Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Átomos de Confusão em Swift

Ian Mateus Vieira Manor

Trabalho de Graduação

Recife
14 de Dezembro de 2018

Universidade Federal de Pernambuco
Centro de Informática

Ian Mateus Vieira Manor

Átomos de Confusão em Swift

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Prof. Dr. Fernando Castor*

Recife
14 de Dezembro de 2018

Agradecimentos

Ao meu orientador Fernando Castor pela disposição que teve ao longo desse trabalho.

Aos estudantes Pedro Queiroga, Jonatas Clementino e Ricardo Ferreira pela ajuda indispensável nesse projeto.

À minha grande amiga Isa pelas dicas de metodologia e por estar comigo em tantos projetos.

A todos os outros integrantes do Centro de Informática da UFPE, do programa Apple Developer Academy - UFPE e do Voxar Labs pelas oportunidades e experiências de aprendizado que me deram.

Aos meus pais pelo incentivo na minha educação e por serem grandes exemplos na minha vida.

Por fim, à minha namorada Karolina por estar sempre comigo quando eu preciso.

A true thing poorly expressed is a lie
—STEPHEN FRY

Resumo

Estudos recentes em engenharia de software identificaram a existência dos chamados “Átomos de confusão” em código C. Esses são os menores possíveis padrões de código confuso. Ou seja, código cuja interpretação humana pode variar em comparação à máquina, mas que pode ser facilmente transformado em uma forma mais compreensível e fácil de manter. Esse trabalho visa replicar estudos existentes porém com outra linguagem: a linguagem de programação Swift. Candidatos à átomo de confusão previamente identificados são estudados junto com novos, levantados através de um estudo de materiais sobre Swift. Uma pesquisa piloto é realizada para viabilizar uma futura validação empírica dos candidatos.

Abstract

Recent work in the software engineering field identified the existence of so-called “Atoms of Confusion” in C code. These are the smallest possible patterns of easily misinterpretable code. That is, code which human interpratation might vary in comparison to that of the machine, but that can easily be transformed into a more understandable and maintainable alternate version. This work aims to replicate existing research with another language: the Swift programming language. Atom candidates that have been previously identified will be studied together with new ones identified through a study of Swift material. A pilot survey is conducted to make way for future empirical validation of the candidates.

Sumário

1 Introdução **1**
1.1 Motivação 1
1.2 Objetivos 2
1.2.1 Objetivos Específicos 2
1.3 Estrutura do trabalho 2

2 Trabalhos Relacionados e Conceitos **3**
2.1 Átomos de Confusão 3
2.2 Trabalhos relacionados 4
2.2.1 Code Smells e Antipatterns 4
2.2.2 Compreensão de Código 5
2.3 A Linguagem Swift 5

3 Candidatos a Átomos de Confusão em Swift **7**
3.1 Metodologia 7
3.2 Novos candidatos 8
3.2.1 Map com Efeitos Colaterais 8
3.2.2 Operador de Negação em Condicional 9
3.2.3 Casamento de Padrões Avançado 10
3.3 Candidatos Descartados 10
3.4 Lista Canônica 11

4 Estudo Piloto **15**
4.1 Metodologia 15
4.2 Resultados 16
4.2.1 Melhorias no formulário 16
4.2.2 Melhorias nas questões 17
4.2.3 Validações preliminares 18

5 Considerações Finais **19**
5.1 Trabalhos Futuros 19
5.1.1 Catálogos de átomos 20

Lista de Figuras

2.1 Exemplo de átomo de confusão, adaptado de [1]. 4

3.1 Candidato a átomo de confusão Map com Efeitos Colaterais. 9
3.2 Candidato a átomo de confusão Operador de Negação em Condicional. 9
3.3 Candidato a átomo de confusão Pattern Matching Avançado. 10
3.4 Candidato a átomo de confusão descartado Dead Code. 11
3.5 Interrogação com Múltiplos Significados. 12
3.6 Candidato a átomo de confusão descartado For Case Let. 14

Lista de Tabelas

3.1 Fontes de confusão. Cada uma representa um motivo em potencial para um padrão de código ser confuso. Adaptado de [2] 8
3.2 Candidatos a átomo de confusão em Swift. Adaptado de [2] 12

Capítulo 1

Introdução

1.1 Motivação

.

Desenvolvimento de software é uma atividade complexa que envolve uma gama de tarefas e subtarefas. Porém, há estimativas de que entre 50% a 70% do tempo de desenvolvimento é gasto em entendimento do código [3]. Outros estudos mostram que quando um código requer um maior esforço de compreensão, atividades relacionadas como edição de código e navegação ocorrem de forma significativamente mais devagar [4].

Entendimento de código por si só é uma atividade complexa, e desenvolvedores usam diversos recursos para entender o sistema de software. A principal fonte de entendimento é a leitura do código-fonte, ou seja tende a ser o primeiro lugar que o desenvolvedor busca para entender a tarefa que precisa realizar [5].

Um estudo recente analisou códigos vencedores do concurso internacional de código ofuscado em C (IOCCC) e descobriu que pequenos padrões de código em C (que tinham em média menos de duas linhas) poderiam afetar significativamente a compreensão de código [1]. A esses padrões foram dados o nome de átomos de confusão.

Estudos posteriores mineraram repositórios populares em C e C++, encontrando milhões de ocorrências desses átomos de confusão. Os átomos estavam presentes em 4,38% das linhas de código. A essas linhas de código estavam associadas uma taxa maior de bugs e tinham uma chance maior de serem comentadas, mesmo quando não associadas a um bug [6].

