As motivações para o rejuvenescimento de software: uma abordagem em grounded theory acompanhada de NLP

Yan Pietro Barcellos Galli

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Rodrigo Bonifácio

Brasília
2021

As motivações para o rejuvenescimento de software: uma abordagem em grounded theory acompanhada de NLP

Yan Pietro Barcellos Galli

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Rodrigo Bonifácio (Orientador)
CIC/UnB

Prof. Dr. Edna Dias Canedo
UnB

Walter Lucas Monteiro de Mendonça
UnB

do Curso de Engenharia da Computação

Brasília, 10 de novembro de 2021

Dedicatória

Dedico este trabalho a todo o curso de Engenharia de Computação da Universidade de Brasília, corpo docente e discente, a quem fico agradeço por dele ter feito parte. Dedico este trabalho também àqueles que me ajudaram ao longo desta caminhada.

Agradecimentos

Aos professores do curso de Engenharia de Computação da Universidade de Brasília, que me forneceram as bases necessárias para a realização deste trabalho, agradeço com profunda admiração por todo o trabalho. Agradeço também ao meu orientador e professor Rodrigo Bonifácio, por todo empenho e ajuda durante o trabalho. Por fim, agradeço à minha família, por sempre me dar todo o suporte para conquistar meus sonhos.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Abstract

Os Engenheiros de Software estão constantemente buscando melhorar seus processos de desenvolvimento de software. Uma parte significativa da melhora do processo de desenvolvimento está contida nas novas formas que vão surgindo de desenvolver o software, por meio das novas funcionalidades que passam a existir quando as linguagens de programação evoluem. Esse processo de evolução, entretanto, não está ligado apenas a questões técnicas e computacionais, mas também a questões sociais e humanas. Na última década, vem ocorrendo um aumento substancial de pesquisas na área de software que exploram as questões humanas, sociais e mercadológicas do software, sendo que em muitos casos é utilizada a metodologia *Grounded Theory* para tal.

No presente artigo, utilizamos a metodologia *Grounded Theory* para evidenciar qual a relação entre a evolução do software e a evolução da linguagem utilizada para o desenvolvimento deste software. Propomos também uma adaptação no processo da metodologia *Grounded Theory*, acrescentando uma etapa quantitativa em seu processo com o uso de Processamento de Linguagem Natural, com o objetivo de encontrar melhores respostas para a pergunta de pesquisa apresentada.

Utilizamos a *Grounded Theory* como um método quanti-qualitativo para estudar 23 entrevistas com desenvolvedores experientes e de experiências diversas para compreender melhor as motivações por trás da evolução do software. Como resultados, apresentamos (a) uma amostra dos resultados alcançados pela aplicação parcial da metodologia de *Grounded Theory* no contexto da evolução do software; (b) uma maneira de aplicar o processamento de linguagem natural no contexto de análises qualitativas.

Keywords: Grounded Theory, NLP, Natural Language Processing, Code Rejuvening, Code Rejuvenation, Software Evolution

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Os Engenheiros de Software estão constantemente buscando melhorar seus processos de desenvolvimento de software, ao passo que os softwares necessitam se adaptar às necessidades de seus usuários à medida que o tempo passa [1]. Com o tempo, as necessidades do mercado e dos mais diversos campos de pesquisa vão evoluindo, tornando os softwares cada vez mais complexos e mais difíceis de serem desenvolvidos e mantidos. Em particular, a evolução das linguagens de computação em muitos casos traz boas soluções para esses problemas que vão surgindo de acordo com as necessidades do próprio mercado [2].

Ao mesmo tempo, podemos observar que a Engenharia de Software vem crescendo seus horizontes quanto aos métodos para validar suas mais diversas hipóteses e compreender o melhor momento para evoluir o código, de maneira eficaz e sem desperdícios. Em especial, os métodos de pesquisa qualitativos estão ganhando muita popularidade neste escopo, dado que envolvem em seus resultados questões de natureza social e comportamental, que são extremamente importantes para a tomada de decisão de evolução do código [3].

Tendo clareza das necessidades dos usuários de um determinado software, torna-se muito mais fácil escolher onde devem ser concentrados os esforços de mudança no software, e, consequentemente, onde o código deve ser mais otimizado e evoluído. Ao compreender essa forte relação entre comportamento social e evolução de software, torna-se natural a busca por alguma metodologia que engloba em seu processo esses dois pontos para entender quais são os grandes motivos que levam à evolução de um software [4].

Um candidato natural para solucionar essa questão é a metodologia *Grounded Theory* (Glaser and Strauss 1999, 2005), que é um método de pesquisa qualitativa cada vez mais procurado no contexto da Engenharia de Software. A metodologia se propõe justamente a gerar como resultado uma teoria embasada em um contexto real a nível social. Buscamos evidenciar, por meio deste artigo, quais os principais motivos que levam à evolução do software, e se existe alguma relação direta entre essa evolução com a evolução das linguagens de programação. Evidentemente essa é uma questão não apenas técnica e

computacional, mas também social. A relação entre as empresas, seus produtos, seus clientes e os desenvolvedores do código que estão por trás dos produtos é algo que deve ser compreendido para que seja tomada uma decisão sobre a evolução do código, e é isso que nos propomos a compreender no presente artigo [4].

Iniciamos a compreensão sobre os fatores que motivam a evolução do código com base na evolução da linguagem, a partir de uma aplicação preliminar da *grounded theory* para algumas de nossas questões de pesquisa, deixando a criação da teoria em si para um trabalho futuro. Por ser um método qualitativo de pesquisa, possui algumas naturais limitações. Dentre elas, estão a dificuldade na generalização de seus resultados e de análises menos suscetíveis às subjetividades do pesquisador. Para mitigar essas limitações, propomos a utilização de uma análise quantitativa utilizando algoritmos de processamento de linguagem natural, com a finalidade de otimizar e objetivar nossos resultados. Para aprofundar em todo esse contexto da evolução do código, aprofundamos aqui nas seguintes perguntas:

> Quais motivos te levam a evoluir o seu software para suportar novos recursos da linguagem?

> Você considera a evolução das linguagens de programação um fator importante para evoluir seu software? Por que?

Com base na aplicação da *grounded theory* no contexto da evolução do código relacionada à evolução da linguagem, nossas contribuições são: (a) uma amostra dos resultados alcançados pela aplicação parcial (não chegamos até a última etapa, do desenvolvimento da teoria final) da metodologia de *grounded theory* no contexto da evolução do software; (b) uma maneira de aplicar o processamento de linguagem natural no contexto de análises qualitativas.

No Capítulo 2 é explicado melhor o contexto de nosso trabalho, a partir de trabalhos relacionados. No Capítulo 3, explicamos com mais detalhes a modalidade da *Grounded Theory* aplicada neste artigo, a forma como abordamos o rejuvenescimento de software e como utilizamos o processamento de linguagem natural para mitigar alguns *gaps* apresentados na metodologia. No Capítulo 4 apresentamos nossos resultados de maneira mais detalhada, e por fim, no Capítulo 5 falamos sobre nossas conclusões e planos para futuros trabalhos.

Capítulo 2

Trabalhos Relacionados

As linguagens de programação de maior sucesso e que são mais usadas são aquelas que são capazes de se adaptar à medida que novas necessidades, em diversos campos, vão surgindo [5]. Essas adaptações se dão por meio de evoluções nas linguagem, e se expressam na prática a partir do versionamento das linguagens. Com o tempo, muitas adaptações vão surgindo, e antigos programas, que utilizam apenas versões mais antigas da própria linguagem, vão ficando mais difíceis de se compreender e consequentemente mais difíceis de se manter [6].

É possível identificar que hoje existe uma grande quantidade dos produtos de mercado que são tecnológicos, e que, logo, necessitam de uma manutenção saudável e eficaz para permanecerem competitivos. Esse processo de manter um software não é algo trivial, pois acaba envolvendo o trabalho de várias áreas em conjunto para que seja feito da maneira mais inteligente [7]. Para tomar esse tipo de decisão na busca de se manter competitivo, é necessário compreender o momento certo para a mudança, compreender se o que propõe as novas versões das linguagens pode trazer algum benefício para o produto de uma forma geral (em termos de desempenho, performance, segurança) e compreender quão grande será o esforço para a mudança.

Em muitos casos, os custos para realizar essa evolução do código são tão altos que já não podem mais ser pagos completamente. Torna-se necessário fazer apenas remendos, ou em alguns casos até reconstruir o produto novamente [8]. Várias são as causas para que esse processo ocorra [7], como: detenção de conhecimento do código por desenvolvedores que saem do projeto e não legam seu conhecimento aos outros desenvolvedores; o software foi desenvolvido sem padrões definidos e acaba virando algo indecifrável, a versão da linguagem utilizada no sistema não evoluiu e poucos desenvolvedores possuem conhecimentos sobre tal versão, entre outros. Vários desses problemas são evitáveis, caso haja, por parte da equipe que gerencia e mantém o software, uma real preocupação em valorizar o software. Fazendo uma simples metáfora, manter o software pode ser comparado a manter

uma casa. De tempos em tempos é necessário realizar uma limpeza, trocar alguns móveis que estão antigos e quebrados e deixar tudo organizado, e isso exige esforço. No caso do código, ele também precisa de manutenções periódicas, além de algumas mudanças de tempos em tempos para que se mantenha limpo, organizado e sustentável.

