Investigando Comunicabilidade e Testabilidade com a ferramenta Signifying APIs

Camila Pardo Garcia Morelli
Universidade Federal Fluminense
camilamorelli@id.uff.br

Vânia de Oliveira Neves
Universidade Federal Fluminense
vania@ic.uff.br

Luciana Salgado
Universidade Federal Fluminense
luciana@ic.uff.br

ABSTRACT

Given the diversity of information systems today, communication between services requires that APIs be well designed and understood by both services producers and consumers. Poorly documented APIs lead to misunderstandings by developers and testers teams who end up designing ineffective test cases. As a result, they may produce software with low quality and avoidable errors. This study investigates the ability of the *SigniFYIng APIs* tool to support the testability of the applications consuming APIs. In this paper, we proposed a process to support APIs' testability with the *SigniFYIng APIs* tool. We validated the process with a real case study based on two Brazilian federal government APIs: the leniency agreement API and the federal servants API. As a result, it was possible to develop better test cases for the chosen APIs, bringing evidence that the proposed process can support designing more suitable test cases for APIs and improving the testability of the software to be produced.

KEYWORDS

APIs, Testabilidade, Comunicabilidade, Teste, SigniFYIng APIs

1 INTRODUÇÃO

O aumento das expectativas no desenvolvimento de software em diversas esferas da sociedade, como em serviços de sistemas e-gov e na criação de cidades inteligentes, bem como o aumento da complexidade desses sistemas, exige cada vez mais que eles tenham a capacidade de se comunicar entre si. Geralmente, essa comunicação se dá por meio de componentes reutilizáveis chamados de serviços [1]. Além disso, essa complexidade muitas vezes é causada pela diversidade de sistemas de informação e pela quantidade de interações entre eles. Sendo assim, esses sistemas, que podem ter sido desenvolvidos por diferentes organizações, necessitam possuir alta interoperabilidade, ou seja, necessitam que haja uma comunicação transparente e eficiente.

As integrações de serviços são realizadas via APIs ou "Interface de Programação de Aplicativos". Em resumo, uma API é uma interface fornecida por terceiros na qual possibilita a equipe de software a utilizá-la e consumir um determinado tipo de serviço sem se preocupar com a implementação do mesmo. Dessa forma, uma API representa abstrações criadas por seus projetistas que precisam ser entendidas pelos seus usuários (programadores, testadores) para ser usada de maneira eficaz. Assim, podemos considerar como um processo de comunicação entre essas duas partes, mediado pelos artefatos de software envolvidos (especificações, documentação, código, binários, mensagens, etc.). Mais precisamente, esses artefatos comunicam os programadores como eles devem interagir com a API [2].

Uma análise realizada em vários projetos no GitHub mostrou que 93,3% dos projetos utilizam APIs [3]. Dada a ubiquidade desses mecanismos, é importante que haja soluções e métodos sólidos de desenvolvimento que garantam a qualidade do software que os consomem. No entanto, manter a qualidade de uma aplicação que utiliza um serviço torna-se uma tarefa não tão trivial. Para que a equipe de qualidade possa determinar se o sistema é adequado ou não, ela deve conferir questões como se os padrões de documentação foram seguidos durante o desenvolvimento, ou se o software foi testado adequadamente [4]. Dessa forma, para a API ser efetiva, ela deve ser compreensível e utilizada da forma correta pela equipe de qualidade e, sendo assim, a qualidade da documentação se torna um fator crítico [5]. Um mal entendimento do uso da API pode ocasionar falhas na aplicação consumidora.

Teste de software é uma das principais técnicas para se garantir a qualidade, mas para que ela possa ser conduzida efetivamente, possibilitando a criação de bons casos de testes, mais uma vez é essencial ter uma boa documentação. O grau em que um testador consegue projetar casos de teste está relacionado ao conceito de testabilidade. Ou seja, segundo [6], testabilidade é definida como:

> *Testabilidade.* O grau em que um sistema, ou componente, facilita o estabelecimento de critérios de teste e a execução deles para determinar se esses critérios foram atendidos; e o grau em que um requisito é declarado, em termos que permitem o estabelecimento de critérios de teste e o desempenho deles para determinar se esses critérios foram atendidos.