Um estudo exploratório sobre átomos de confusão criou uma base para encontrar candidatos em outras linguagens e gerou uma lista de candidatos de átomos de confusão em Swift [2]. Com o resultado desse estudo, notou-se a necessidade de generalizar o conceito de átomos de confusão para outras linguagens de programação, seguindo a metodologia dos estudos originais [1, 6, 2].

1.2 Objetivos

Este trabalho tem como objetivo replicar estudos anteriores na busca de átomos de confusão em Swift. A validação desses candidatos pode conduzir novas pesquisas sobre clareza de código em Swift e outras linguagens.

1.2.1 Objetivos Específicos

- Construir em cima de uma lista existente de candidatos a átomos de confusão em Swift, fazendo uma análise crítica dos candidatos presentes e propondo novos candidatos.
- Conduzir um estudo piloto para determinar mudanças necessárias na lista de candidatos, no formulário e suas questões ou nos métodos de distribuição dos últimos.

1.3 Estrutura do trabalho

O restante deste trabalho é organizado da seguinte forma: O Capítulo 2 apresenta a definição e contextualização de átomos de confusão. O Capítulo 3 introduz a linguagem Swift e os candidatos propostos nela; O Capítulo 4 descreve o estudo piloto realizado através da sua metodologia e os resultados obtidos; O Capítulo 5 mostra as conclusões e trabalhos futuros; O Apêndice A contém alguns detalhes do formulário usado no estudo piloto; O Apêndice B expõe alguns dos candidatos que foram descartados durante o levantamento de átomos de confusão.

Capítulo 2

Trabalhos Relacionados e Conceitos

2.1 Átomos de Confusão

Fred Brooks categoriza complexidade de software como essencial ou acidental. As dificuldades essenciais são inerentes aos sistemas de software e portanto inevitáveis. Enquanto isso, a complexidade acidental ocorre por causa de limitações tecnológicas ou decisões humanas [7]. Um exemplo de complexidade acidental recém-descoberto são os átomos de confusão [1, 6, 2].

Átomos de confusão são pequenos padrões de código que tendem a causar confusão em desenvolvedores. Confusão aqui significa que ao ler um trecho de código, o desenvolvedor chegará a uma conclusão errônea sobre a execução do código [1]. Castor dá uma definição mais completa para átomos de confusão citando quatro características necessárias para um padrão de código ser considerado um átomo de confusão [2]:

- Ser precisamente identificável.
- Ser provável de causar confusão.
- Ser substituível por um padrão que causa menos confusão e que tenha a mesma função.
- Ser indivisível.

É importante notar que a confusão nesse caso não deve ser uma avaliação subjetiva do código, e sim uma característica verificada através de estudos empíricos com desenvolvedores. E quanto à identificação precisa, ela também se refere a um certo grau de objetividade que é necessário ao definir o que é um átomo de confusão.

No exemplo de átomo a seguir na linguagem C (veja Figura 2.1), uma variável V1 de tipo "char"recebe o valor 104. Esse valor 104 se refere ao valor ascii 'h' e a saída desse programa seria a impressão do caracter h, porém não seria absurdo imaginar que o comando "print"imprimiria o valor 104. A versão transformada remove essa confusão atribuíndo à variável V1 diretamente ao valor 'h'.

```
void main() {
    char V1 = 104

    printf("%c", V1)
}
```

(a) Átomo

```
void main() {
    char V1 = 'h'

    printf("%c", V1)
}
```

(b) Transformado

Figura 2.1: Exemplo de átomo de confusão, adaptado de [1].

2.2 Trabalhos relacionados

2.2.1 Code Smells e Antipatterns

Code smells foi um termo criado por Kent Beck para definir algo no código que era indicativo de um problema mais profundo [8]. Um exemplo de code smell é código duplicado. Em geral, elementos de código devem ser definidos de forma coesa para serem reutilizáveis em diferentes partes de uma sistema de software. Duplicação então pode significar um design que não modela o problema da forma mais eficiente, e que pode acabar dificultando a manutenção e modificação do código.

Antipatterns são coisas que parecem soluções mas que na verdade são uma espécie de armadilha para desenvolvedores [9]. Um exemplo em Swift é o uso excessivo de "try!"e "try?". Try é uma palavra-chave que é usada para chamar funções que podem retornar erros. Normalmente os erros retornados por essas funções são tratados pelo chamado contido dentro de um bloco "catch". Quando programadores sabem que a função nunca irá retornar um erro, podem usar "try!"para escrever o código de uma forma mais concisa, sem se preocupar com o "catch". Porém caso a funçâo dê erro, a aplicação irá falhar com um erro de execução. E o "try?"é usado quando o programador não precisa se importar com os erros específicos lançados pela função chamada e apenas quer saber no resultado se houve sucesso ou não. Esses são usos válidos de "try!"e "try?", mas por serem tão convenientes, programadores acabam os usando em excesso, o que compromete a qualidade do código de tratamento de erros de uma aplicação [10].

Code smells e antipatterns são dois conceitos que se assemelham aos átomos de confusão pela sua relação com a complexidade acidental mencionada anteriormente. O estudo de átomo de confusão também está principalmente preocupado com legibilidade de código e os seus benefícios. Entretanto, há algumas diferenças.

Primeiramente, uma diferença entre átomos de confusão e os conceitos de code smells e antipatterns [2] está na natureza empírica e não subjetiva dos átomos de consfusão. Os átomos de confusão são apenas candidatos até as pesquisas comprovarem que eles se encaixam nos 4 critérios mencionados

acima.

