Identifying Confusing Code in Swift Programs

Fernando Castor[1]

[1]Informatics Center – Federal University of Pernambuco (UFPE)
Recife – PE – Brazil

castor@cin.ufpe.br

Abstract. *Atoms of confusion are small code patterns that are likely to cause confusion in developers and can be replaced by functionally equivalent alternatives that are less likely to confuse. The only existing catalog of atoms of confusion targets the C language. Our long term goal is to devise a similar catalog for the Swift programming language. However, this is not straightforward because it requires specialist knowledge about the programming language under study. Also, there is currently no direct definition of atom of confusion that makes it possible to unequivocally differentiate atoms from related concepts and there are no principles or guidelines for identifying new atoms. This paper presents preliminary work towards our long term goal and makes three contributions: (i) provides a direct, structured definition of atoms of confusion; (ii) examines factors that cause existing atoms to be confusing; and (iii) presents a preliminary catalog of atoms of confusion for Swift.*

1. Introduction

Understanding code is an important activity in the development of software systems. Code that is difficult to understand can negatively impact various development-related activities, such as testing, code review [2, 6], and editing and navigation [16]. For example, it is one of the factors that negatively affects the performance of reviewers during code reviews, causing confusion [6] and slowing down the process [2]. One of the obstacles faced by developers performing the activity of understanding the code of software systems is their complexity. Complexity can create a conflict between what a developer trying to understand the functioning of a system understands and the actual behavior of that system, i.e., how a computer would understand it. Fred Brooks [3] stated that there are two types of complexity in software systems: essential and accidental. In the first case, the problem that the program solves is inherently difficult and its solution is therefore complex. In the second one, the complexity stems from decisions made by developers and is not inherent. Accidental complexity can manifest itself in a variety of ways: code smells [7], anti-patterns [4], identifiers that either are too short or do not clearly express the intent of developers [1, 11], among others. Recent work has identified another form of accidental complexity, the **atoms of confusion** [8]. According to Gopstein et al. [8], atoms of confusion are

> *"[...] the smallest pieces of code that can routinely cause programmers to misunderstand code. [...] These atoms can serve as an empirical and quantitative foundation for understanding what makes code confusing."*

Two experiments, one with 73 participants and the other with 43, found that small C programs including atoms of confusion are more difficult to understand than functionally equivalent programs that do not include these atoms [8]. In addition, a recent study [9] involving 14 large-scale open-source projects written in the C language has revealed that there are plenty of atoms in real and successful projects (such as Git and the Linux kernel). The presence of these atoms in a project has a strong correlation with bug fixing commits and long code comments.

In this position paper, we present preliminary work on the study of atoms of confusion in the Swift programming language.[1] According to Apple[2], Swift is a *"powerful language that is also easy to learn"*, *"safe by design"*, and *"intuitive"*. In fact, Swift avoids by construction some of the atoms of confusion presented by Gopstein et al. [8], such as omitted curly braces and assignment expressions. This motivates us to ask the question: **What are the atoms of confusion of the Swift programming language?** Identifying new atoms for a programming language is non-trivial, however. Firstly, because it requires specialist knowledge about the programming language under study. Secondly, because there is currently no direct definition of atom of confusion that makes it possible to unequivocally differentiate atoms from related concepts, such as code smells. Thirdly, because there are no principles or guidelines for identifying new atoms. In this paper, we address the latter two issues and present a preliminary list atom candidates for Swift. Furthermore, we discuss a number of future avenues for research in this area.

2. Atoms of Confusion

This section starts by providing a structured definition of atom of confusion (Section 2.1). It then proceeds by showing that this definition makes it possible to clearly distinguish between atoms of confusion and two related concepts, code smells and antipatterns (Section 2.2).

2.1. A definition

In their original work, Gopstein et al. [8] present a set of 15 atoms they extracted from winning programs of the International Obfuscated C Code Contest[3]. As the authors themselves acknowledge, they target *"[...]specific situations where a programmer might tend to misunderstand the behavior of a piece of code."*. Furthermore, they state that *" We restricted our definition of an atom to only minimal portions of code so that our findings would be generalizable and occur frequently in real projects."*. Table 1 presents some examples of atoms they have identified, together with functionally equivalent alternative code patterns that do not include them. A complete list is available elsewhere [8, 9].

