir desse caminho informado, processadores responsáveis pela detecção de átomos específicos, chamados de *Finders*, atuam nas buscas por átomos de confusão. Essa ferramenta permite ao analista selecionar os *Finders* desejados a cada execução da ferramenta. As informações dos átomos encontrados são armazenadas em um *Dataset*, podendo serem exportadas em um relatório em forma de arquivo CSV.

O arquivo CSV apresenta os trechos de código que contêm átomos de confusão, os tipos deles, os nomes das classes e as linhas em que os átomos foram encontrados. A utilização do tipo de arquivo CSV possibilita a exploração dos dados por vários tipos de softwares compatíveis com esse formato (softwares geradores de planilhas, APIs de software, sistemas de bancos de dados, softwares de Business Intelligence etc.). Assim, pesquisadores podem manipular os dados gerados da forma que desejarem. A partir desse relatório, é possível, por exemplo, verificar a prevalência de átomos de confusão e extrair dados que podem substanciar diferentes análises de impactos causados pela presença deles.

Como a área de estudo de átomos de confusão ainda é recente, entendeu-se que novos padrões de códigos confusos podem surgir, por isso, o BOHR também inclui uma API que permite ao pesquisador implementar seu próprio *Finder* para a detecção de novos padrões de código confusos. A Figura 1 apresenta um visão geral de funcionamento da ferramenta.

O desenvolvimento do BOHR foi feito utilizando o Spoon [16], uma biblioteca de código aberto que oferece uma API para analisar, reescrever, transformar e transpilar o código-fonte Java. O Spoon analisa os arquivos .java, construindo uma árvore sintática abstrata (AST - *Abstract Syntax Tree*) para análise e manipulação de código-fonte Java. Os testes unitários para validação do BOHR foram implementados utilizando o *framework open-source* JUnit [20].

A Listagem 1 apresenta a implementação de um *Finder* responsável por identificar o AC *Conditional Operator*. O método *process*

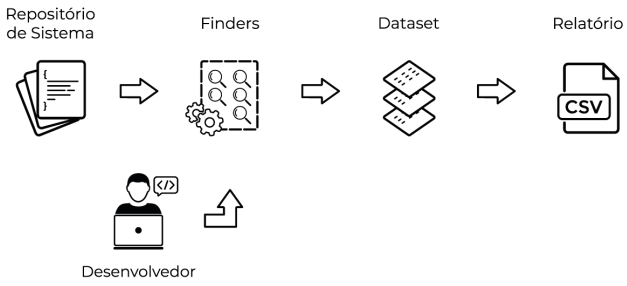

Figura 1: Visão geral de funcionamento do BOHR

recebe uma Classe Java como parâmetro, onde primeiro verifica se este elemento é válido, se temos acesso ao ao código-fonte dela por meio de seu arquivo .java, em seguida um filtro é definido para a captura de expressões condicionais usando o operador "?"(expressões ternárias). Por fim, o laço *for* percorre e armazena em um *dataset* todas as expressões condicionais ternárias contidas na Classe.

Listing 1: Exemplo de Processador

```
1  public class ConditionalOperatorFinder extends AbstractProcessor<CtClass<?>> {
2
3    public void process(CtClass<?> element) {
4      if (Util.isValid(element)) {
5            String className = element.getQualifiedName();
6
7        TypeFilter<CtConditional<?>> filter = new TypeFilter<CtConditional<?>>(
                 CtConditional.class);
8        for (CtConditional<?> condOpr : element.getElements(filter)) {
9          if ((condOpr.getParent() != null)
10             && ((condOpr.getParent() instanceof CtAssignment)
11             || condOpr.getParent() instanceof CtLocalVariable)) {
12           int lineNumber = condOpr.getParent().getPosition().getLine();
13           String snippet = condOpr.getParent().prettyprint();
14           Dataset.store(className, new AoCInfo(AoC.CoO, lineNumber, snippet));
15         }
16       }
17     }
18   }
19 }
```

O programador pode criar seu próprio *Finder* extendendo a classe abstrata `AbstractProcessor` do Spoon [16] e implementando o seu método process. Depois disso, para utilizar o novo *Finder* implementado, utiliza-se o método *findAoC* da classe *BohrAPI*. Esse método pode receber como parâmetro uma lista de *Strings*, onde cada item dessa lista corresponde a uma *String* que especifica o nome qualificado da classe que implementa o *Finder* que se deseja utilizar.

4 AVALIAÇÃO INICIAL

Com o intuito de avaliar os algoritmos implementados para detectar a presença de AC, optou-se por uma primeira avaliação de performance com foco na medição da precisão dos algoritmos de detecção. Nesse sentido, testes unitários foram desenvolvidos e, além disso, o BOHR foi executado em projetos Java de código aberto. Dessa forma, esta avaliação consistiu na realização das atividades descritas nas próximas subseções.