Além disso, átomos de confusão ficam restritos aos mínimos padrões de código, enquanto code smells e antipatterns podem descrever problemas arquiteturais que envolvem a relação entre várias classes contendo milhares de linhas de código, com intuito de reduzir a complexidade acidental de código de maneira mais ampla.

2.2.2 Compreensão de Código

Minelli et al. usou um programa que registra batidas de teclas e movimentos de mouse dentro da ambiente de desenvolvimento integrado (IDE) Pharo com programadores da linguagem Smalltalk para determinar a quantidade de tempo de desenvolvimento gasta com cada atividade, incluindo atividades de entendimento de código como a leitura do código [3]. A estimativa feita é que 70% do tempo dos desenvolvedores tenha sido gasta com entendimento de código. Acreditamos que um futuro estudo sobre átomos de confusão poderia ser feito com base em dados de interação com IDEs, investigando a diferença de tempo de entendimento de código em arquivos com maior número de átomos de confusão.

Dolado et al. relatou uma redução significativa na capacidade de compreensão de códigos em C com efeitos colaterais através de um questionário. Efeitos colaterais são qualquer mudança no estado do programa após a avaliação de uma expressão [11]. A dificuldade de entender código com efeitos colaterais dá embasamento adicional para um dos candidatos a átomo de confusão propostos no Capítulo 3.

Alguns estudos usaram a neurociência para analisar a atividade cerebral durante tarefas de compreensão de código. Siegmund et. al. detectou atravês de ressonância magnética que as áreas do cérebro responsáveis por atividades que envolvem memória de trabalho, atenção, e processamento de linguagem são ativadas durante tarefas de compreensão de código [12]. Martin et. al. investigou a diferença em atividade cerebral (medida pelas correntes elétricas registradas numa eletroencefalografia (EEG)) na compreensão de pequenos trechos de código com diferentes dificuldades [13]. Os resultados substanciam que há uma desigualdade entre as leituras de códigos com e sem átomos de confusão.

2.3 A Linguagem Swift

Swift é uma linguagem de programação moderna criada pela Apple para ser usada na criação de software para os dispositivos do seu ecossistema. Algumas de suas características são Generics, Opcionais, Closures e Extensions. Swift é uma linguagem orientada a objetos, com alguns aspectos funcionais (funções map, filter e reduce) e algumas capacidades de baixo nível (manipula-

ção de bits). A Apple desenvolve bibliotecas disponíveis em Swift para diversos tipo de aplicações (UIKit, ARKit, CoreML, GameplayKit) [14]. A empresa Apple vem usando Swift também para investir na educação de jovens e crianças através do aprendizado em programação. O aplicativo Swift Playgrounds, disponível para iPad, oferece uma forma interativa de aprender a programar [15].

Swift é uma linguagem com uma grande comunidade. No Stack Overflow 2018 Developer Survey, Swift alcançou décimo quarto lugar em popularidade e recebeu a colocação de sexta linguagem mais amada [16]. Em 2015 Swift se tornou código aberto [17], e desde então há uma participação maior de membros da comunidade no desenvolvimento das suas características, através das chamadas propostas de evolução [18]. Um dos resultados dessa abertura foi a adaptação da linguagem para uso em sistemas Linux. A partir disso, várias frameworks web também foram lançadas que permitem que web apps e APIs sejam escritas em Swift [19].

Capítulo 3

Candidatos a Átomos de Confusão em Swift

Este capítulo apresenta a abordagem para complementar uma lista de candidatos de átomos de confusão para a linguagem Swift e de trechos de códigos feitos para representá-los. Mostramos uma descrição de cada um dos candidatos que foram usados no estudo piloto descrito no Capítulo 4. A lista de candidatos completa está presente no final do capítulo (veja Tabela 3.2 na página 12).

3.1 Metodologia

Tivemos acesso a uma lista inicial de 16 candidatos para átomo de confusão em Swift e 24 pares de questões que exemplificavam alguns dos mesmos [2]. Desses 16, seis eram átomos confirmados do estudo de existência de átomos de confusão em C [1] que eram traduzíveis para Swift. Dois candidatos foram apresentados sem explicação suficiente sobre a sua origem e imaginamos que tenham surgido a partir do conhecimento prévio do autor do estudo em Swift. Um dos candidatos foi encontrado através de uma análise estática de repositórios Swift para detectar as características menos usadas da linguagem. Os candidatos restantes foram descobertos a partir de conhecimento de especialistas e a documentação da linguagem.

Esses átomos de confusão propostos são categorizados de acordo com as fontes de confusão [2], ou seja, diferentes átomos de confusão podem ser confusos por motivos diferentes e não mutualmente exclusivos. Um átomo de confusão pode ser confuso por envolver uma característica pouco usada da linguagem de programação, ou por ser uma característica da linguagem que envolve regras complexas e não-óbvias. Dado isso, o fato de desenvolvedores mais experientes serem menos sujeitos a interpretarem erroneamente um padrão de código contendo um átomo faz sentido, já que têm maior conhecimento dos diversos aspectos e regras de uma determinada linguagem. [1]. (veja Tabela 3.1 na página 8 para uma lista das fontes de confusão estudadas).