The creators of the concept of atom of confusion do a good job of providing an intuition about what it means for a code pattern to be an atom. Nonetheless, the concept of atom is not defined in a direct manner, clearly specifying the necessary conditions for a code pattern to be considered an atom. To assist researchers interested in the study of atoms of confusion, we believe it is useful to have a direct, structured definition. Based on the information available in the aforementioned two papers, we define an atom of confusion as a code pattern that is

- precisely identifiable,
- likely to cause confusion,

[1] https://docs.swift.org/swift-book/GuidedTour/GuidedTour.html
[2] https://developer.apple.com/swift/
[3] https://www.ioccc.org

Atom Name	Atom Example	Transformed
Literal Encoding	`printf("%d", 013)`	`printf("%d", 11)`
Assignment as Value	`V1 = V2 = 3;`	`V2=3;` `V1 = V2;`
Post- and Pre-Increment	`V1 = V2++;`	`V1 = V2;` `V2 += 1;`
Conditional Operator	`V2 = (V1==3)?2:V2`	`if (V1==3) { V2 = 2; }`
Omitted Curly Brace	`if (V) F(); G();`	`if (V) { F(); } G();`
Repurposed Variable	`argc = 7;`	`int V1=7;`
Implicit Predicate	`if(4%2)`	`if(4%2 != 0)`
Logic as Control Flow	`V1 && F2();`	`if (V1) F2();`
Comma Operator	`V3 = (V1 += 1, V1);`	`V1 += 1; V3 = V1;`

Table 1. Some atoms of confusion for the C language and functionally equivalent, less confusing transformed versions. Adapted from [9]

.

- replaceable by a functionally equivalent code pattern that is less likely to cause confusion, and
- indivisible.

In the definition above, "code pattern" may be a specific language construct, e.g., the Conditional Operator or Assignment as Value, or a usage pattern, e.g., Repurposed Variable or Implicit Predicate. By "precisely identifiable", we mean that the atom can be identified independently of subjective opinions. For example, it may be the case that the `F2()` function call is in fact part of the condition in the atom example of Logic as Control Flow in Table 1. In this scenario, the transformed code pattern would not be appropriate. Nonetheless, either the call is part of the condition or it is not. This is different, for example, from a long method code smell [7], where the definition of what constitutes "long" is dependent on personal opinion [13]. Furthermore, "precisely identifiable" is not the same as "mechanically identifiable". For example, because of cases such as the aforementioned example, it is not possible to build a program that detects instances of Logic as Control Flow with precision.

The "likely to cause confusion" part of the definition is implied by a basic premise: that an empirical evaluation has been conducted showing that the atom candidate is in fact confusing. The paper introducing the concept of atoms [8] describes an experiment where the performance of developers in understanding small programs including atom candidates was measured in terms of how often they correctly identified the output of these programs and how long it took them to do so. Moreover, this part of the definition is strongly connected to the third one ("can be replaced..."). Gopstein et al. [8] state that *"We define atoms relative to functionally equivalent code that does not confuse programmers."* For example, one of the atom candidates of the aforementioned paper is Pointer Arithmetic. Although this candidate is verifiably confusing, it is not considered an atom of confusion because the developers participating in the experiment were as confused by the evaluated alternative code patterns as they were by the atom candidate. The existence of a less confusing, functionally equivalent alternative code pattern where the atom is not present is a necessary condition for a code pattern to be considered an atom.

Finally, an atom is "indivisible" in the sense that it cannot be simplified and remain an atom nor be broken down into smaller atoms. This part of the definition emphasizes that an atom represents the simplest possible example of a potential cause for confusion.

2.2. Atoms, code smells, antipatterns

Atoms of confusion are similar to the well-known code smells [7] which have been studied extensively (For example, [12, 14, 17]). According to Fowler [7], code smells are symptoms of deeper problems that may occur in software systems. Atoms differ from code smells at the conceptual and practical level. At the conceptual level, atoms of confusion are hints of readability problems. On the other hand, code smells may indicate problems related to a wide range of quality attributes, e.g., maintainability [7], performance [10], and energy-efficiency [18]. Furthermore, atoms of confusion exist at a fine level of granularity, in the context of a definition, statement, or expression. Code smells can involve several classes and high level design issues. At the practical level, atoms of confusion can be precisely identified. Code smells, on the other hand, often depend on subjective judgement, which may lead to conflicting understandings about their presence or absence [13]. Furthermore, the requirements of empirical validation and indivisibility are not part of the definition of code smell. The existence of a less confusing, functionally equivalent alternative solution, however, is implied because code smells were introduced as a motivation for refactoring [7].