De acordo com [7], quando um serviço apresenta uma boa testabilidade a qualidade de serviço aumenta e consequentemente reduz o custo de teste. Como visto em [5], a testabilidade não apresenta uma forma clara de definir quais aspectos do software estão realmente relacionados a ela, o que sugere a busca de meios e técnicas para ajudar a melhorar a testabilidade em um software.

Diante dessa lacuna, o principal objetivo deste artigo é propor o uso da ferramenta conceitual *SigniFYIng APIs* chamado também pela sigla SAPI, como método de investigação da capacidade da ferramenta em identificar, em documentação de APIs, os fatores que possam vir a interferir na testabilidade de softwares que consomem essas APIs. Essa ferramenta utiliza os conceitos da engenharia semiótica para obter as principais falhas de comunicabilidade e, diante disso, percebeu-se que ela tem potencial para se tornar uma grande aliada para a identificação dos fatores que influenciam a testabilidade de uma API. Para atender ao objetivo, relatamos a experiência obtida do uso da ferramenta *SigniFYIng APIs* mediante a um estudo de caso de duas APIs do Portal da Transparência do Governo Federal: a API de acordos de leniência e a API de servidores. A análise das falhas de comunicabilidade, fornecidas e identificadas pela ferramenta *SigniFYIng APIs* possibilitou investigarmos maneiras de melhorar a testabilidade das aplicações consumidoras

dessas APIs. Uma vez que não há relatos na literatura sobre o uso do *SigniFYIng APIs* para identificar fatores de testabilidade, esse trabalho mostra-se como uma primeira contribuição. Como resultado, obtivemos que a aplicação da ferramenta na documentação das APIs possibilitou a identificação de falhas de comunicação e, a partir delas, foi possível projetar melhores casos de testes. Dessa forma, há indícios que o *SigniFYIng APIs* pode ser utilizado para auxiliar a equipe de qualidade da aplicação consumidora a conduzir a atividade de teste de maneira mais efetiva, bem como auxiliar a equipe desenvolvedora do serviço a melhorar a testabilidade do serviço oferecido.

Este artigo está organizado da seguinte forma: Na Seção 2 é apresentado a fundamentação teórica do método *SigniFYIng APIs* e seu funcionamento. Na Seção 3 é relatado os trabalhos relacionados ao contexto de documentações de APIs e de testabilidade em APIs. Na Seção 4 é descrita a metodologia de pesquisa bem como o processo para utilizar o *SigniFYIng APIs* para apoiar a testabilidade em APIs. Na Seção 5 é descrito e executado o estudo de caso das APIs selecionadas. Na Seção 6 é apresentado os resultados obtidos das análises de comunicabilidade que o método gera. Na Seção 7 obtém-se os resultados das análises de testabilidade mostrando como a utilização do *SigniFYIng APIs* poderia ajudar a equipe de testadores a projetar melhores casos de teste para as APIs consideradas. Por fim, a Seção 8 apresenta as conclusões e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Engenharia semiótica e comunicabilidade

A engenharia semiótica proposta em [8] é baseada na comunicação entre pessoas, que considera a interação humano-computador (IHC) como um caso especial de comunicação humana mediada por computador, a metacomunicação. Em [9], é explorado a metacomunicação no desenvolvimento de software através de comunicação indireta por meio de modelos de software, na qual pode ser caracterizada como um processo de entendimento mútuo entre os produtores "*producers*- quem concebe a modelagem do software - e os consumidores "*consumers*"– quem pega a informação dos modelos para desenvolver os artefatos.

A comunicabilidade, como vista em [10] , é definida como um conceito chave na engenharia semiótica e tem evoluído ao longo dos anos, juntamente com o refinamento da teoria. Em complemento ao conceito de comunicação, em [8], diz que a comunicação eficiente e eficaz é simplesmente a comunicação que é organizada e com recursos (eficiente), e que alcança o objetivo desejado (eficaz).