Dado que um dos objetivos do estudo era analisar os átomos de confusão existentes, primeiro elaboramos exemplos novos para os candidatos presentes na lista inicial. Depois levantamos candidatos adicionais e analisamos usando

Fontes de Confusão	**Descrição**	**Átomos Afetados**
Aspectos incomuns	Uso de aspectos pouco conhecidos da linguagem. Essa é uma fonte de muitos dos átomos de confusão.	Codificação de Literais Operador Ternário Declaração Defer Hexadecimais com expoentes
Uso de um aspecto além do significado léxico	Alguns aspectos normalmente são usados de uma forma específica. No entanto, há outras formas menos comuns que tendem a ofuscar o significado do programa.	Map com Efeitos Colaterais
Regras complexas	Ambiguidade e falta de clareza porque regras da linguagem não são óbvias ou explícitas.	Precedência de Operadores Operador como Argumento Encadeamento de Opcionais Opcionais Implicitamente Desembrulhados Casamento de Padrões Avançado Trailing Closures Atualizar Opcional com Valor Nil
Práticas de programação ofuscantes	Padrões de uso de aspectos comuns da linguagem de programação que ofuscam o significado do programa.	Operador de Negação em Condicional Argumentos Abreviados Extensões

Tabela 3.1: Fontes de confusão. Cada uma representa um motivo em potencial para um padrão de código ser confuso. Adaptado de [2]

as mesmas técnicas. Esse levantamento teve início com um aprofundamento na documentação da linguagem nas suas diversas formas (manuais, livros, fóruns e até podcasts) [20, 21, 22].

3.2 Novos candidatos

Dentre muitos candidatos levantados ao longo do estudo, foram adicionados à lista três novos candidatos: Map com Efeitos Colaterais, Operador de Negação em Condicional e Casamento de Padrões Avançado.

3.2.1 Map com Efeitos Colaterais

Map com Efeitos Colaterais (veja Figura 3.1 na página 9) foi pensado a partir de uma recomendação contida num livro de Swift, onde era mencionado que a função map deveria ser usada apenas para realizar transformações no array que está sendo mapeado, e que para iterações que realizassem efeitos colaterais seria preferível usar um "for".

A função map faz parte da Swift Standard Library assim como os tipos Int e String e outras funções de alta ordem como reduce e filter.

Uma potencial fonte de confusão relacionada a esse átomo é o fato da função map e outras funções como forEach retornarem uma sequência de tuplas vazias quando o código dentro do closure retorna Void (Void em Swift na verdade é um typealias para a tupla vazia). Durante a pesquisa chegamos a encontrar um exemplo real de um bug causado pela expectativa de ter um array de valores e receber essa sequência de tuplas vazias.

```
func main() {
    var V1 = [1, 2, 3]
    var V2 = [1, 2]

    V1.map { V1 in
        V2.append(V1)
    }

    print(V2)
}
```

(a) Átomo

```
func main() {
    var V1 = [1, 2, nil, 3]
    var V2 = 1

    for V3 in V1 {
        if let V4 = V3 {
            V2 = V2 * V4
        }
    }

    print(V1, V2)
}
```

(b) Transformado

Figura 3.1: Candidato a átomo de confusão Map com Efeitos Colaterais.

3.2.2 Operador de Negação em Condicional

```
func main() {
    let V1 = true
    var V2 = 8

    if !V1 {
        V2 = V2 * 2
    } else {
        V2 = V2 + 2
    }

    print(V2)
}
```

(a) Átomo

```
func main() {
    let V1 = true
    var V2 = 8

    if !V1 {
        V2 = V2 * 2
    } else {
        V2 = V2 + 2
    }

    print(V2)
}
```

(b) Transformado

Figura 3.2: Candidato a átomo de confusão Operador de Negação em Condicional.

O candidato Operador de Negação em Condicional (veja Figura 3.2) foi encontrado quando uma investigação sobre uma proposta de evolução da linguagem relacionada ao uso de booleanos referenciava uma discussão nos fóruns oficiais de Swift [22]. Num dos posts foi apresentado tanto a forma supostamente confusa quanto a alternativa. Embora possa parecer para alguns programadores um padrão que não causaria confusão pela sua simplicidade e por ser usado em uma grande quantidade de linguagens de programação, ainda pode ser confuso para programadore iniciantes. Além disso, classifica-

mos esse candidato como uma prática de código ofuscante [2] (veja Tabela 3.1 na página 8). Isso significa que a confusão na leitura desse código seria ligado não apenas a dificuldade de saber o que o operador faz, que seria confuso apenas para pessoas muito iniciantes na programação, mas principalmente por ser um código um pouco menos explícito na sua intenção e que requer um pouco mais de atenção do que a alternativa.

3.2.3 Casamento de Padrões Avançado

```
func main() {
    var V1 = 10
    var V2 = 8

    if case 0..<V1 = V2 {
        V2 = 1
    } else {
        V2 = 2
    }

    print(V1)
}
```

(a) Átomo

```
func main() {
    var V1 = 10
    var V2 = 5

    if 0 <= V2 && V2 < V1  {
        V2 = 1
    } else {
        V2 = 2
    }

    print(V2)
}
```

(b) Transformado

Figura 3.3: Candidato a átomo de confusão Pattern Matching Avançado.

Esse candidato surgiu como uma amálgama de dois protocandidatos que envolviam usos peculiares de casamento de padrão permitidos em Swift. Embora haja diversos padrões de código para se estudar em relação a casamento de padrões em Swift, foi decidido que esse candidato representaria o uso das palavras-chave "case let" fora de um switch. Nesse caso "if case let" , "for case let" e "guard case let" (veja Figura 3.3 na página 14).