4.1 Validação por Testes Unitários

Foram utilizados os exemplos de AC apresentados no trabalho de Langhout [11] para a formação do *Dataset* para esta validação. A partir desses *snippets* obtidos, foram criados testes unitários automatizados, utilizando JUnit, um teste para cada tipo de AC contemplado. Dessa forma, cada AC teve seu teste unitário correspondente. Além disso, foi criado uma grande classe de átomos compilados, com 590 linhas, 59 métodos e contendo 44 ocorrências de AC, onde foram adicionadas mais variações dos átomos apresentados em [11]. O intuito foi verificar com mais exatidão a performance do BOHR na detecção de variações dos AC.

4.2 Seleção de Sistemas

A busca por sistemas foi realizada em repositórios do GitHub. A seleção dos sistemas seguiu os seguintes critérios: (i) ser desenvolvido em Java; (ii) ter seu código aberto; (iii) ser relevante; e (iv) ser de tamanho pequeno. A relevância do sistema foi avaliada levando em consideração sua popularidade na comunidade de desenvolvedores, medida por meio do números de estrelas no GitHub e da quantidade de *forks* realizados [15]. Já o critério (iv) foi aplicado a fim de viabilizar a conferência manual das ocorrências de AC apontados pelo BOHR. Nesse sentido, observando a classificação de Pinto et al. [17], foram escolhidos sistemas considerados pequenos, contendo até 20.000 linhas de código (*LoC - Lines of Code*).

Dessa forma, foram selecionados três sistemas reais, de código aberto, largamente adotados pela comunidade de desenvolvimento de software Java em suas respectivas versões de *releases*: Picasso (V2.8[1]), greenDAO (V3.3.0[2]) e Socket.IO-client Java (V2.0.1[3])

4.3 Validação da Precisão

Consistiu na execução do BOHR nos arquivos de código-fonte dos sistemas selecionados, excluindo os códigos de testes. Nesta etapa foi verificada a precisão dessa ferramenta da seguinte forma: foi executada a busca por AC nos códigos e verificado, manualmente, se cada ocorrência de AC apontada pelo BOHR realmente consistia no AC indicado, ou seja, se o tipo, *snippet*, linha e classe de ocorrência do AC foram informados corretamente por ele. Dessa forma, a precisão verificada de todos os AC encontrados pelo BOHR nos três projetos analisados foi de 100%, todas as informações dos AC encontrados estavam corretas.

[1]https://github.com/square/picasso/releases/tag/2.8
[2]https://github.com/greenrobot/greenDAO/releases/tag/V3.3.0
[3]https://github.com/socketio/socket.io-client-java/releases/tag/socket.io-client-2.0.1

Tabela 2: Resultados da busca por Átomos de Confusão

Tipo de átomo de confusão	Picasso	greenDAO	Socket. IO-client
Post-Increment/Decrement	-	-	2
Pre-Increment/Decrement	-	1	-
Conditional Operator	18	16	10
Arithmetic as Logic	-	-	-
Logic as Control Flow	8	8	9
Change of Literal Encoding	-	-	-
Omitted Curly Braces	10	-	22
Type Conversion	1	-	-

4.4 Resultados

O BOHR foi capaz de encontrar AC presentes nos sistemas analisados. Ele apontou, de forma correta, os *snippets* dos AC, assim como o seu tipo, nome da classe que era pertencente e o número da linha de sua ocorrência. No total, foram encontrados 105 AC, sendo 37 no Picasso, 25 no greenDAO e 43 no Socket.IO-client Java.

Os ACs *Conditional Operator, Logic as Control Flow* e *Omitted Curly Braces* se mostraram comuns (i.e., mais prevalentes) nos sistemas analisados, enquanto os ACs *Post-Increment/Decrement, Pre-Increment/Decrement* e *Type Conversion* apareceram pontualmente nesses mesmos sistemas. Os ACs *Arithmetic as Logic* e *Change of Literal Encoding* não foram encontrados nos sistemas estudados. A Tabela 2 apresenta os resultados desta avaliação, mostrando os tipos de AC encontrados em cada sistema e os números de suas ocorrências. Os relatórios CSV gerados pelo BOHR nesta avaliação se encontram disponíveis em: https://github.com/wendellmfm/bohr-aoc-api/tree/master/atomsreports.

5 CONSIDERAÇÕES FINAIS

Neste trabalho, é descrita a ferramenta BOHR que detecta Átomos de Confusão em sistemas Java. Também é fornecida uma API para que pesquisadores interessados possam criar seus próprios detectores de padrões de código confusos. As avaliações iniciais em três repositórios demonstraram a capacidade da ferramenta de identificar com precisão a presença dos AC por meio dos *Finders* implementados. Entretanto, uma limitação da pesquisa foi a não medição da revocação da ferramenta BOHR. Isso em função da inexistência de um *dataset* com AC devidamente anotados. Essa tarefa está em andamento e deve ser finalizada em uma próxima versão da ferramenta.