2.2 *Engenharia Semiótica e API*

Uma API tende a usar signos estáticos, dinâmicos e metalinguísticos, que são elementos da engenharia semiótica. Por exemplo, a assinatura de um procedimento pertencente a uma biblioteca emprega o nome do procedimento e dos seus argumentos; esses nomes são exemplos de signos estáticos. Um procedimento frequentemente produz um valor retornado a quem o invocou; esse valor ilustra um signo dinâmico. Um procedimento pode produzir um valor de retorno de um tipo de dados específico como "número inteiro não-negativo"; a especificação do tipo de dados através de um nome como `unsigned int` ilustra um signo metalinguístico: descreve o signo dinâmico que é produzido. O contexto de uma API implementada sobre o protocolo HTTP é análogo. O ponto de acesso de uma função é descrito por um endereço [1], podendo este ser visto como o nome do procedimento. Se o mecanismo que serve às requisições submetidas a este endereço recebe parâmetros [2], então `id` é um argumento, isto é, um parâmetro do procedimento. Um grupo de procedimentos é análogo a uma biblioteca.

A estratégia de composição desses signos determina a estratégia de comunicação entre o usuário e o sistema e faz a ligação entre esses dois interlocutores. A gramática que determina o que é válido ou inválido na comunicação delimita as possibilidades da interface do sistema, impondo as regras às quais o usuário é obrigado a respeitar para se comunicar com o sistema.

2.3 Uma visão geral do *SigniFYIng APIs*

A ferramenta *SigniFYIng APIs* é baseada em métodos da engenharia semiótica [8, 11] e também da engenharia cognitiva [12]. A novidade reivindicada [2] pela ferramenta é o uso combinado de elementos de ambas as teorias.

O *SigniFYIng APIs* institui três fases para a análise da interface de uma API qualquer, como pode-se observar na Figura 1, adaptada de [8]. As três fases compõem uma iteração do método, sendo possíveis várias iterações. Na primeira, fase denominada intenção, a análise centra-se na intenção do projetista da API, onde se inclui, por exemplo, uma definição da audiência da interface, por meio de um *frame* de metacomunicação. Uma API de programação, por exemplo, não é construída para uso de um usuário leigo de sistemas de computação, o que é percebido na observação dos signos da interface.

Na segunda fase, denominada efeito, a análise é centrada no vocabulário, na sintaxe e na semântica da API. O analista vislumbra os efeitos de se usar a API, ou seja, determina-se uma análise cognitiva das notações de programação denominadas CDNs (*Dimensão Cognitiva de Notação*[3]).

Na terceira fase, denominada falha, ocorre a interação com a API. Testes concebidos na fase anterior agora são executados, sendo sucessos e falhas documentados com o uso de etiquetas de comunicabilidade (por exemplo, *Epa!*[4], *Vai de outro jeito*[5], *Desisto*[6] e etc.) e com o vocabulário providos pelo *SigniFYIng APIs* com o objetivo de estabelecer uma linguagem comum. Por exemplo, se o cenário concebido na primeira fase for de impossível realização, usa-se a etiqueta *Falha consciente da tarefa*[7] associada à etiqueta de comunicabilidade *Desisto*[6], assumindo neste exemplo que o analista tenha percebido a impossibilidade de se satisfazer o objetivo do cenário e, portanto, nada há a fazer exceto desistir.

[1] `https://example.com/api/get-data`
[2] `https://example.com/api/get-data?id=1031`
[3] **Dimensão Cognitiva de Notação:** do inglês, *"Cognitive Dimension of Notation"*.
[4] **Epa!:** do inglês, *"Oops!"*.
[5] **Vai de outro jeito:** do inglês, *"I can do otherwise"*.
[6] **Desisto:** do inglês, *"I give up"*.
[7] **Falha consciente da tarefa:** do inglês, *"Conscious task failure"*.

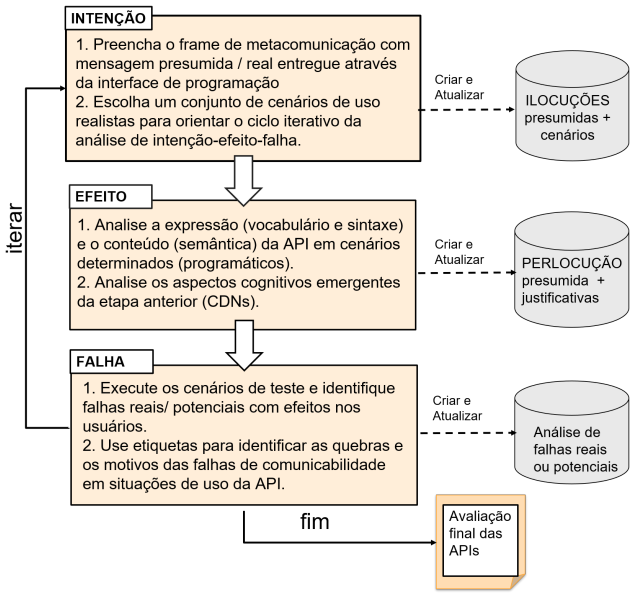

Figura 1: Visão geral das etapas do *SigniFYIng APIs*.