3.3 Candidatos Descartados

Vários candidatos a átomo de confusão propostos foram descartados ao longo da pesquisa. Quatro dos candidatos descartados (Código Morto, Interrogação com Múltiplos Significados, Exclamação com Múltiplos Significados e Variáveis Reutilizadas) estavam na lista inicial de candidatos a átomos de confusão em Swift. Nessa seção descreveremos alguns desses candidatos junto com os motivos para serem descartados.

O primeiro candidato a ser descartado foi Código Morto (veja Figura 3.4 na página 11). Chegamos a traduzir um conjunto das questões da lista de

```
func main() {
    var V1 = 1

    V1 = 3
    V1 = 2

    print(V1)
}
```

(a) Átomo

```
func main() {
    var V1 = 1

    V1 = 2

    print(V1)
}
```

(b) Transformado

Figura 3.4: Candidato a átomo de confusão descartado Dead Code.

questões de [1] de C para Swift, incluindo as do candidato Código Morto. Entretanto, devido ao fato de que o estudo de existência de [1] avaliou Código Morto como tendo uma confusão insignificativa, não vimos necessidade de inclui-lo no nosso estudo.

Os candidatos Interrogação com Múltiplos Significados (veja Figura 3.5 na página 12) e Exclamação com Múltiplos Significados foram excluídos por não terem um significado claro e atômico.

Ao decorrer do tempo, vimos que muitos dos significados do símbolo ? tinham sido explorados em candidatos como Operador Ternário, Encadeamento de Opcionais e Atualizar Opcional com Valor Nil. Enquanto isso, o símbolo ! foi explorado nos candidatos Opcional Explicitamente Desembrulhado e Operador de Negação em Condicional.

Enquanto Interrogação com Múltiplos Significados e Exclamação com Múltiplos Significados foram excluídos por serem genéricos demais, os candidatos For Case Let (veja Figura 3.6) e Ligação de Valor Externa foram considerados específicos demais. A combinação de palavras-chave "case let" não é específica ao laço "for", e pode ser usado em vários tipos de declarações. Enquanto isso, a Ligação de Valor Externa também estava associada ao uso do "case let". Por isso, foram trocados pelo candidato Casamento de Padrões Avançado, que representa diferentes usos de "case let".

3.4 Lista Canônica

Começamos com uma lisra de 16 candidatos, dessa lista descartamos 4 e propomos mais 3 para obter no final uma lista com 15 candidatos (veja Tabela 3.2 na página 12). A partir dessa lista modificada preparamos um estudo piloto para começar a validação dos candidatos, descrito no Capítulo 4.

Alguns dos candidatos são características bastante úteis da linguagem Swift, exemplos disso são os candidatos Defer e Extensões. Extensões são bastante comuns, enquanto Defer é muito pouco usado. Outros candidatos são for-

```
func main() {
    var V1 = false
    var V2 = (V1 ?? false) == false
        ? "abc"
        : "xyz"

    print(V2!)
}
```

(a) Átomo

```
func main() {
    var V1 = false
    var V2: String?

    if let V3 = V1 {
        if V3 == false {
            V2 = "abc"
        } else {
            V2 = "xyz"
    } else {
        V2 = "xyz"
    }

    print(V2!)
}
```

(b) Transformado

Figura 3.5: Interrogação com Múltiplos Significados.

mas de açúcar sintático também comuns, como Operador Ternário e Encadeamento de Opcionais. Por causa disso, acreditamos que os átomos de confusão que forem confirmados devem ser divulgados para a comunidade Swift e incorporados em ferramentas como as de análise estática, para que programadores que decidam não removê-los completamente dos seus códigos pelo menos tenham um maior cuidado na hora de usá-los.

Tabela 3.2: Candidatos a átomo de confusão em Swift. Adaptado de [2]

Nome do candidato	**Exemplo**	**Transformado**
Codificação de Literais	`V1 = 0x12 | 0x6`	`V1 = 0b10010 | 0b00110`
Hexadecimals with Exponents	`V1 = 0x50p-2`	`V1 = 0x50 / 4`

Continua na próxima página

Tabela 3.2 – *Continação da página anterior*

Nome do candidato	**Exemplo**	**Transformado**

Operador Ternário	`let V2 = V1 == 3 ?  2 :  1`	`if V1 {`
		`  V2=2`
		`} else {`
		`  V2=1`
		`}`

Precedência de Operadores	`let V1 = 2 - 4 / 2`	`let V1 = 2 - (4 / 2)`
Declaração Defer	`defer { V1 += 1 }` `V1 = 3`	`V1 = 3` `V1 += 1`
Argumentos Abreviados	`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`
		`}`

Encadeamento de Opcionais	`var V2 = V1[0]?[2]?.magnitude`	`if let V2 = V1[0] {`
		`  if let V3 = V2[2] {`
		`  V2 = V3.magnitude`
		`  }`
		`}`

Operador como Argumento	`V2 = V1.reduce(0,-)`	`V2 = V1.reduce(0) {`
		`  V3, V4 in`
		`  V3 - V4`
		`}`

Opcionais Implicitamente Desembrulhados	`let V1:  Int!  = 10`	`let V1:  Int?  = 10`

Extensões	`extension Int {`
	`  func V2() -> Int {`
	`  if (self < 10) {`
	`  return self`
	`  } else {`
	`  return -1`
	`  }`
	`  }`
	`}`
	
	`let V1 = 1`
	`let V2 = V1.V2()`
		`let V1 = 1`
		`let V2 = V1.V2()`
		`if (V1 < 10) {`
		`  V2 = V2`
		`} else {`
		`  V2 = -1`
		`}`