Além disso, o design de software é algo que também está em constante evolução [5]. A criação do software é uma arte que busca abstrair as leis da natureza, e que de alguma forma vai ganhando vida à medida que as pessoas utilizam o software. Quanto mais mercados o software atinge, mais as mudanças de design e arquitetura tornam-se necessárias, para se adequarem a esses novos horizontes. As linguagens de programação acompanham esse desenvolvimento, dado que elas são a principal ferramenta para dar vida a essas novas abstrações. A linguagem evolui à medida que o ser humano passa a compreender melhor uma ideia, um comportamento, e as formas de expressão dessa linguagem vão se tornando cada vez mais adequadas para contemplar essa mudança de comportamento [9]. Essa questão, naturalmente, envolve não apenas aspectos técnicos: é uma mudança humana. Sendo assim, o rejuvenescimento do software pode ser visto como uma questão de natureza humana, que para ser realizado, necessita de um esforço conjunto entre as diversas áreas de uma empresa para que seja de fato benéfico e útil.

2.1 Rejuvenescimento de Software

À medida que surgem novas demandas de mercado e de pesquisa, o software evolui para se adaptar da melhor forma a essas novas circunstâncias [1]. Os usuários pedem uma nova funcionalidade e ela necessita ser desenvolvida, ou então melhorias de performance necessitam ser feitas para adaptar o software a um determinado público. Um software que não evolui para atender as necessidades de seus clientes [2] acaba também se tornando obsoleto e perdendo sua popularidade.

Seguindo esse mesmo raciocínio, as linguagens de programação de sucesso são aquelas capazes de se adaptar às necessidades de seus usuários [5], dos programadores, de modo a facilitar seu uso para o desenvolvimento daquilo que é pedido pelo mercado e pelas diversas áreas de pesquisa, mantendo ou melhorando, naturalmente, a performance do sistema [10]. Quanto mais otimizada e mais simples for a implementação de uma funcionalidade em determinada linguagem, maior é a chance de um desenvolvedor buscar utilizar essa forma de resolver o problema.

A partir de nossos estudos, buscamos compreender de maneira mais profunda qual a relação entre a evolução de código e a evolução de sua linguagem, incluindo questões como a facilidade de se evoluir o código hoje, quando vale a pena evoluir o código e algumas outras questões. Para realizar esse estudo, utilizamos uma metodologia qualitativa chamada

Grounded Theory, por ser uma metodologia muito adequada para o desenvolvimento de uma teoria científica na área de software.

2.2 Grounded Theory

A *Grounded Theory* (GT) é uma metodologia de pesquisa qualitativa que surgiu no ramo da sociologia. Ela foi criada a partir da junção das pesquisas de dois sociólogos, Barney G. Glaser e Anselm L. Strauss, que foram os pesquisadores responsáveis por estruturar os procedimentos e estratégias e formular o que hoje conhecemos como a metodologia de GT, documentada no livro *"The Discovery of Grounded Theory – strategies for qualitative research" (Glaser and Strauss; 1967).* Como dizem os autores em seu livro, o nome *Grounded Theory* vem justamente do foco que foi dado em desenvolver uma teoria fortemente fundamentada (*grounded*) em evidências coletadas a partir de práticas do mundo real, com pessoas reais.

À medida que a metodologia foi se desenvolvendo, foram surgindo algumas variações clássicas de sua aplicação, sendo elas: Clássica ou Glaserina (Glaser e Strauss; 1997), Strauss-Corbiniana (1990), Construtivista (Charmaz; 2006), e algumas outras versões mais modernas. Por ter nascido na sociologia, é muito utilizada para compreender o comportamento do ser humano perante as mais diversas situações. Por conta disso, passou a ser muito utilizada na Engenharia de Software, dado que cada vez mais percebe-se a importância de entender o comportamento humano diante de uma aplicação para que posteriormente possam ser propostas mudanças que realmente estejam de acordo com o que foi validado metodologicamente. Essa compreensão otimiza o processo de desenvolvimento do software, pois a partir da compreensão do comportamento do usuário é muito mais simples e eficaz propor novas funcionalidades que realmente serão utilizadas pelos usuários do sistema em questão.

A GT, apesar de ser um método qualitativo de pesquisa, propõe-se a ser um método rigoroso e sistemático, com técnicas e procedimentos muito bem documentados e fundados. Por essa razão passou a ser muito utilizada em pesquisas científicas na área de tecnologia. É um método de pesquisa completo, que dentro de seu processo inclui como devem ser coletados os dados, como os dados devem ser analisados, como deve ser feita a revisão da literatura, como escolher as melhores perguntas de pesquisa e várias outras questões. A partir de toda essa metodologia e de uma terminologia bem definida, propõe-se a realizar investigações de um fenômeno social, de uma maneira iterativa e ágil [11]. A figura 2.1 mostra o modelo de GT que seguimos em nossos estudos, seguindo o modelo clássico.

A metodologia propõe como ponto de partida a seleção de uma área de interesse a ser aprofundada. A partir dessa área de interesse, pode ser realizada uma pequena revisão de

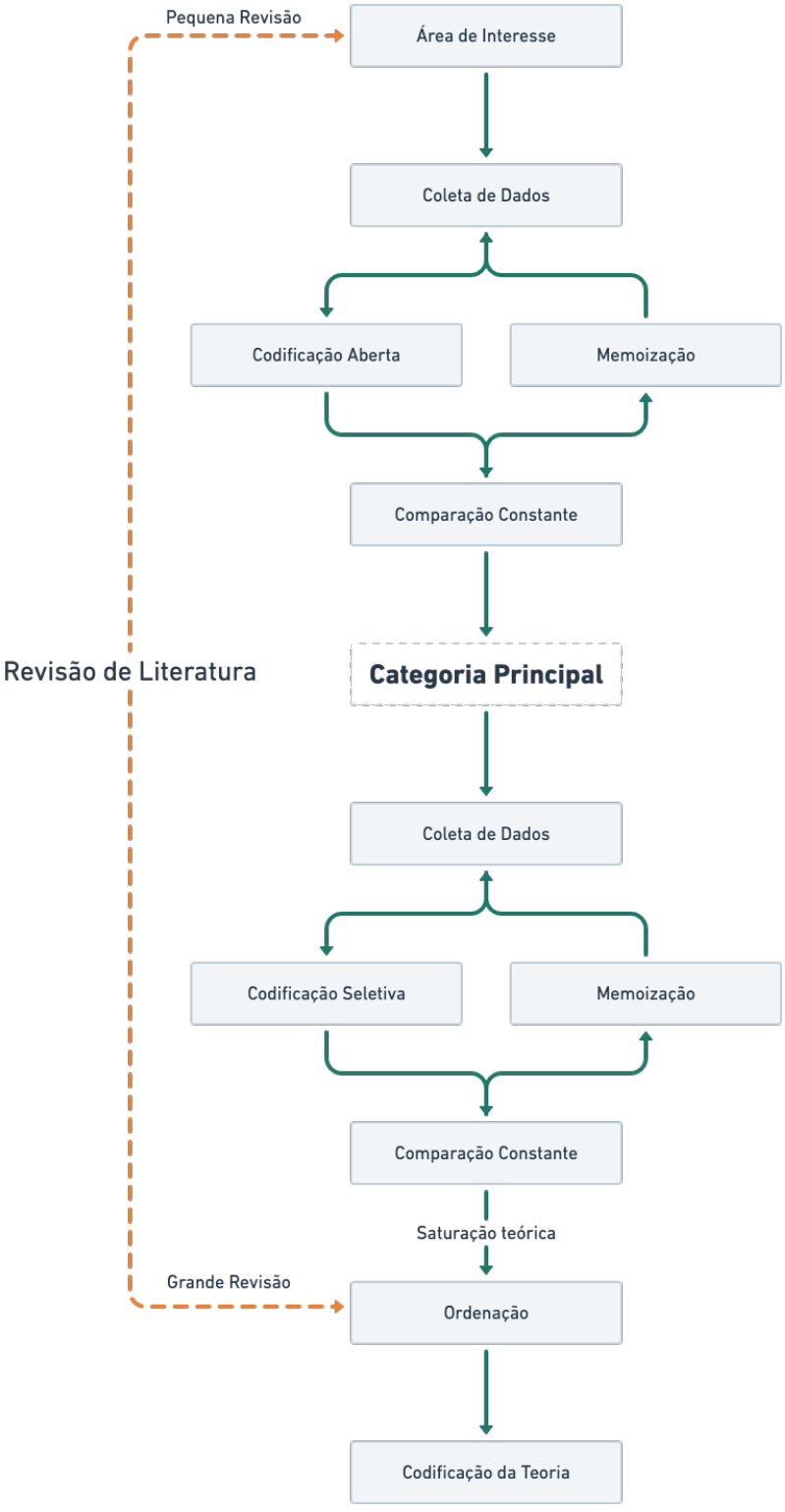

Figura 2.1: Estrutura da Grounded Theory.

literatura ou uma revisão mais extensa, dependendo dos objetivos da pesquisa e do que já está bem fundamentado sobre o assunto [11]. Em seguida, é iniciada a etapa de coleta de dados, que utiliza a técnica de amostragem teórica como base para seu desenvolvimento. São selecionados vários participantes para participar da pesquisa, de acordo com algum critério escolhido pela equipe, e em seguida são realizadas algumas entrevistas. Esse processo de entrevistas deve ocorrer com pessoas que possuem um perfil adequado para o contexto da pesquisa. Para gerarem melhores resultados, as entrevistas devem ser semi-estruturadas, ou seja, devem seguir um protocolo ou roteiro básico, ao mesmo tempo que devem também ser flexíveis, permitindo perguntas extras caso sejam interessantes para uma posterior análise. As entrevistas devem ser transcritas, para que então comece de fato a parte de análise dos textos.