3 TRABALHOS RELACIONADOS

Este estudo baseou-se em analisar o uso de APIs e entender se o que ela se propõe a fazer é compreendida pela equipe de desenvolvimento e de qualidade. Dessa forma, os trabalhos relacionados a este dividem-se em dois grupos.

O primeiro refere-se a trabalhos no contexto de análise da comunicabilidade da documentação em APIs. Em [13] é abordado o desafio de aprender a utilizar uma nova API. O estudo exploratório realizado com a equipe de desenvolvimento mostrou como que os mesmos resolvem os problemas enfrentados na utilização de APIs novas. Como resultado da análise foi gerado um conjunto de diretrizes para ajudar a tornar a documentação mais eficiente.

Um pouco similar a este estudo foi o de [5]. Os autores fizeram uma análise na documentação de APIs de código aberto com o intuito de verificar o que os desenvolvedores de software procuravam ou precisavam quando olhavam a documentação de uma API. Alguns tópicos requisitados foram: uma visão geral da documentação, pequenos códigos que demonstrem o uso da API em determinado contexto, exemplos de códigos que mostram boas práticas com uma API, documentação baseada em cenários e tarefas, documentação sem conter conteúdo que agrega pouco valor.

Os estudos relatados [13] e [5] procuram identificar o bom entendimento da documentação da API como também buscar e identificar como melhorar a documentação presente, porém não é fornecido um meio de detectar se a documentação apresenta uma boa comunicabilidade com quem irá utilizar. No presente artigo propomos a aplicação do *SigniFYIng APIs* a fim de identificar aonde a comunicação da documentação falha.

O segundo grupo foi voltado ao contexto de testabilidade em APIs, no entanto, poucos trabalhos foram encontrados. O trabalho de [7] é o que mais se destaca em que é abordado como a arquitetura orientada a serviços (SOA) pode afetar todo o ciclo de desenvolvimento. Os autores propõem a avaliação de critérios de testabilidade com o intuito de apoiar a etapa de teste de software e, baseado na arquitetura SOA, estabelecem quatro critérios para avaliação de testabilidade.

Apesar dos autores descreverem os fatores que podem influenciar para alterar a testabilidade de um software, o estudo não leva em conta as falhas de comunicação que podem interferir no projeto. Diante disso, é proposto um processo unindo a técnica *SigniFYIng APIs* com o conceito de testabilidade, com o intuito de identificar os fatores que influenciam na testabilidade de um software.

4 METODOLOGIA

O objetivo deste estudo é a investigação da capacidade da ferramenta conceitual *SigniFYIng APIs* em ser utilizada como um método capaz de identificar fatores que interferem na testabilidade de um software. Sabemos que o *SigniFYIng APIs* nos fornece as principais falhas de comunicação presentes em uma API. Com base na aplicação desse contexto surge o seguinte questionamento:

(i) Será que a identificação de problemas de comunicabilidade pode nos levar a elementos para a melhoria dos casos de testes e apoiar a testabilidade?

Para responder esse questionamento a pesquisa foi dividida em quatro etapas. Na primeira etapa foi elaborada a proposta de testabilidade em API com o SAPI, conforme apresentado na Figura 2. Nessa Figura é ilustrado uma situação genérica em que temos um projetista de API, que estabelece algum tipo de comunicação, normalmente via documentação, com a equipe de desenvolvimento e de qualidade do software que utiliza a API. A equipe que irá utilizar a API tem em mãos a documentação que é fornecida pelo projetista e realiza o processo do *SigniFYIng APIs* a fim de encontrar possíveis falhas nessa documentação. Em mãos dos tipos de falhas, a equipe de teste consegue projetar casos de testes abrangendo tais falhas de comunicabilidade e, consequentemente, garantindo uma melhor qualidade do software produzido.