Atualizar Opcional com Valor Nil	`var V1:Int?  = nil`
	`V1?  = 4`
		`var V1:Int?  = nil`
		`if V1 != nil {`
		`  V1 = 4`
		`}`

Map com Efeitos Colaterais	`V1=[1,2,3]`
	`V1.map{V2 in doSideEffect(V2)}`
		`V1=[1,2,3]`
		`for V2 in V1 {`
		`  doSideEffect(V2)`
		`}`

Operador de Negação em Condicional	`if !V1 {`
	`  print(V2)`
	`} else {`
	`  print(V3)`
	`}`
		`if V1 == false {`
		`  print(V2)`
		`} else {`
		`  print(V3)`
		`}`

Casamento de padrões avançado	`if case 0..<V1 = V2 {`
	`  print(V1)`
	`} else {`
	`  print(V2)`
	`}`
		`if case 0 <= V2 && V2 < V1 {`
		`  print(V1)`
		`} else {`
		`  print(V2)`
		`}`

```
func main() {
    var V1 = [1, 2, nil, 3]
    var V2 = 1

    for case let i? in V1 {
        V2 = V2 * i
    }

    print(V1, V2)
}
```

(a) Atom

```
func main() {
    var V1 = [1, 2, nil, 3]
    var V2 = 1

    for V3 in V1 {
        if let V4 = V3 {
            V2 = V2 * V4
        }
    }

    print(V1, V2)
}
```

(b) Transformed

Figura 3.6: Candidato a átomo de confusão descartado For Case Let.

Capítulo 4

Estudo Piloto

Esse capítulo descreve o estudo piloto feito para validar o formulário construído e os trechos de código de cada candidato a átomo de confusão. Primeiro é apresentada a metodologia que descreve o objetivo. Depois, a organização do estudo piloto junto com alguns detalhes técnicos da pesquisa. Após isso, são apresentados os resultados, sendo divididos em melhorias no formulário e melhorias nos trechos de código.

4.1 Metodologia

Esse estudo não foi feito para validar os candidatos em si mas, sim, para poder reforçar nossa confiança num estudo posterior feito com um formulário e conjunto de questões aprimoradas.

Um aluno de iniciação científica criou aplicação web usando Google Apps Script para que pudéssemos preparar o formulário a partir de tabelas feitas com as listas de perguntas e fazendo a devida aleatorização das perguntas para cada pessoa que fosse realizar o questionário.

Com a ajuda de dois estudantes, foram preparados um total de 90 pares de trechos de código para os 15 candidatos canônicos, mencionados no Capítulo 3. Cada trecho de código foi utilizado como uma questão de código colocada formulário. Os estudantes não tinham experiência em Swift, então precisaram pesquisar sobre a linguagem na documentação para fazer as questões. Ainda assim, houve uma verificação para que cada questão criada estivesse dentro do padrão que queríamos para a linguagem Swift. Uma das modificações mais notáveis feitas às questões preparadas por eles foi a remoção de ponto e vírgula no final das declarações. O uso de ponto e vírgula não é necessário em Swift, podendo ser útil em algumas raras ocasiões como na separação de duas declarações na mesma linha. O seu uso normalmente indica falta de experiência com a linguagem [20]. A preparação das questões fez parte do processo de entendimento de cada átomo, sendo uma forma de explorar as diversas situações em que cada padrão de código poderia aparecer.

Assim como no estudo original, o formulário escolhe aleatoriamente dois pares de cada três pares de questões para diminuir o tempo para completá-lo. Com um total de 15 átomos e 90 pares de questóes, cada sujeito do estudo piloto respondeu a 60 questões. Exemplos das questões podem ser vistos nas

figuras 3.1, 3.2 e 3.3.

O estudo original busca prevenir um efeito de aprendizagem através das medidas de aleatorização da ordem de apresentação das questões e de uma distância mínima entre cada questão e seu par transformado.

A aleatorização da ordem de apresentação da sequência de questões foi feita de forma gulosa com base no algoritmo Fisher-Yates, embaralhando cada elemento na sequência até a sequência inteira estar aleatorizada. Adicionalmente, foi feita a modificação de reiniciar o embaralhamento cada vez que um elemento sorteado não atendesse a restrição de distancia mínima entre as questões.

No questionário medimos o tempo de resposta para cada questão para poder comparar a diferença de tempo de cada questão com o seu par transformado. Nas instruções pedimos que as questões fossem respondidas de maneira rápida e sem pausas. Como o formulário foi online, não podemos garantir que as questões foram respondidas sem nenhuma interferência de distrações irrelevantes. Para melhorar a validade dos resultados, ignoramos outliers no tempo de resposta. Sugerimos que pesquisas futuras sejam feitas com supervisão para garantir maior acurácia.

Com as questões e o formulário prontos, fizemos um estudo piloto para determinar pontos a melhorar tanto na elaboração e apresentação das questões quanto no próprio sistema do formulário.

4.2 Resultados

O formulário foi enviado para 5 participantes através de um contato com pessoas próximas que eram conhecidas pelo autor. Todos os participantes tinham alguma conexão com o programa Apple Developer Academy, sendo três alunos e dois professores. O programa é focado em desenvolvimento iOS, e é comum alunos concluirem o programa tendo publicado aplicativos na App Store. Todos os que receberam o formulário tinham mais de dois anos de experiência em Swift. Pelo menos dois dos estudantes também tinham experiência ensinando sobre Swift em projetos promovidos pela UFPE. A partir do feedback desses participantes, foi possível melhorar tanto o formulário quanto as questões.