O processo de análise dos dados é realizado, idealmente, de maneira iterativa. À medida que certos aprendizados vão sendo concretizados, o foco das entrevistas pode ir mudando para que seja encontrado o ponto central da pesquisa, o fenômeno principal (*core phenomenon*). Vão sendo realizadas rodadas de codificação aberta, com comparações entre os pontos chave das análises, *memoização* dos resultados e novamente mais coletas de dados, até que emerja uma categoria chave, uma categoria principal. O fim da etapa de codificação aberta é marcado pela definição da categoria principal, que é um ponto que sintetiza as entrevistas por aparecer com muita recorrência.

A categoria principal passa a ser a principal pergunta de pesquisa, e a partir dela é possível passar ao próximo passo: a codificação seletiva. Esse processo consiste na codificação apenas dos dados que se relacionam com a categoria principal. Ao ser atingida a saturação dos dados, é iniciado o processo de ordenação (*sorting*), que consiste em selecionar os principais *memos* gerados no processo anterior. Os memos nada mais são que pequenos resumos das partes mais importantes das entrevistas, e capturam os links entre as diversas categorias que vão surgindo. A partir dessa síntese, torna-se possível o desenvolvimento da teoria final, que evidencia a relação entre os pontos mais importantes que foram encontrados na pesquisa.

Alguns pontos importantes sobre a GT é que ela deve se encaixar muito bem nos dados, explicar bem os dados, ter uma relevância real para o campo estudado e deve ser modificável. A partir dela é possível entrar no problema estudado de maneira profunda, permitindo então a criação de uma teoria fortemente embasada sobre um determinado assunto. Porém, por ser uma metodologia tradicionalmente qualitativa, em muitos momentos revela interpretações subjetivas, pré-disposições e aspectos pessoais de quem realiza a análise, de modo a ter em seus resultados uma dependência da experiência de quem aplica o método. Por conta disso, é um método de difícil escalabilidade e reprodutibilidade. Em busca de mitigar em algum grau esses pontos, propõe-se a utilização de técnicas computacionais

como suporte à metodologia, de modo a evolui-la para um processo quanti-qualitativo. No presente artigo foi aplicado o Processamento de Linguagem Natural (*Natural Language Processing - NLP*) para suprir esse gap que existe na metodologia.

O Processamento de Linguagem Natural (*Natural Language Processing - NLP*) é um subcampo da computação e linguística que se propõe a estudar as melhores maneiras de lidar com a análise de textos em linguagem natural por meio de algoritmos computacionais [12]. Em nosso estudo, por exemplo, foi realizada a análise das transcrições de várias entrevistas por meio desse processamento de linguagem natural. Nosso objetivo principal para o uso da *NLP* está ligado à possibilidade de realizar uma classificação de dados segundo um critério, dentro da metodologia da *grounded theory*. Como a GT propõe a criação de categorias a partir de um processo de codificação, um algoritmo de classificação pode realizar essa tarefa de encontrar os pontos em comum das várias transcrições de uma maneira mais objetiva e otimizada [13].

Capítulo 3

Metodologia

3.1 Questões do Roteiro

Com a finalidade de encontrar os verdadeiros motivos que levam os desenvolvedores a iniciar o rejuvenescimento de seu código, dentro do contexto da *grounded theory*, foram realizadas diversas entrevistas com experientes engenheiros de software que atuam tanto no contexto de mercado quanto no contexto de pesquisa. Algumas das perguntas que foram feitas são as seguintes:

(Q1) Com que linguagem ou quais linguagens você trabalha atualmente?

(Q2) Quanto tempo de experiência você tem com essa ou essas linguagens?

(Q3) Com qual versão da linguagem você trabalha atualmente?

(Q4) Você poderia falar um pouco para nós qual é o seu trabalho, e como você se relaciona com códigos?

(Q5) Você considera a evolução da linguagem um fator importante para evoluir seu código? Por que?

(Q6) Quais fatores te levam ou te levariam a evoluir o seu código?

(Q7) Quais motivos te levam a evoluir o código para suportar novos recursos da linguagem?

(Q8) Quais novos recursos da linguagem você adotou?

(Q9) Quais benefícios você obteve ao adotar as novas construções da linguagem?

(Q10) Qual processo você realiza para atualizar seu código, em softwares em desenvolvimento?

(Q11) Em sua opinião, quais são os motivos que podem impedir um desenvolvedor de evoluir um sistema, a fim de acomodar novas construções de linguagem de programação?

(Q12) Quais são os principais desafios para evoluir o código a partir da evolução da linguagem?

(Q13) Você atualiza seus conhecimentos sobre a linguagem à medida em que esta evolui? Por quê/por quais meios?

(Q14) Com que frequência você aplica os conhecimentos acerca das novas versões da linguagem em seus projetos em andamento? Explique o motivo da sua resposta.

(Q15) É difícil se manter atualizado com as novas versões da linguagem que você utiliza?

(Q16) Você utiliza ou já utilizou alguma ferramenta automatizada para evoluir seu código?

(Q17) Você tem interesse em utilizar ferramentas automatizadas para evoluir seus softwares?

(Q18) Você já teve problemas com a utilização de alguma ferramenta?

3.2 Revisão de Literatura Inicial

Após a escolha da área de interesse para a aplicação da *GT*, segundo a metodologia clássica proposta por *Glaser*, é recomendado que o pesquisador comece a etapa de análise de dados sem se ater a um problema muito específico, para que seja possível, a partir do próprio processo, encontrar qual é o fenômeno principal, a partir das respostas dados. Dessa forma, selecionamos algumas perguntas pertinentes ao escopo de nossa pesquisa, para entender a relação entre a evolução do código e a evolução da linguagem.

Quando há uma boa teoria na literatura sobre o tema buscado, há a possibilidade de partir dessa teoria como um ponto de partida para propor outras teorias emergentes [11]. Porém, como não há nenhuma teoria bem fundada sobre a evolução do software motivada pela evolução da linguagem, buscamos começar nossos estudos e busca por dados sem utilizar da ferramenta de uma revisão de literatura extensa. Fizemos estudos sobre essa área de interesse de modo que fosse possível aprofundar em pontos importantes durante as reuniões.

3.3 Coleta de Dados

Na GT, o processo de coleta de dados é guiado pelo processo de Amostragem Teórica. Esse processo consiste na coleta, codificação e posterior análise dos dados, para que depois seja possível a decisão de quais dados serão coletados em seguida. A amostragem teórica é um processo iterativo e adaptativo, assim como pede a metodologia da GT, pois é construído à medida que são feitas as coletas dos dados com os entrevistados. Faremos aqui uma descrição do nosso processo de coleta de dados como um todo, e quais foram nossas decisões à medida que nossos estudos foram sendo conduzidos.

3.3.1 Escolha dos Participantes

Com base nas perguntas a serem respondidas, percebemos que um perfil específico de participantes seria mais adequado: pessoas com uma boa experiência de mercado ou de pesquisa e que utilizam uma mesma linguagem a um bom tempo. O participante ideal seria, então, uma pessoa com uma boa experiência em alguma linguagem e com uma boa bagagem de projetos que o colocaram à prova sobre ter ou não que realizar a evolução de seu código.

Durante o período de 3 meses nós buscamos inicialmente participantes que eram contatos próximos de todos os envolvidos na pesquisa e que tinham essas características buscadas. Depois da saturação desses contatos, utilizamos a metodologia de *snowballing*, que consiste em pedir para os participantes entrevistados indicarem outras pessoas que eles conhecem para serem também entrevistadas. Os requisitos para a indicação eram os mesmos: ter uma boa experiência de mercado ou de pesquisa e ter tido alguma experiência ou necessidade de evoluir o código em algum momento.

Utilizando essa metodologia, entrevistamos 23 participantes, com uma vasta variedade de experiências e de três países diferentes. A maioria dos participantes está localizado no Brasil, contabilizando vinte e um. Dois participantes estão localizados na Hungria e um participante no Canadá. É importante salientar que todas as entrevistas foram feitas seguindo os procedimentos éticos necessários [14]. As entrevistas foram gravadas para posteriores análises, e seus vídeos e áudios foram mantidos sobre acesso restrito apenas para as pessoas envolvidas no desenvolvimento do presente artigo. Em razão das restrições do período com relação ao vírus COVID-19, todas as entrevistas foram realizadas online. Os vídeos tiveram uma duração média de 32 minutos, sendo que a maior entrevista durou 76 minutos e a menor 11 minutos. Identificamos que a discrepância entre o tempo de algumas entrevistas está relacionada principalmente ao tempo de experiência dos entrevistados. Alguns entrevistados, por terem bastante experiência, traziam muitos

Tabela 3.1: Descrição dos participantes em termos de Experiência.

Experiência	**Quantidade**
Menos de 5 anos	2
De 5 a 10 anos	8
De 11 a 15 anos	5
De 16 a 20 anos	4
Mais de 20 anos	6

Tabela 3.2: Descrição dos participantes em termos do papel de atuação.