Atoms and smells are sometimes directly related. For example, one of the code smells from Fowler's book [7] is Switch Statements. Since `switch` statements are trivially identifiable, this smell could be easily explained as an atom of confusion, as long as it can be empirically shown that (i) it hinders readability and (ii) there is a functionally equivalent alternative that is less confusing. As an additional example, the presence of Comments is another smell from the book. As pointed out by Gopstein et al. [9], code that includes atoms usually exhibits longer comments than code that does not.

Antipatterns [4] are bad solutions to recurring design problems. They have a negative impact on the quality attributes of a system. According to Brown et al. [4], antipatterns manifest mainly at the development, architecture, and process levels. This definition emphasizes the broad scope of antipatterns when compared to atoms of confusion. For example, the is no correspondence between process level antipatterns and atoms. As shown by Moha and colleagues [14], development and architecture level antipatterns and code smells have a direct relationship; code smells are concretizations of these antipatterns. Since the latter are very abstract, even when referring directly to code elements, none of the elements of the definition presented in Section 2.1 apply to antipatterns in general. Antipatterns sometimes negatively impact readability, e.g., Spaghetti Code may include long methods and pervasive use of global variables. Notwithstanding, they may impact other quality attributes of software development processes and products.

3. Sources of atoms of confusion

To devise an atom catalog for C programs, Gopstein et al. [8] relied on purposely complex code examples that appeared in the International Obfuscated C Code Contest. To win this competition, developers need to build programs that are difficult to understand, using potentially esoteric C-language features. As the name implies, IOCCC is unique to C programs and there are no similar competitions for other languages.[4] Thus, this methodology cannot be used to devise new atom catalogs targeting different programming languages.

[4]There used to be an analog competition for Perl programs, but the last edition occurred in 2000.

The development of a catalog of atoms of confusion requires expert knowledge about the language and also about the typical **sources of confusion**, the underlying reasons that make a code pattern difficult to understand. Although it is not possible to leverage IOCCC programs to devise atoms for other languages in a direct manner, we hypothesize that the sources of confusion for the identified set of atoms can be leveraged to devise new atom catalogs. Taking that hypothesis as a starting point, we performed reverse engineering of the atom catalog of Gopstein and colleagues [8]. We classified the atoms based on the sources of confusion that each one exhibits, as discussed by the authors themselves (Table 1 of [8]). The resulting classification can then serve as a basis for the identification of atoms in other languages. Since this process relies on subjective judgement, we employed open card sorting [19], assigning one atom description to each card. We then proceeded to grouping the cards, sometimes combining simple groups, sometimes splitting complex ones. Since our goal in eliciting sources of confusion is to provide guidance to designers of new atom catalogs and not define a rigid categorization, we admitted the possibility of a card belonging to more than one group. The card sorting process yielded 5 sources of confusion, presented in Table 2. We use these sources of confusion as a starting point to devise a catalog of atom candidates for Swift (Section 4).

4. Atom of Confusion Candidates for Swift

We devised a preliminary set of atom candidates for Swift. To identify the atom candidates, we conducted the following steps, guided by the atom catalog of Gopstein and colleagues [8] and the sources of confusion of Table 2:

Existing catalog. We started out by analyzing each atom in the original catalog and excluded the ones that cannot happen in Swift due to its differences from C. This step excluded 9 of the original 15 atoms. For example, Assignment as Value, Omitted Curly Braces, and Pre- and Post-Increments do not exist in Swift. The remaining 6 atoms are all atom candidates for Swift.

Uncommon Constructs. We then proceeded by adapting the Swift-AST[5] Swift parser to collect information about the frequency of use of the syntactic constructs from 5 large Swift projects that were analyzed in a recent study [5]. For each project, we selected the 5 least popular ones and checked the feasibility of implementing functionally equivalent code patterns that do not include these constructs. This process produced one additional atom candidate, Defer Statement, and showed that usage of Conditional Expression is uncommon in these Swift projects.

Usage beyond Lexical Meaning and Syntactically Similar, Semantically Different Constructs. Based on expert knowledge[6] and language documentation, we identified cases where there are different ways of using a construct and also cases where the same symbol or identifier is employed with different semantics, depending on the context of use. There are many instances of these cases in Swift. This step yielded 7 atom candidates.

Obscuring programming practices. This is the most pragmatics-dependent source of

[5]https://github.com/yanagiba/swift-ast

[6]The authors have been working with, teaching, and researching on Swift programming since January 2015, 7 months after the first release of the language.