4.2.1 Melhorias no formulário

Foi necessário fazer um ajuste no algoritmo de ordenação das perguntas pois vimos que havia um bug em que, em alguns casos, a distância mínima não era mantida.

Pensávamos que o carregamento das questões em formato de imagens poderia ser muito lento, mas isso não foi uma questão significativa para nenhum dos participantes.

Um dos participantes errou ao deixar uma resposta em branco acidental-

mente, foi feito uma modificação para pedir a confirmação do usuário antes de submeter uma resposta em branco.

O formulário não funcionava em dispositivos móveis. Um dos participantes tentou preencher o formulário no seu celular e acabou não podendo submeter as respostas.

Sendo o formulário bastante extenso, houve um caso em que um dos participantes acidentalmente recarregou a página e perdeu o seu progresso. Pedimos para que esse participante não refizesse o formulário pois seria provável que as suas respostas fossem diferentes devido ao efeito de aprendizagem. Foi feita uma modificação para que o usuário pudesse restaurar a sua sessão no caso de um acidente.

4.2.2 Melhorias nas questões

Um dos professores que fez a pesquisa questionou o uso de variáveis começando com letra maiúscula (V1, V2, V3...) por não seguir uma diretriz de código Swift. Após discussão, foi decidido manter a forma em que as variáveis foram nomeadas, pela importância de ter variáveis sem valor semântico associado e pelo fato que não afetaria o resultado dos trechos de código. Porém, foi visto que devemos preservar o estilo "Swifty" de código quando possível para evitar complexidade acidental.

Algumas questões estavam extensas e/ou complexas demais e portanto continham uma dificuldade adicional não relacionada ao candidato a átomo de confusão sendo testado. Por exemplo, uma das questões de código que investigava o candidato Trailing Closure envolvia uma função além da função main() e dentro dessa função havia um laço while. O exemplo foi modificado para que houvesse apenas a função main, sem nenhum loop. Outra questão do candidato Trailing Closure tinha uma expressão com trailing closure que retornava um valor que era usado como input para uma outra função.

Em outra questão era necessário somar uma progressão aritmética de 10 a 100 com 10 intervalos. Mais de um participante se complicou especificamente com essa soma e chegaram a errar o resultado da questão.

Foi observado também que a maioria dos participantes usou a caixa de texto de resposta como registrador para gravar valores calculados durante a execução mental do código. Isso foi visto como um sinal de que a confusão poderia vir da complexidade de armazenar mentalmente um número de variáveis e não apenas da sintaxe e semântica de um candidato a átomo de confusão.

O fato do candidato gastar tantos ciclos de energia cerebral em aspectos de código que não tem a ver com o candidato prejudicaria a validade das respostas. Dado essas observações, essas questões e outras com problemáticas semelhantes foram simplificadas para focar a atenção dos exemplos apenas nas partes que têm relação com os padrões de código dos candidatos à átomo, eliminando códigos irrelevantes e operações matemáticas complexas e reduzindo o número de variáveis.

4.2.3 Validações preliminares

Por ter sido um estudo piloto, não foi feita análise estatística das respostas por serem tiradas de uma amostra muito pequena. Mesmo assim, tentamos extrair alguns conhecimentos sobre os candidatos de confusão através de uma breve observação das respostas.

Ignoramos em parte as respostas para questões que foram modificadas após as modificações descritas anteriormente, dado que a demora maior para responder algumas das perguntas poderia então ser explicado por uma má formulação das mesmas.

Um dos participantes, por exemplo, disse não entender o significado do comando Defer, chegando a deixar algumas das questões relacionadas a esse átomo em branco. Os participantes tiveram dificuldades nas questões que envolviam operadores de mais baixo nível, também deixando em branco questões que envolviam literais com valor em hexadecimal como Mudança de Codificação de Literal e Hexadecimais com Exponentes. Nenhum participante errou as questões do candidato Operador como Argumento. Sobre o candidato Trailing Closures, um participante acreditou que havia questões repetidas devido ao fato de ver duas perguntas muito parecidas, sendo a única diferença a sintaxe do "closure". A participante acertou os dois pares de questões relacionados a esse candidato, e esse comentário pode indicar que um Trailing Closure não faz diferença significativa na legibilidade de um trecho de código.

Capítulo 5

Considerações Finais

Entendimento de código é fundamental para a prática de programação. A identificação de átomos de confusão é um modo promissor de melhorar o entendimento de programas através da detecção dos menores padrões de código confusos com dados empíricos.

Este trabalho buscou expandir e refinar uma pesquisa inicial de candidatos a átomo de confusão na linguagem Swift com base nos estudos originais sobre átomos de confusão e por meio de um estudo de materiais específicos da linguagem Swift. A partir desses candidatos, foi preparado um formulário e realizado um estudo piloto com 5 participantes.

Com base nos resultados obtidos, foi possível aperfeiçoar os exemplos de código para ter maior certeza que a confusão medida na pesquisa final seja devido à capacidade de confusão dos candidatos e não relacionada a outros aspectos confusos nos trechos de código.

Por fim, foram feitas algumas alterações e correções no sistema do formulário para facilitar seu preenchimento correto e viabilizar a pesquisa final.

5.1 Trabalhos Futuros

O próximo passo dessa pesquisa é distribuir o formulário com as questões modificiadas para uma população maior e fazer uma avaliação do nível de confusão de cada candidato, testando a existência dos átomos de confusão em Swift.