Papel	**Quantidade**
Desenvolvedor	20
Líder de Tecnologia	5
Professor	3

pontos relevantes para uma única pergunta, e nesses casos foi comum fazer perguntas extras, para complementar suas respostas sobre algum ponto de interesse.

Como suporte aos nossos resultados, foram incluídas várias citações das entrevistas, como uma forma de evidenciar algumas ideias trazidas pelos nossos entrevistados e como uma forma de mostrar como esses pensamentos contribuíram para os resultados em nossas análises. Como sugere a própria GT, essas citações dos usuários mostram que os entrevistados são uma parte importante da pesquisa, tanto quanto os entrevistadores.

Nas Tabelas, 3.2 e 3.3, observa-se que há uma discrepância entre o número de participantes e as quantidades para cada pepel. A razão para esse discrepância está no fato de alguns entrevistados atuarem em diferentes frentes e também por trabalharem em diferentes tipos de organizações.

3.3.2 Entrevistas

Para a realização das entrevistas, utilizamos os softwares *Microsoft Teams* e *Google Meet* para conversar com os participantes e o software *OBS Studio* para a gravação das entrevistas. Foi criado um roteiro inicial de perguntas a serem feitas aos entrevistados, sendo a maioria delas perguntas abertas. Alguns fluxos de entrevista foram criados, com base nas respostas dadas. Em alguns fluxos, quando a reposta foi negativa para uma

Tabela 3.3: Descrição dos participantes em termos de tipo de organização.

Papel	**Quantidade**
Iniciativa Privada	18
Iniciativa Pública	5
Iniciativa de Pesquisa	2

determinada pergunta, algumas perguntas posteriores deixavam de fazer sentido, e por conta disso eram puladas. Por exemplo, se o entrevistado diz que não acha importante o processo de evolução do código, deixa de fazer sentido perguntar em quais contextos ele acha importante evoluir o código.

Fizemos um treinamento com os entrevistadores sobre como conduzir as reuniões de acordo com o roteiro que foi desenvolvido por toda a equipe, de modo a manter uma homogeneidade de condução, independente do entrevistador. No total, tivemos quatro pessoas diferentes conduzindo as entrevistas a partir desse roteiro semi-estruturado. O fato de serem mais pessoas conduzindo as entrevistas contribui no aspecto da busca por diferentes pontos de vista para propor novas perguntas, com base no que os entrevistados iam respondendo. A escolha das perguntas foi realizada em conjunto com os professores orientadores do projeto e também com base em reuniões que foram sendo feitas entre os envolvidos na pesquisa à medida que o processo como um todo foi se desenvolvendo. Como diz a metodologia da GT, as perguntas de pesquisa para uma boa coleta de dados devem estar ligadas ao *core phenomenon* da pesquisa, ou seja, o fenômeno principal que se busca ser compreendido, que no caso desta pesquisa é a relação entre a evolução do código e a evolução das linguagens de programação.

Além disso, as perguntas devem abordar o que levou esse processo a ocorrer (*causal conditions*), quais as ações que foram tomadas para que o processo ocorresse (*strategies*) e quais foram os resultados após a realização dessas ações (*consequences*). É importante frisar que o roteiro foi apenas um guia de condução das entrevistas. Cada entrevistador tinha a liberdade para fazer perguntas extras, de maneira moderada, caso sentisse que essa nova pergunta pudesse trazer boas respostas para nossas perguntas de pesquisa. Foi perceptível para todos os entrevistadores que quanto mais natural era a entrevista, mais o entrevistado se sentia à vontade para trazer sua experiência e consequentemente mais relevantes eram suas respostas.

3.4 Análise dos dados das Entrevistas

Para iniciar o processo de análise das entrevistas, foi necessário realizar a transcrição das mesmas, como propõe a própria GT. Contratamos profissionais para realizar este trabalho, ao mesmo tempo que parte de nossa equipe também realizou transcreveu e revisou este trabalho, seguindo sempre todas as normas éticas necessárias para tal. As entrevistas foram todas transcritas, e a partir dessas transcrições iniciamos a análise qualitativa das respostas das entrevistas, à medida que elas iam sendo realizadas. Foi, então, realizada uma análise preliminar de algumas das perguntas de pesquisa levantadas.

Dentro do escopo deste trabalho, foram selecionadas duas perguntas para serem analisadas com maior profundidade, sendo elas as perguntas Q5 e Q7. Após a leitura das várias entrevistas, foram escolhidas essas duas perguntas por tratarem diretamente da área de interesse de nossos estudos. As respostas para essas perguntas foram colocadas em uma planilha de comparação, de modo a permitir uma melhor compreensão sobre os pontos em comum entre as respostas. Dos pontos mais relevantes e mais comentados nas entrevistas, fizemos uma síntese para servir como suporte para o trabalho futuro de criar o *framework* conceitual contendo os pontos chave para a criação da teoria.

3.4.1 Análise quantitativa dos dados das Entrevistas

Com o objetivo de adaptar a metodologia de *grounded theory* para a realidade da Engenharia de software e para os contextos mais atuais de pesquisa, é possível evoluir o método de modo a tornar os seus resultados mais objetivos e replicáveis. Isso pode ser alcançado a partir de uma complementação computacional ao método, possibilitando uma análise dos dados mais imparcial e objetiva, e uma extração mais eficaz dos pontos mais comentados nas entrevistas.

A *grounded theory* possui alguns desafios, que devem ser superados se o objetivo é alcançar resultados mais realistas. Por ser um método qualitativo, em sua natureza, é possível que em várias das análises realizadas a partir de seu método, entrem questões subjetivas, pré-disposições e aspectos imparciais oriundos de quem conduz a investigação (Saldana 2015:8). Em razão disso, é difícil replicar o método, mesmo que em contextos muito similares e com um uso bem aplicado da metodologia. Uma das maneiras de mitigar esse possível problema está na aplicação de técnicas computacionais em algumas etapas do processo de análise de dados, para servirem de base para uma análise mais imparcial dos dados e também para permitir uma maior escalabilidade do processo, dado que a análise de várias entrevistas pode levar uma grande quantidade de tempo. A partir dessa adição de uma etapa quantitativa à metodologia, e combinando-a com a análise interpretativa dos dados, é possível gerar resultados mais objetivos, com maior agilidade e maior escalabilidade, dado que um mesmo algoritmo sempre dará resultados iguais, para uma mesma base de dados.

Neste artigo, aplicamos conceitos de *Machine Learning* em nosso conjunto de textos gerados na codificação aberta, com o objetivo de compreender com mais objetividade quais são os pontos de destaque na compreensão da relação entre a evolução do código com base na evolução da linguagem. Inicialmente, realizamos um pré-processamento dos dados, aplicando uma série de tratamentos em nossa base de dados para que esta fosse bem representada por um modelo estatístico, antes de ser alimentada pelo algoritmo de agrupamento aplicado. A rotina de pré-processamento que aplicamos foi divida em: 1)

Tokenização da base de dados: quebra do conjunto de respostas transcritas em palavras individuais; 2) Remoção de *stop words*: remoção de palavras comuns (a, são, o, etc) que não contribuem para o significado semântico do texto; 3) Remoção de ruído do texto: remoção de palavras com caracteres especiais, palavras com símbolos não ASCII. Em resumo, tudo que não dá para ser reconhecido como uma palavra.

Após a realização do pré-processamento dos dados, buscamos compreender e representar o quanto uma palavra é importante dentro de um conjunto de palavras, de modo a prepará-las para serem aplicadas em um algoritmo de classificação. Este processo de mapeamento de dados textuais é chamado de *feature extraction*, e ele gera como resultado o mapeamento dos dados textuais em vetores. Nós utilizamos dois tipos de algoritmo para a realização dessa tarefa. O primeiro foi um algoritmo de *Term Frequency-Inverse Document Frequency* (*TF-IDF*), ou Frequência de Termo-Frequência Inversa de Documento. Seu objetivo é gerar uma estatística numérica que busca refletir a importância de uma dada palavra em uma base de dados. Os termos com peso mais alto são considerados os mais relevantes dentro dessa base de dados. Explicando simplificadamente, esse algoritmo é capaz de realizar o aumento do peso de importância de uma palavra quando ela aparece muitas vezes em um documento, ao mesmo tempo que diminui o peso da mesma palavra quando ela é comum em muitos documentos. Dessa forma, é possível entender quais termos são muito importantes para um certo entrevistado, mas que não são exatamente um senso comum entre todos. O segundo algoritmo utilizado foi um algoritmo de contagem normalizada, que realiza uma contagem básica das palavras mais encontradas dentre todos os documentos, ao mesmo tempo que filtra certas palavras com uma frequência alta demais. Escolhemos utilizar esse algoritmo pois percebeu-se que em muitas ocasiões apareciam palavras como evolução, linguagem, código, e algumas outras palavras que não possuem muito valor em nossa análise.

Para realizar a clusterização dos nossos dados, foi utilizado o algoritmo *K-Means*, em razão de sua simplicidade de implementação e de seus bons resultados para bases de dados não muito grandes. O *K-means* é um algoritmo de aprendizado não supervisionado, dado que não precisa de rotulação prévia antes de ser aplicado. É também um algoritmo de *clustering* (aglomeração, agrupamento) que separa o conjunto de dados em um número K pré definido de *clusters* (grupos). Essa divisão em K grupos é realizada a partir de K centróides, sendo que cada centróide é um ponto que representa o centro de um grupo. Este algoritmo funciona de forma iterativa, de modo que os centróides são arbitrariamente alocados em um espaço vetorial do conjunto de dados e movem-se para o centro dos pontos mais próximos deles. A cada nova iteração, recalcula-se a distância entre os centróides e os pontos e eles se movem novamente para o centro dos seus pontos mais próximos. Quando a posição ou os grupos não são mais alterados, o algoritmo alcança seu fim. O

algoritmo também é finalizado quando a distância da mudança dos centróides atinge um certo limiar pré-definido.