Nesse sentido, em trabalhos futuros pretende-se a realização de avaliação do BOHR seguindo uma abordagem com foco na medição de precisão e revocação. Isso por meio da criação de um *dataset* formado por códigos Java com todos os seus AC previamente anotados e mapeados, estabelecendo um padrão *"Gold Standard"* de avaliação. Além disso, intenta-se a aplicação do BOHR em uma

grande variedade de sistemas Java de código aberto, com o objetivo de obter dados de prevalência e evolução de AC. Dessa forma, pretende-se a investigação dos impactos que AC podem causar sobre diversos aspectos do desenvolvimento de software. Também, em trabalho futuro, deseja-se expandir a cobertura dessa ferramenta para contemplar a detecção de todos os casos do AC *Type Conversion* possíveis em Java, além da implementação da detecção dos AC *Infix Operator Precedence* e *Repurposed Variables*. O BOHR está sob a licença GPL *(General Public License)* e seu código-fonte é público, disponível em: https://github.com/wendellmfm/bohr-aoc-api.

REFERÊNCIAS

[1] Shulamyt Ajami, Yonatan Woodbridge, and Dror G Feitelson. 2019. Syntax, predicates, idioms—what really affects code complexity? *Empirical Software Engineering* 24, 1 (2019), 287–328.

[2] K.H. Bennett, V.T. Rajlich, and N. Wilde. 2002. Software Evolution and the Staged Model of the Software Lifecycle. Advances in Computers, Vol. 56. Elsevier, 1–54. https://doi.org/10.1016/S0065-2458(02)80003-1

[3] Pierre Carbonnelle. 2021. *PYPL PopularitY of Programming Language.* https://pypl.github.io/PYPL.html

[4] Fernando Castor. 2018. Identifying confusing code in Swift programs. In *Proceedings of the VI CBSoft Workshop on Visualization, Evolution, and Maintenance. ACM.*

[5] Benedito de Oliveira, Márcio Ribeiro, José Aldo Silva da Costa, Rohit Gheyi, Guilherme Amaral, Rafael de Mello, Anderson Oliveira, Alessandro Garcia, Rodrigo Bonifácio, and Baldoino Fonseca. 2020. Atoms of Confusion: The Eyes Do Not Lie. In *Proceedings of the 34th Brazilian Symposium on Software Engineering* (Natal, Brazil) *(SBES '20).* Association for Computing Machinery, New York, NY, USA, 243–252. https://doi.org/10.1145/3422392.3422437

[6] Rodrigo Magalhães dos Santos and Marco Aurélio Gerosa. 2018. Impacts of Coding Practices on Readability. In *Proceedings of the 26th Conference on Program Comprehension* (Gothenburg, Sweden) *(ICPC '18).* Association for Computing Machinery, New York, NY, USA, 277–285. https://doi.org/10.1145/3196321.3196342

[7] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2017. Confusion Detection in Code Reviews. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME).* 549–553. https://doi.org/10.1109/ICSME.2017.40

[8] Dan Gopstein, Anne-Laure Fayard, Sven Apel, and Justin Cappos. 2020. Thinking Aloud about Confusing Code: A Qualitative Investigation of Program Comprehension and Atoms of Confusion *(ESEC/FSE 2020).* Association for Computing Machinery, New York, NY, USA, 605–616. https://doi.org/10.1145/3368089.3409714

[9] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. 2017. Understanding Misunderstandings in Source Code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017).* Association for Computing Machinery, New York, NY, USA, 129–139. https://doi.org/10.1145/3106237.3106264

[10] Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. 2018. Prevalence of Confusing Code in Software Projects: Atoms of Confusion in the Wild *(MSR '18).* Association for Computing Machinery, New York, NY, USA, 281–291. https://doi.org/10.1145/3196398.3196432

[11] Chris Langhout. 2020. *Investigating the Perception and Effects of Misunderstandings in Java Code.* Master's thesis. Delft University of Technology.

[12] Chris Langhout and Maurício Aniche. 2021. Atoms of Confusion in Java. arXiv:2103.05424 [cs.SE]

[13] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. In *2015 IEEE 23rd International Conference on Program Comprehension.* 25–35. https://doi.org/10.1109/ICPC.2015.12

[14] Stephen O'Grady. 2021. *The RedMonk Programming Language Rankings: January 2021.* https://redmonk.com/sogrady/2021/03/01/language-rankings-1-21/

[15] Michail Papamichail, Themistoklis Diamantopoulos, and Andreas Symeonidis. 2016. User-Perceived Source Code Quality Estimation Based on Static Analysis Metrics. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS).* 100–107. https://doi.org/10.1109/QRS.2016.22

[16] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179. https://doi.org/10.1002/spe.2346

[17] Gustavo Pinto, Weslley Torres, Benito Fernandes, Fernando Castor, and Roberto S.M. Barros. 2015. A large-scale study on the usage of Java's concurrent programming constructs. *Journal of Systems and Software* 106 (2015), 59–81. https://doi.org/10.1016/j.jss.2015.04.064

[18] Akond Rahman. 2018. Comprehension Effort and Programming Activities: Related? Or Not Related?. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18).* Association for Computing Machinery, New York, NY, USA, 66–69. https://doi.org/10.1145/3196398.3196470

[19] Spencer Rugaber. 1995. Program comprehension. *Encyclopedia of Computer Science and Technology* 35, 20 (1995), 341–368.

[20] The JUnit Team. 2021. *JUnit 5.* https://junit.org/junit5/

[21] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering* 44, 10 (2018), 951–976. https://doi.org/10.1109/TSE.2017.2734091