Uma eventual validação dos candidatos a átomo de confusão em Swift poderia se transformar numa série de propostas de evolução para a linguagem, onde cada proposta seria a de eliminar um átomo de confusão específico. Caso o padrão de código não pudesse ser eliminado por motivos como de retrocompatibilidade de código ou apenas não recebesse apoio suficiente, a detecção de átomos de confusão em código poderia ser uma nova categoria de avisos do compilador de Swift. Outro cenário seria a simples adição de regras de detecção de átomos de confusão para ferramentas de análise estática, como o SwiftLint [23].

Caso seja validada a existência dos átomos, deverá ser feito um estudo de impacto, também descrito por [1]. No estudo de impacto, são feitas uma quantidade menor de questões incluindo múltiplos átomos de confusão por questão

para determinar o impacto da existência de átomos de confusão no entendimento de um código mais genérico.

Quando for determinado o impacto dos átomos, será interessante saber a prevalência desses átomos em repositórios open source e investigar possíveis relações entre bugs e a existência de átomos de confusão, como feito em [6]. Um trabalho que já fez esse tipo de mineração em repositórios Swift foi [10].

5.1.1 Catálogos de átomos

Um objetivo a longo prazo dessa investigação é criar um repositório de átomos de confusão para diferentes linguagens de programação. Para esse fim, será necessário experimentar novas técnicas de descobrir átomos de confusão em diferentes linguagens. Uma opção seria o uso de crowdsourcing, com a disponibilização de um catálogo online onde diversos desenvolvedores poderiam contribuir com candidatos.

Uma alternativa seria o uso de analytics em sites e aplicativos voltados ao aprendizado de programação. O tipo de questão feito no formulário poderia ser facilmente embutida em sites como CodeAcademy ou apps como Unwrap que já oferecem questões muito semelhantes no intuito de familiarizar desenvolvedores, principalmente novatos, com aspectos de linguagens de programação [24, 25].

Referências Bibliográficas

[1] Dan Gopstein, Jake Iannacone, Yu Yan, Lois Anne Delong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. Understanding misunderstandings in source code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 129–139. ACM, 2017.

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

[3] R. Minelli, A. Mocci, and M. Lanza. I know what you did last summer - an investigation of how developers spend their time. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35, May 2015.

[4] Akond Rahman. Comprehension effort and programming activities: Related? or not related? In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 66–69, New York, NY, USA, 2018. ACM.

[5] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM.

[6] Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. Prevalence of confusing code in software projects: Atoms of confusion in the wild. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 281–291. ACM, 2018.

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

[8] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[9] William H. Brown, Raphael C. Malveau, Hays W. "Skip"McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures,*

and Projects in Crisis. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1998.

[10] Nathan Cassee, Gustavo Pinto, Fernando Castor, and Alexander Serebrenik. How swift developers handle errors. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 292–302, New York, NY, USA, 2018. ACM.

[11] J. J. Dolado, M. Harman, M. C. Otero, and L. Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE Trans. Softw. Eng.*, 29(7):665–670, July 2003.

[12] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 378–389, New York, NY, USA, 2014. ACM.

[13] M. K. . Yeh, D. Gopstein, Y. Yan, and Y. Zhuang. Detecting and comparing brain activity in short program comprehension using eeg. In *2017 IEEE Frontiers in Education Conference (FIE)*, pages 1–5, Oct 2017.

[14] Apple. Swift language website. `https://swift.org`.

[15] Apple. Swift playgrounds. `https://www.apple.com/swift/playgrounds/`.

[16] Stack Overflow. 2018 stack overflow developer survey. `https://insights.stackoverflow.com/survey/2018/`. Accessed: 2018-11-25.

[17] Apple. Swift is open source. `https://developer.apple.com/swift/blog/?id=34`.

[18] Apple. Swift evolution proposals. `https://github.com/apple/swift-evolution`. Accessed: 2018-11-25.

[19] Vapor. Vapor framework website. `https://vapor.codes`.

[20] Erica Sadun. *Swift Style: An Opinionated Guide to an Opinionated Language*. Pragmatic Bookshelf, 1st edition, 2017.

[21] Swift By Sundell. 19: "proposal tour", with special guest erica sadun. `https://www.swiftbysundell.com/podcast/19`. Accessed: 2018-11-25.

[22] Chris Eidhof, Ole Begemann, and Airspeed Velocity. *Advanced Swift: Updated for Swift 4*. CreateSpace Independent Publishing Platform, USA, 3rd edition, 2017.

[23] yonaskolb. Swiftlint. https://github.com/realm/SwiftLint.

[24] Paul Hudson. Unwrap. https://github.com/twostraws/Unwrap/.

[25] CodeAcademy. Codeacademy. https://www.codecademy.com.

[26] Raymond P.L. Buse and Westley R. Weimer. A metric for software readability. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 121–130, New York, NY, USA, 2008. ACM.

[27] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations, 1997.

[28] Forrest Shull, Janice Singer, and Dag I.K. Sjøberg. *Guide to Advanced Empirical Software Engineering*. Springer-Verlag, Berlin, Heidelberg, 2007.

[29] E. Avidan and D. G. Feitelson. Effects of variable names on comprehension: An empirical study. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 55–65, May 2017.

[30] Chris Eidhof. Se-0199: Adding toggle method to bool. https://forums.swift.org/t/se-0199-adding-toggle-method-to-bool/9841. Accessed: 2018-11-25.

[31] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall. Context is king: The developer perspective on the usage of static analysis tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, volume 00, pages 38–49, March 2018.

Este volume foi tipografado em LaTeX na classe UFPEThesis (www.cin.ufpe.br/~paguso/ufpethesis).