É necessário também escolher o melhor número de *clusters* para um melhor resultado. Escolhemos aplicar a média *Silhouette* para definir o melhor valor. Dessa forma, foi possível visualizar o grupamento a partir dos gráficos das palavras mais dominantes em cada grupo. Depois desse processo, tornam-se claras as categorias principais que as palavras inferem em cada grupo, o que pode ser de extrema importância para encontrar a categoria principal e categorias adjacentes na GT. A partir desse processamento qualitativo, tornou-se mais fácil observar as categorias que poderiam emergir do processo de codificação aberta, além do fato que uma vez implementada a lógica, torna-se possível escalar o seu uso para as outras perguntas e outras etapas da metodologia, como a codificação seletiva.

Na próxima seção são apresentados alguns dos nossos resultados para a etapa de análise de dados, tanto qualitativa quanto quantitativa, com base no que fomos coletando nas entrevistas. Seguimos o processo tradicional proposto pela GT, começando-o pela codificação aberta, aplicando o método de comparação constante. A continuação da GT seria o emergir de uma categoria principal do processo de codificação aberta, mas no presente artigo realizamos apenas a codificação aberta, deixando as etapas posteriores para um futuro artigo. O processo da codificação aberta e seus resultados estão descritos no Capítulo 4.

Capítulo 4

Resultados

4.1 Codificação Aberta - Resultados Quantitativos Preliminares

Com o objetivo de adaptar a metodologia de *grounded theory* para a realidade da Engenharia de software e para os contextos mais atuais de pesquisa, é possível evoluir o método de modo a tornar os seus resultados mais objetivos e replicáveis. Isso pode ser alcançado a partir de uma complementação computacional ao método, possibilitando uma análise dos dados mais imparcial e objetiva, alcançando assim uma extração mais eficaz dos pontos mais comentados nas entrevistas.

Apresentamos aqui os resultados da análise quantitativa da pergunta Q7: "Você considera a evolução da linguagem um fator importante para evoluir seu código? Por que?". Um processo análogo também foi realizado para a pergunta Q5. A partir dos textos transcritos, realizamos um pré-processamento dos dados, de modo a retirar as palavras que não possuem relevância para nossas análises. Ele seguiu as seguintes etapas:

(1) *Tokenização* da base de dados: quebra do conjunto de respostas transcritas em palavras individuais;

(2) Remoção de *stop words*: remoção de palavras comuns (a, são, o, etc) que não contribuem para o significado semântico do texto;

(3) Remoção de ruído do texto: remoção de palavras com caracteres especiais, palavras com símbolos não ASCII. Em resumo, tudo que não dá para ser reconhecido como uma palavra.

Para a remoção de *stop words*, foi utilizado um arquivo com várias palavras brasileiras que não possuem muito valor semântico, além de terem sido removidas também todas

as palavras com menos de dois caracteres. Além disso, foram removidas as vírgulas, as quebras de linha e por fim todos os caracteres foram passados para *lowercase*. Na Figura 4.1 vemos um exemplo de como estava um texto antes da etapa de pré-processamento, e na Figura 4.2 um exemplo do texto após o pré-processamento.

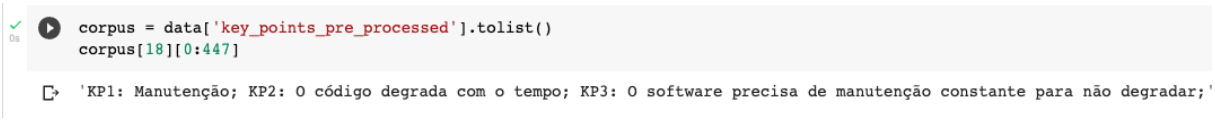

Figura 4.1: Resposta não pré-processada.

Figura 4.2: Resposta pré-processada.

Depois da etapa de pré-processamento, aplicamos a *feature extraction*, ou extração de funcionalidades, com o uso do algoritmo de contagem normalizada. O resultado dessa etapa é um vetor da contagem das palavras de todas as transcrições, como podemos observar na Figura 4.3.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
dados	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
manutenção	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	2	0	1	1
proteção	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
segurança	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0
acompanhada	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 4.3: Exemplo de vetor de palavras.

Em seguida, aplicamos o K-means. Foram feitos vários testes, e percebemos que com o valor 5 para o K e 4 para o número de grupos, o algoritmo nos trouxe resultados melhores. As imagens para a pergunta Q7 podem ser vistas nas Figura 4.4, 4.5, 4.6, 4.7.

A partir das imagens, podemos observar que o primeiro grupo está muito ligado a motivos de mercado para a evolução do software. Questões como suporte, performance, facilidades oriundas das novas funcionalidades da linguagem, desempenho, segurança, todos têm muito a ver com a forma que o software deve evoluir para se adaptar cada vez melhor às necessidades do mercado, e por isso tem o mesmo peso.

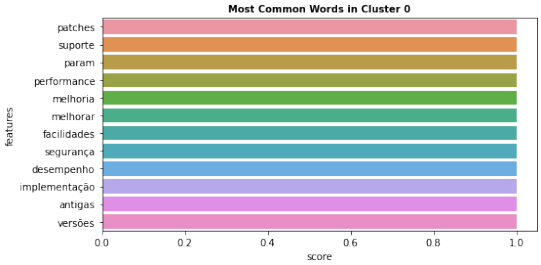

Figura 4.4: Primeiro grupo para a Q7

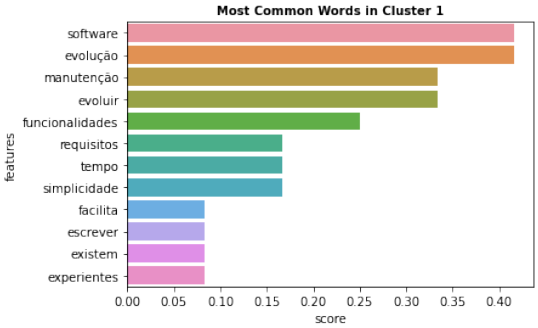

Figura 4.5: Segundo grupo para a Q7

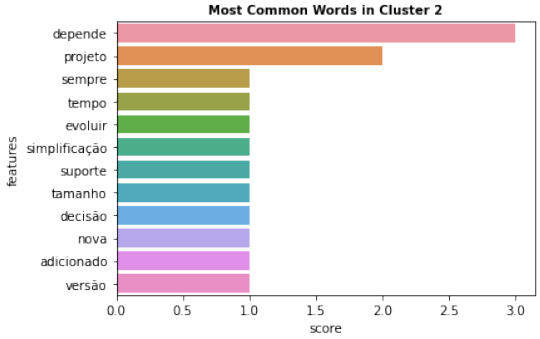

Figura 4.6: Terceiro grupo para a Q7

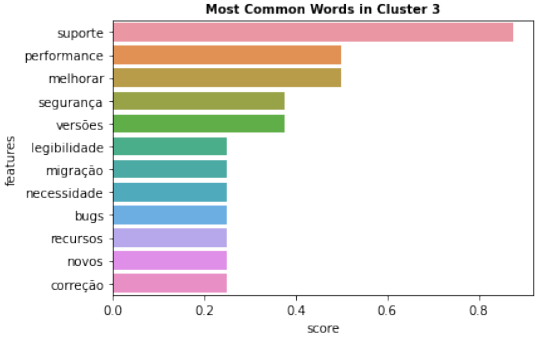

Figura 4.7: Quarto grupo para a Q7

No segundo grupo, vemos que o centro está nas palavras software, evolução e evolução. Pelo resultado, vemos que normalmente a evolução do software está bastante ligada à manutenção do software, e também que as funcionalidades que são adicionadas na evolução da linguagem tem um grande peso na decisão de evoluir ou não a linguagem.

No terceiro grupo vemos que o centróide está na palavra depende. Em algumas entrevistas, foi comum a resposta que a evolução depende de vários fatores. É uma questão que deve ser bem avaliada, com base no tempo do projeto, em seu tamanho, se há suporte à linguagem atual, entre outros vários fatores. A decisão de evoluir o software acaba dependendo de vários fatores complexos que devem ser analisados, na maioria dos casos.

Por fim, no último grupo, vemos que o ponto principal é o suporte. Normalmente as respostas relacionadas ao suporte incluem questões como migração de versões, simplicidade, versões da linguagem, dado que quando a versão da linguagem está defasada e deixa de ter um bom suporte, deixa-se de ter acesso a vários recursos da linguagem, a legibilidade é afetada pois não há muita documentação para versões não mais suportadas, os *bugs* passam a ser mais dificilmente corrigidos se o programador não tiver muita experiência, e funcionalidades que poderiam ser feitas com muita simplicidade podem acabar ficando complexas.