Source of Confusion	**Description**	**Affected Atoms**
Uncommon constructs	The use of little-known language constructs. This is a source of confusion for many of the atoms of confusion.	Literal Encoding Comma Operator Conditional Operator
Construct usage beyond lexical meaning	Some constructs are typically used in a certain way. However, additional, less common usage patterns are also possible and tend to obscure program meaning.	Logic as Control Flow Assignment as Value
Syntactically similar, semantically different constructs	The semantics of a symbol or identifier changes depending on the context of use.	Post- and Pre-Increment
Complex rules	Ambiguity and lack of clarity because language rules are non-obvious and not explicit.	Operator precedence
Obscuring programming practices	Usage patterns of common programming language constructs that obscure the meaning of the program.	Repurposed Variables Omitted Curly Braces Implicit Predicate

Table 2. Sources of confusion. Each one represents a potential reason for a code pattern to be confusing.

confusion. In this work, we have identified one example of programming practice that is commonplace but obscures readability, based on previous work on visualizing Swift code [15].

We also identified one additional atom candidate that is a rarely-used Swift-specific derivation of Literal Encoding. Furthermore, we could not find more atom candidates besides Operator Precedence when considering the Complex Rules source of confusion. Table 3 presents a subset of the atom candidates we identified, together with the functionally equivalent alternatives. We omit candidates that have a direct correspondence to previously identified atoms [8], except for Literal Encoding because Swift has additional options and slightly different conventions for expressing number encoding. In Table 3, Hexadecimals with Exponents is a special case of number encoding where a floating point hexadecimal is written as a product of an hexadecimal integer and a power of 2. The `defer` statement is rarely used in the Swift projects we analyzed. The code block associated with a `defer` statement is executed right before jumping out of the enclosing block. If multiple `defer` statements are executed within the same scope, they create a nested structure where the code block of the last `defer` is the first one to be executed.

Swift supports higher-order functions and two atom candidates are related to that language feature. Shorthand Arguments refers to the use of implicit parameter names when employing closures. In the example, `$0` is the name of an implicitly defined parameter of the closure `{$0 > 11}(V1)`, which takes a number as argument and checks whether it is greater than 11. In the transformed version. the parameter `V` is explicitly named. Trailing Closures are special syntax for cases where the last parameter of a function is a closure. In the example, the use of the `reduce` function will sum all the elements of array `V1`, since it takes an initial value and a two parameter function and combines that initial value with all the elements of the array using the given function.

Swift programs make frequent use of optional types. A variable of an optional type is one that potentially holds `nil`. For example, a variable of type optional integer (`Int?`) either stores a integer number (`Int`) or stores `nil`, but a variable of type `Int` is guaranteed to store an integer. If a variable of an optional type actually stores an integer, obtaining this integer requires a process known as unwrapping. Swift makes a number of unwrapping approaches available for developers. One of the atom candidates

Atom Candidate Name	Example	Transformed
Literal Encoding	`V1 = 0x12 | 0x6`	`V1 = 0b10010 | 0b00110`
Hexadecimals with Exponents	`V1 = 0x50p-2`	`V1 = 0x50 / 4`
Defer Statement	`defer { V1 += 1 }` `V1 = 3`	`V1 = 3` `V1 += 1`
Shorthand Arguments	`V1 = 9` `{$0 > 11}(V1)`	`V1 = 9` `{V in return V > 11}(V1)`
Trailing Closures	`V1=[5,7,8,10,3]` `V3=V1.reduce(0) {V2,V4 in V2+V4}`	`V1=[5,7,8,10,3]` `V3=V1.reduce(0, {V2,V4 in V2+V4})`
Overloaded “!”	`V1 = readLine()` `if V1! != "OK" {` `print("Not ok")` `}`	`V1 = readLine()!` `if V1 != "OK" {` `print("Not ok")` `}`

Table 3. Swift atom candidates.

of Table 3 refers to unwrapping operators. In the example of Overloaded “!”, function `readLine()` returns a value of type `String?`. The following line unwraps the values stored in `V1` using the forced unwrapping operator `(!)` and then checks whether the wrapped string is different `(!=)` from `"OK"`. The problem is that the two semantically different, textually close uses of the “!” symbol in this example may be surprising for developers and cause them to assume that the program has a bug.