Tendo como base essa análise quantitativa, tornou-se possível encontrar bons pontos

Figura 4.8: Primeira nuvem de palavras para a Q7

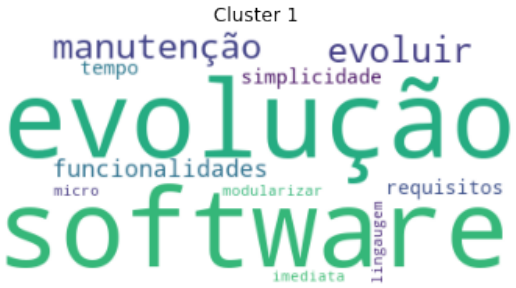

Figura 4.9: Segunda nuvem de palavras para a Q7

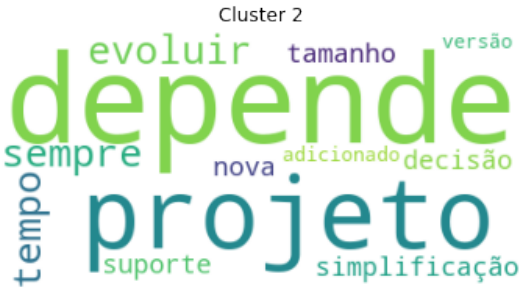

Figura 4.10: Terceira nuvem de palavras para a Q7

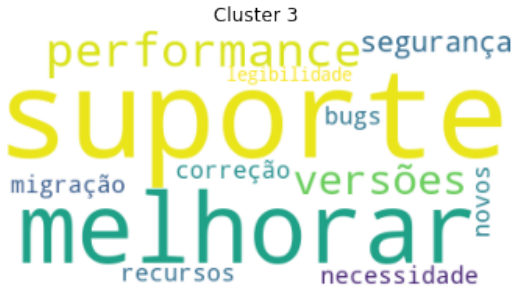

Figura 4.11: Quarta nuvem de palavras para a Q7

de partida para encontrar as principais categorias de nossa base de dados de maneira escalável. Podemos ver que para evoluir o código, existem vários fatores que devem ser considerados. É uma decisão que depende de aspectos como o tamanho do projeto, o tempo do projeto, se a linguagem ainda possui um bom suporte, se as funcionalidades novas geram algum impacto direto de performance, se a evolução proporcionará melhorias de desempenho, segurança, legibilidade. As Figuras 4.8, 4.9, 4.10, 4.11 nos permitem uma melhor visualização do peso que cada palavra assume neste contexto da evolução do software. Elas podem servir como base de interpretação para o emergir de boas categorias, a partir dos *key points* definidos. A seguir, partimos para os resultados qualitativos, que evidenciam vários dos pontos encontrados em nossa análise quantitativa.

4.2 Codificação Aberta - Resultados Qualitativos Preliminares

A codificação aberta é a primeira etapa da análise de dados, sendo normalmente aplicada com uma análise detalhada inicial das entrevistas transcritas. Seu processo consiste em coletar os pontos principais de cada entrevista, para posteriormente sintetizá-los em có-

Tabela 4.1: Classificação dos Entrevistados.

Entrevista	Experiência	Organização	Papel
E1	10 a 15 anos	Iniciativa Privada	Desenvolvedor
E2	15 a 20 anos	Iniciativa Privada	Desenvolvedor
E3	15 a 20 anos	Iniciativa Privada	Líder Técnico
E4	5 a 10 anos	Iniciativa Privada	Desenvolvedor
E5	Menos de 5 anos	Iniciativa Pública	Desenvolvedor
E6	15 a 20 anos	Iniciativa Pública	Desenvolvedor
E7	10 a 15 anos	Iniciativa Privada	Líder Técnico e desenvolvedor
E8	5 a 10 anos	Iniciativa Privada	Desenvolvedor
E9	5 a 10 anos	Iniciativa Privada	Desenvolvedor
E10	5 a 10 anos	Iniciativa Privada	Desenvolvedor
E11	Mais de 20 anos	Iniciativa Privada	Desenvolvedor
E12	15 a 20 anos	Iniciativa Privada	Desenvolvedor
E13	5 a 10 anos	Iniciativa Privada	Desenvolvedor
E14	Menos de 5 anos	Iniciativa Privada	Desenvolvedor
E15	10 a 15 anos	Iniciativa Privada	Líder Técnico e desenvolvedor
E16	Mais de 20 anos	Iniciativa Privada	Líder Técnico
E17	10 a 15 anos	Iniciativa Privada	Líder Técnico e Mentor
E18	5 a 10 anos	Iniciativa Pública	Desenvolvedor
E19	5 a 10 anos	Iniciativa Pública	Desenvolvedor
E20	Mais de 20 anos	Iniciativa Privada	Desenvolvedor
E21	10 a 15 anos	Iniciativa Pública	Desenvolvedor
E22	Mais de 20 anos	Indústria e Pesquisa	Desenvolvedor e Professor
E23	5 a 10 anos	Indústria e Pesquisa	Desenvolvedor

digos, que são frases curtas que contém essa síntese alcançada. Para compreender melhor os resultados alcançados na etapa de codificação, é interessante compreender o perfil de nossos entrevistados. Como mencionado anteriormente, iremos utilizar as citações de nossos entrevistados para evidenciar os pontos mais relevantes encontrados para cada uma das perguntas analisadas nesta dissertação. A Tabela 4.1 mostra a classificação de cada entrevista, e cada citação utiliza como referência algum trecho das entrevistas realizadas com algum de nossos participantes.

4.2.1 (Q5) Você considera a evolução da linguagem um fator importante para evoluir seu código? Por que?

A partir da análise qualitativa realizada com os 23 participantes, foi possível observar que a grande maioria dos entrevistados considera a evolução da linguagem um fator importante para a evolução do código. Dos 23 entrevistados, apenas dois responderam que não consideram a evolução da linguagem como um fator relevante, enquanto dois responderam

de maneira neutra. Dos que responderam positivamente, é perceptível que grande parte dos entrevistados compreendem que as linguagens evoluem sempre na busca de propor melhorias. Este ponto é levantado por vários dos entrevistados, e podemos ver alguns exemplos a seguir:

> E1: "[...] as linguagens, no geral, evoluem para tornar algumas coisas mais fáceis."

> E3: "[...] a cada versão, a cada / o que a gente chama de *major version* / a cada atualização principal da versão, tanto da ferramenta que te / da plataforma, da linguagem de programação, quanto dos *frameworks*, ele sempre traz várias atualizações que facilitam muito a vida do desenvolvedor [...]"

O que em um momento era feito de maneira mais verbosa e com maior complexidade de código passa a ser mais simples, tanto em termos de escrita do código quanto em termos de performance. Esse é, então, um dos motivos pelo qual os entrevistados veem que vale a pena evoluir o código de acordo com a evolução da linguagem. Outro ponto levantado nas entrevistas está relacionado com as novas funcionalidades que são adicionadas nas novas versões das linguagens. Quando são introduzidas funcionalidades que podem afetar diretamente e positivamente a parte principal de seu produto, é comum que valha a pena acompanhar a evolução do software com a da linguagem, pois isso refletirá diretamente nos resultados da empresa. Porém, quando são introduzidas funcionalidades que não geram um verdadeiro impacto no software, é comum que não seja gerado um esforço para a evolução do código.

Uma questão que foi unânime foi a importância de simplificar constantemente as *code bases*, e a importância de deixar o código cada vez mais limpo, legível e simples, não só para você, mas também para todas as pessoas que trabalham e trabalharão no código desenvolvido. Como exemplo, podemos ver o que foi comentado na entrevista 10:

> E10: "[...] A evolução das linguagens de programação, de uma forma geral, acredito que contribuem para a evolução do meu código por conta das *features* que foram adicionadas. Em algumas situações, as linguagens tendem a fazer a simplificação... tornar o trabalho um pouco mais simples e mais fácil para outros seres humanos lerem. [...]"

Algo interessante que foi comentado em algumas entrevistas está relacionado ao fato de em muitos momentos a evolução da linguagem pedir um período de adequação por parte do desenvolvedor. Há uma curva de conhecimento e aprendizado para novas funcionalidades de uma linguagem, e o processo de evoluir o código com base na evolução da linguagem dificilmente será automático. Para refatorar o código de uma funcionalidade, há que compreendê-lo novamente, para que seja feita uma revalidação da implementação, e esse

processo também possui um custo. Logo, não é um processo que se faz da noite para o dia, mas sim de maneira iterativa e de pouco em pouco. Nas entrevistas 4 e 9 e 23 comenta-se sobre esse assunto:

> E4: "[...] quando você tá trabalhando um sistema que já é bastante antigo, normalmente a gente evita fazer *refactoring* de *features* que já estão em produção, a menos que seja estritamente necessário; porque senão você vai ter que passar por todo um processo de novo de revalidar aquela implementação. [...]"

> E9: "[...] Principalmente nos projetos internos da minha empresa, a questão de motivar funcionalidades novas: não é imediatamente que essas funcionalidades são lançadas, mas conforme a gente tem um entendimento da versão, conforme a gente entende o uso daquela nova funcionalidade, a gente acaba utilizando nos nossos projetos. Então leva-se um pouquinho de tempo para aprender sobre essa nova funcionalidade e para o que ela serve. Geralmente está relacionado à demanda, se a gente precisar daquela funcionalidade depois de ter entendido para que ela serve, a gente utiliza sem problema. [...]"

> E23: "[...] Our code base has more than two million lines of code. Modifying that as a whole would not be a great idea and would require a lot of time. I often choose the method, or a class, if I have the time, and modify that, and of course, if I modify a method that is very similar to another method, I often modify that too, even if I don't have much time for finishing a feature request or something like that. Because they are very similar, I don't want those codes to differ that much. So, I try to enhance that method. So, even if this has nothing to do with what I'm working on. [...]"

Muitos dos entrevistados comentaram que a evolução deve ser feita com muita cautela e atenção, por conta das incompatibilidades de versões. Em linguagens como Java, C, e outras, é muito comum que quando ocorre uma mudança de *Major Versions*, haja muitas incompatibilidades. Por conta disso, a evolução do código nesses casos acaba sendo muito custosa, e em muitos casos acaba nem sendo feita por conta desse alto custo. Na entrevista 3 comenta-se sobre este ponto:

> E3: "[...] a cada versão, quando muda o primeiro número da versão, tem também vários problemas de compatibilidade com a versão anterior, então esse é um processo que tem que ser bem pensado dentro da evolução dos sistemas. [...]"

Por fim, foi comentado por alguns entrevistados que quando há um processo inteligente de evolução de código com base na evolução das linguagens, é natural que os projetos como um todo acompanhem essa evolução, o que leva os desenvolvedores a serem capazes de implementar funcionalidades muito mais complexas com mais eficácia. Como exemplo, a entrevista 9 retrata um pouco essa ideia:

> E9: "[...] principalmente pensando nos projetos internos da empresa, conforme a linguagem foi evoluindo, os nossos projetos evoluíram junto, a gente conseguia fazer coisas bem mais complexas conforme essas funcionalidades foram aparecendo."

Como uma síntese dos aprendizados alcançados pela análise dessa pergunta, podemos dizer que grande parte dos entrevistados veem a evolução da linguagem como um fator muito importante para a evolução do código, por vários motivos. A evolução da linguagem existe para ajudar o desenvolvedor a realizar seu trabalho de maneira mais inteligente e eficaz, e quando aplicada a um projeto, ajuda-os a simplificar o código para que outros desenvolvedores possam também compreendê-lo. Ao mesmo tempo, possibilita a evolução da forma de implementação de várias funcionalidades do código - um código que era feito antes de forma muito complexa, passa a ser implementado de forma mais simples, otimizada e legível. Ficou evidente também que esse processo de evolução não é de graça, pois existe um custo de tempo para o aprendizado das novas funcionalidades, e também existe a possibilidade da utilização incorreta das novas funcionalidades trazidas pela linguagem. Como grande parte dos entrevistados possui muitos anos de experiência de desenvolvimento, não foi muito comum encontrar essas barreiras de aprendizados por parte deles. Foi possível compreender que quando as novas funcionalidades da linguagem são bem utilizadas, a manutenção do código se torna muito mais barata e simples, gerando como consequência a otimização do tempo de trabalho dos desenvolvedores e uma maior adaptabilidade do próprio código a posteriores mudanças.

4.2.2 (Q7) Quais motivos te levam a evoluir o código para suportar novos recursos da linguagem?

Um dos pontos importantes observados nessa questão foi que em muitos momentos não está na mão dos desenvolvedores a decisão da evolução da linguagem. Para empresas maiores, essa acaba sendo uma decisão de grande impacto, por inúmeros motivos. Essa decisão deve ser tomada pensando no número de funcionários, se será necessário um treinamento na nova versão da linguagem, os custos para a adequação à nova versão, o valor que a mudança pode gerar em termos do desenvolvimento do produto trabalhado e outros vários motivos. Assim, acaba sendo uma decisão que depende bastante da gerência das empresas, principalmente aquelas que possuem um modelo organizacional mais tradicional. Nas entrevistas 1 e 5, o entrevistado comenta sobre esse ponto:

> E1: "[...] geralmente, uma experiência que eu tenho, essa evolução é feita em termos corporativos, então, a corporação decide que / o grupo, a equipe decide que é melhor a gente evoluir conforme a linguagem vai evoluindo, e aí é feita essa autorização, digamos assim, esse acordo para evoluir os sistemas para aquela versão nova da linguagem. [...]"

> E5: "[...] Em minha opinião, isso vai depender de alguns fatores. Um deles é se você pode alterar esse código. No caso do principal projeto aqui da empresa, infelizmente é inviável que isso aconteça. [...]"

Sob um ponto de vista dos desenvolvedores, seja quando podem tomar a decisão de evoluir a linguagem ou quando devem ajudar na decisão a nível corporativo, existem alguns fatores que apareceram com muita frequência como pontos relevantes para essa questão. Um deles, que não era esperado pela nossa equipe, foi a questão de segurança. Quando um código torna-se legado, passa a gerar custos cada vez maiores para ser mantido, ao mesmo tempo que se torna muito difícil de ser compreendido. Por conta disso, fica passível de ser removido ou modificado de maneira incorreta, o que pode gerar problemas irreversíveis. Pensando ainda pelo ponto de vista da segurança, novas versões em muitos casos trazem atualizações de segurança contra novos tipos de ataque que vão surgindo, além de questões de perda de memória, o que pode ser um motivador importante para a evolução do código em alguns casos de sistemas críticos.

> E5: "[...] Então é superimportante você estar evoluindo sua base de código à medida que as linguagens evoluem porque você vai ter novos recursos, ganhos de performance, correções de bug de segurança e evita o trauma da migração. [...]"

> E21: "[...] Proteção de dados, segurança. Você tinha diversas construções que eram feitas anteriormente em cima de vetores que, em algum momento, corrompia a memória. E a partir daí, era uma bomba relógio, em algum momento vai explodir. Só que se você começa a subir dados absurdos, começa vir reclamação de cima - o pessoal reclamando que estava acontecendo coisas que não deveriam acontecer a partir de um vazamento de memória. Fora isso, tem um problema de concorrência, que tudo acessa tudo e você não tem garantia [...]. Um dado esperando outro, outro esperando outro, dando problema de condição de corrida e a aplicação ficando travada. [...]"

Outro motivo muito comentado que leva à evolução do código é a melhora de desempenho e performance. É muito comum que a evolução da linguagem esteja ligada à evolução de todo seu ecossistema. A melhoria de performance em muitos casos é muito interessante comercialmente, o que leva as grandes empresas a estarem constantemente evoluindo as linguagens e a forma como utilizam as linguagens, justamente para encontrar formas mais otimizadas de fazer o que já fazem e melhorar os seus produtos. O entrevistado de número 23 comentou algo interessante sobre este ponto:

> E22: "[...] The third thing is, that if you have something with an extraordinary performance gate, performance is still important in certain projects. You know, if you have a Java project, or web application or database application performance, it is not that important, but if you are in investment banking and you are an automatic trading system, then 2 percent speedup can mean billions of dollars a year. Picture

> that. Or you just think about the Mars Rover, which was programmed in C++, 2 percent improvement in efficiency might mean that the solar energy is just enough for another two months. So, C++ is a specific language, which is mostly used when performance issues are coming into the discussion. [...]"

Um outro motivo muito comentado pelos desenvolvedores, que acaba sendo uma consequência dos outros motivos, é o aumento da manutenibilidade do código. As linguagens evoluem para facilitar a vida do desenvolvedor em vários aspectos. A evolução da linguagem impacta diretamente na limpeza do código, em sua legibilidade, dado que com o tempo, o código naturalmente vai se degradando. Quando o código está mais legível e mais limpo, os desenvolvedores demoram menos tempo para desenvolver novas funcionalidades e consequentemente diminui-se drasticamente o custo de aprimoramento do produto desenvolvido. As empresas maiores estão cada vez mais percebendo essa realidade e investindo seus esforços para manter toda sua base de código o mais manutenível possível, mais adaptável às constantes mudanças do próprio mercado. Os entrevistados 16 e 19 comentam sobre essa questão:

> E16: "[...] antigamente você ia fazer os *getters* e *setters* e aí você tinha que fazer na mão, conjunto, depois tinha um pouco de apoio de alguns *IDEs* que geravam código, mas você tinha que desenhar, escrever todos os *sets* e *gets*. Hoje em dia você põe uma anotação daquela lá, enfim, ele já faz tudo, tem muita coisa que diminui e fica um código mais legível, fica um código em outro nível, fica um código mais limpo, com menos código para se perder, manutenção mais fácil. Tem muita coisa, o Lambda em si ele facilita muita coisa, [...]"

> E19: "[...] a modernização é um processo de evolução. No final das contas, é para que novos desenvolvedores que irão ler o código numa versão mais recente da linguagem estejam mais confortáveis a dar manutenção naquele código. E outra coisa, tem a questão da degradação também; o código, com o tempo, degrada-se e essa degradação, com as novas versões, testes unitários e coberturas, ele deixa de se degradar. [...]"

Um ponto muito comentado foi que em um sistema real, essa decisão de evoluir o código é algo muito sério. Há que haver um motivo muito claro para que seja realizada a evolução da linguagem no sistema, e normalmente envolverá questões financeiras. Um bom exemplo para isso pode ser visto na entrevista 12:

> E12: "[...] Existem lugares que são o oposto, que querem sempre evoluir e também tem lugares que avaliam o impacto disso. Então de uma forma geral, eu acho que a postura mais correta que eu considero é: até que ponto vale mexer em um projeto para evoluir a linguagem para atingir um novo recurso, ou uma nova funcionalidade? O impacto no projeto deve ser levado em consideração. Vamos dizer assim: se essa modificação tem a chance de deixar o negócio extremamente instável ou falho, ou por algum motivo provocar uma instabilidade em funcionamento, que é

> *runtime* ou em compilação, essa modificação é uma coisa que possivelmente vai ser recusada. Não dá para falar assim: vou atualizar porque a versão está mais nova. Não, tem que ver o impacto que vai causar no projeto, entendeu? [...]"