5. Concluding Remarks

In this paper, we proposed a definition for atoms of confusion, identified some sources of confusion that exist in the original atoms catalog, and presented a preliminary atom catalog for Swift. Atoms of confusion are an exciting topic of research for which many avenues remain open. The first and most obvious one is to identify new atom candidates, including atoms for other languages, and empirically evaluate their proneness to cause confusion. In fact, this is the next step we intend to take with the preliminary set of candidates presented here. Furthermore, deriving a theory to explain why certain code patterns are harder to read than others is a worthy goal. In this paper, we have given a first step in that direction by introducing the notion of sources of confusion.

We also think that attempting to understand how atoms impact real world development activities is important. Recent work [9] has shown that atoms are often related to bugs. We believe that attempting to understand how atoms impact code review and testing activities may help developers become more productive and the code produced by them more readable. In addition, although atoms of confusion are simple code patterns whose impact should be measurable in isolation, studying the ways in which these atoms are typically combined can pave the way for the development of new analysis and reengineering solutions.

Acknowledgements. We’d like to thank Felipe Ebert and Dan Gopstein, who read preliminary versions of this paper. This research was partially funded by INES 2.0, (FACEPE PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17, and CNPq 465614/2014-0), FACEPE APQ-0839-1.03/14, and CNPq 304220/2017-5.

References

[1] Eran Avidan and Dror G. Feitelson. Effects of variable names on comprehension: an empirical study. In *Proceedings of the 25th ICPC*, pages 55–65, Buenos Aires, Argentina, May 2017.

[2] Amiangshu Bosu, Jeffrey C. Carver, Christian Bird, Jonathan D. Orbeck, and Christopher Chockley. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. *IEEE Trans. Software Eng.*, 43(1):56–75, 2017.

[3] Frederick P. Brooks Jr. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[4] William J. Brown, Raphael C. Malveau, Hays W. McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1st edition, 1998.

[5] Nathan Cassee, Gustavo Pinto, Fernando Castor, and Alexander Serebrenik. How swift developers handle errors. In *Proceedings of the 15th MSR*, Gothenburg, Sweden, May 2018.

[6] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. Confusion detection in code reviews. In *Proceedings of the 33rd ICSME*, pages 549–553, Shanghai, China, September 2017.

[7] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. Understanding misunderstandings in source code. In *Proceedings of the 11th ESEC/FSE*, pages 129–139, Paderborn, Germany, September 2017.

[9] Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. Prevalence of confusing code in software projects. In *Proceedings of the 15th MSR*, Gothenburg, Sweden, May 2018.

[10] G. Hecht, N. Moha, and R. Rouvoy. An empirical study of the performance impacts of android code smells. In *Proceedings of the 3rd MOBILESoft*, pages 59–69, Austin, USA, May 2016.

[11] Dawn J. Lawrie, Christopher Morrell, Henry Feild, and David W. Binkley. What's in a name? A study of identifiers. In *Proceedings of the 14th ICPC*, pages 3–12, Athens, Greece, June 2006.

[12] Isela Macia, Joshua Garcia, Daniel Popescu, Alessandro Garcia, Nenad Medvidovic, and Arndt von Staa. Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems. In *Proceedings of the 11th AOSD*, pages 167–178, Potsdam, Germany, 2012.

[13] Mika Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11(3):395–431, 2006.

[14] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Trans. Software Eng.*, 36(1):20–36, 2010.

[15] Rafael Nunes, Marcel Rebouças, Francisco Soares-Neto, and Fernando Castor. Visualizing swift projects as cities. In *Companion to the Proceedings of the 39th ICSE*, pages 368–370, Buenos Aires, Argentina, May 2017.

[16] Akond Rahman. Comprehension effort and programming activities: Related? or not related? In *Proceedings of the 15th MSR*, Gothenburg, Sweden, May 2018.

[17] Jan Schumacher, Nico Zazworka, Forrest Shull, Carolyn Seaman, and Michele Shaw. Building empirical support for automated code smell detection. In *Proceedings of ESEM 2010*, pages 8:1–8:10, Bolzano-Bozen, Italy, 2010.

[18] Roberto Verdecchia, René Aparicio Saez, Giuseppe Procaccianti, and Patricia Lago. Empirical evaluation of the energy impact of refactoring code smells. In *5th ICT4S*, pages 365–383, Toronto, Canada, May 2018.

[19] T. Zimmermann. Card-sorting: From text to themes. In Tim Menzies, Laurie Williams, and Thomas Zimmermann, editors, *Perspectives on Data Science for Software Engineering*, pages 137–141. Morgan Kaufmann, 2016.