Um ponto muito interessante que foi levantado foi que com a migração de paradigma de arquitetura monolítica para micro-serviços, torna-se muito mais fácil manter a evolução do código a par com a evolução da linguagem. Tendo serviços cada vez menores, e com equipes com cada vez mais autonomia, os softwares passam a ser cada vez mais adaptáveis às mudanças. Dessa forma, tornam-se muito menos suscetíveis a virarem sistemas legado, tendo custos cada vez menores de manutenção e de desenvolvimento.

> E4: "[...] hoje em dia a gente consegue quebrar o software em serviços, certo? Então o software não é só aquela *stack* enorme, que você antes... pra mudar era um trabalho enorme. Agora, você quebrando em micro-serviços ajuda bastante a manter e a evoluir o software, certo? Porque você vai conseguindo só fazer esse tipo de migração, adicionando novas *features*, aonde realmente pode, e isolando o problema. Então você não precisa fazer como um todo [...]"

Por fim, sintetizando os conhecimentos trazidos à tona a partir da pergunta Q7, vemos que as organizações adotam diferentes posturas com relação à evolução do código. Existem organizações que estão sempre em busca de evoluir seu código, algumas outras que raramente veem o valor nessa evolução e organizações que decidem por evoluir o código apenas quando há um valor muito claro na mudança proposta. Vemos que a decisão de evoluir o código perpassa por diversos personagens dentro das empresas e organizações, e que cada organização adota uma postura diferente quanto a essa questão. Existem organizações que permitem que a equipe de tecnologia escolha o melhor momento para a evolução do código; em outras organizações, a decisão fica a cargo da diretoria; e em outros casos, é uma decisão tomada em conjunto por todos os envolvidos no processo.

É visível que muitos são os motivadores para a evolução do código, e que quanto mais demorada é a decisão de evolução, mais custoso será o processo. Dentre os principais motivadores, vemos questões de performance, manutenibilidade, legibilidade e simplicidade do código, que são questões que afetam diretamente os desenvolvedores, mas que para quem não compreende muito o dia a dia de desenvolvimento do software podem parecer questões não muito relevantes para o produto. Para os desenvolvedores, e também para quem analisa o produto de uma forma mais ampla, é evidente que quanto mais rápido e simples for o processo de implementação ou correção de uma funcionalidade do sistema, mais eficaz é o processo de evolução e crescimento de um produto ou serviço. Outros motivadores muito comentados para a evolução do código foram as questões de desempenho, performance e segurança que são muito mais tangíveis para os gerentes e administradores de um produto.

A partir dessa análise, torna-se evidente que a decisão de evoluir ou não um software não é algo trivial, e que de fato envolve muitas questões complexas de mercado e de equipe, necessitando assim de uma compreensão social de todos os envolvidos no processo para que a melhor decisão seja tomada.

Capítulo 5

Conclusão

Neste artigo, foram apresentados os resultados de uma aplicação parcial da *grounded theory* em cima das perguntas "Quais motivos te levam a evoluir o código para suportar novos recursos da linguagem?"e "Você considera a evolução da linguagem um fator importante para evoluir seu código? Por que?". Foram gerados resultados tanto quantitativos, por meio de algoritmos de processamento de linguagem natural, quanto qualitativos. Ambos os resultados nos deram bons direcionamentos para as futuras etapas da *grounded theory*, posteriores à codificação aberta.

A *Grounded Theory* é uma metodologia científica muito útil e eficaz quando o objetivo é compreender uma área de interesse por um ponto de vista mais abrangente e completo. Pelo fato de contemplar os aspectos sociais e humanos em seu método, busca as causas de uma questão de pesquisa de uma maneira fundamentada e a partir das experiências reais de quem vive o contexto estudado diariamente. A partir de nossas experiências, foi possível compreender que a análise qualitativa com o uso de algoritmos de processamento de linguagem natural serve como um bom suporte e acréscimo à metodologia *grounded theory*. Essa evolução acrescenta objetividade às análises e permite em um bom grau diminuir as subjetividades oriundas das análises estritamente qualitativas.

Como resultado para ambas as análises, vimos que a evolução da linguagem é um fator muito importante como motivador para a evolução do software. Os principais motivos são: desempenho, performance, legibilidade de código, facilidade de manutenção, possibilidade de suporte, correções de *bugs* e melhorias em termos de segurança. Descobrimos que as diferentes empresas têm diferentes posições sobre o processo de decisão de atualizar o software: algumas estão sempre atualizando o software com base na atualização da linguagem; outras atualizam apenas quando há um motivo muito relevante para a mudança; e algumas evoluem apenas quando há uma grande urgência. É evidente, sob o ponto de vista dos desenvolvedores, que quanto maior o tamanho do projeto e maior o tempo desde a última atualização, maior será o custo da evolução do código.

Ficou claro que para os desenvolvedores que atuam em iniciativas privadas, a decisão sobre a evolução do software passa por um fluxo de compreensão das necessidades dos usuários, para que então sejam buscadas soluções no software que podem gerar melhorias para o produto. Para os desenvolvedores e os gestores da área de tecnologia, e também para quem analisa o produto de uma forma mais ampla, é evidente que quanto mais rápido e simples for o processo de implementação ou correção de uma funcionalidade do sistema, mais eficaz é o processo de evolução e crescimento de um produto ou serviço.

5.1 Trabalhos Futuros

Como comentado em seções anteriores, o presente trabalho apresenta a aplicação da metodologia de *grounded theory* até o ponto da codificação aberta. Planejamos a continuação deste trabalho, de modo a englobar as próximas etapas para a aplicação completa da *grounded theory*, chegando por fim em uma teoria bem fundada para evidenciar as principais relações entre a evolução do código e a evolução das linguagens de programação.

Um segundo aprimoramento à nossa maneira de aplicação da metodologia surge da necessidade de torná-la mais escalável e menos suscetível a subjetividades. A aplicação da *Computational Grounded Theory* na etapa de análise de dados é uma possível solução que pode possibilitar essa evolução ainda maior do método [15], permitindo uma análise mais ágil, objetiva e escalável dos dados gerados a partir das entrevistas, para além de uma aplicação simples de processamento de linguagem natural. O processo se aplicaria não apenas à etapa de codificação aberta, mas à etapa de análise de dados como um todo.

Referências

[1] Godfrey, M. W. e D. M. German: *The past, present, and future of software evolution.* Frontiers of Software Maintenance, página 129–138, 2008. 1, 4

[2] Lehman, M. M. e J. F. Ramil: *Rules and tools for software evolution planning and management.* Annals of software engineering, página 11(1):15–44, 2001. 1, 4

[3] Williams, Courtney, Margaret-Anne D. Storey andNeil A. Ernst, Alexey Zagalsky e Eirini Kalliamvakou: *The who, what, how of software engineering research: a socio-technical framework.* Empirical Software Engineering, 25, 2020. 1

[4] Hoda, Rashina: *Socio-technical grounded theory for software engineering.* IEEE Transactions on Software Engineering, 2021. 1, 2

[5] Overbey, Jeffrey L. e Ralph E. Johnson.: *Regrowing a language: Refactoring tools allow programming languages to evolve.* In Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '09, página 493–502, 2009. 3, 4

[6] Lehman, Meir e Juan Fernandez-Ramil: *Software evolution and software evolution processes.* Annals of Software Engineering 14, página 275–309, 2002. 3

[7] Chapin, Ned, Joanne Hale, Khaled Khan, Juan Fernandez-Ramil e Wui Gee Tan: *Types of software evolution and software maintenance.* Journal of Software Maintenance, 13:3–30, janeiro 2001. 3

[8] Dig, Danny e Ralph Johnson: *How do apis evolve? a story of refactoring.* Journal of Software Maintenance, 18:83–107, março 2006. 3

[9] Urma, Raoul Gabriel: *Programming language evolution.* 2017. 4

[10] Dantas, Reno, Antônio Carvalho, Diego Marcílio, Luísa Fantin, Uriel Silva, Walter Lucas e Rodrigo Bonifácio: *Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate java programs.* Em *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, páginas 497–501, 2018. 4

[11] Hoda, Rashina, James Noble e Stuart Marshall: *Developing a grounded theory to explain the practices of self-organizing agile teams.* Empirical Software Engineering - ESE, 17:1–31, dezembro 2012. 5, 7, 10

[12] Gobinda, Chowdhury: *Natural language processing.* Annu. Rev. Inf. Sci. Technol. 37(1), página 51–89, 2005. 8

[13] Chen, Nan Chen, Margaret Drouhard, Rafal Kocielnik, Jina Suh e Cecilia Aragon: *Using machine learning to support qualitative coding in social science: Shifting the focus to ambiguity.* ACM Transactions on Interactive Intelligent Systems, 8:1–20, junho 2018. 8

[14] Strandberg, Per Erik: *Ethical interviews in software engineering.* Em *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, páginas 1–11, 2019. 11

[15] Nelson, Laura: *Computational grounded theory: A methodological framework.* 49:004912411772970, novembro 2017. 30