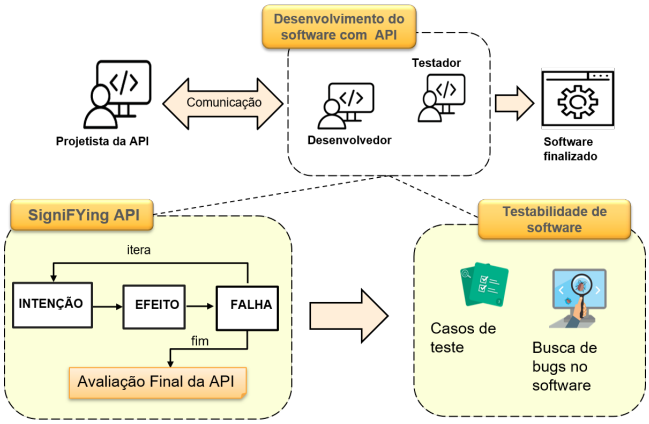

Figura 2: Testabilidade em API com o *SigniFYIng APIs*

A segunda etapa, descrita em mais detalhes na Seção 5, envolveu a elaboração e execução de um estudo de caso. O objeto de estudo escolhido foi o portal de transparência [14]. De forma aleatória, foram selecionadas duas APIs: a API de servidores e a API de acordos de leniência, ambas do sistema e-gov. O *SigniFYIng APIs* foi executado realizando iterações nas três fases que consistem o método:

intenção, efeito e falha. Gerando, ao seu término, uma avaliação da comunicabilidade da API, ou seja, determinando as principais rupturas dessa comunicação.

Na terceira etapa, descrita na Seção 6 se tem a análise das falhas em forma de CDNs, etiquetas de comunicabilidade e tipo de falhas. Por fim, na quarta etapa, descrita na Seção 7, se tem a análise de testabilidade, vislumbrado como as falhas de comunicação obtidas na etapa anterior poderiam impactar na qualidade dos casos de testes projetados.

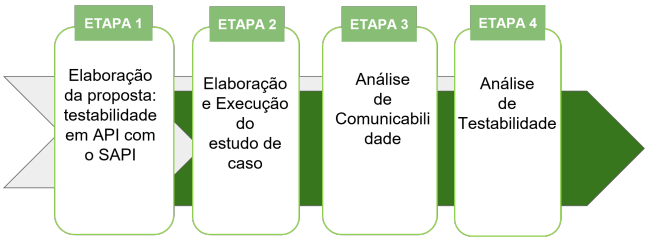

Figura 3: Etapas da metodologia abordada na pesquisa.

5 APLICANDO *SIGNIFYING APIS*

O Portal da Transparência do Governo Federal oferece várias APIs [14] documentadas utilizando um software chamado *Swagger* [15]. O uso do *Swagger* auxilia o desenvolvedor, o usuário dessa API, a utilizar a mesma por meio da documentação do serviço. Como exemplo, essa documentação fornece os parâmetros aceitos, respectivamente com as suas descrições, tipo de parâmetro utilizado e o tipo de dado. Para efeito de análise do método *SigniFYIng APIs*, foi escolhido duas APIs no Portal de Transparência: a de servidores públicos do poder executivo federal e a de acordos de leniência, que serão explicadas em detalhes nas subseções 5.1 e 5.2.

5.1 Análise da API de servidores

A análise da API de servidores públicos do poder executivo federal é iniciada pela breve observação do que se vê na apresentação produzida pelo *Swagger*, permitindo assim o inicio do primeiro passo da primeira fase do método *SigniFYIng APIs*, que é a criação do *frame de metacomunicação*.

> *Frame de metacomunicação.* Entendo que você é um programador com um mínimo de experiência com o protocolo HTTP. Você vem a mim com o propósito de obter dados sobre servidores públicos federais. Seu método preferido de obtenção de dados é via uma API HTTP. Por isso, projetei para você uma API HTTP para consumo dos dados, mas incluo também uma página na Internet com a qual você pode experimentar cada função da API isoladamente. O conjunto de funcionalidades que lhe ofereço lhe permite obter diversas informações sobre cada servidor público do poder executivo, sendo possível consultar por funções, cargos e órgãos. Quando um argumento é obrigatório, o fazemos pela sintaxe `{argumento-obrigatorio}`, como se observa em `/servidores/{id}`. Por ser um programador, você entende que um argumento obrigatório significa que é necessário prover informação antes de receber uma resposta.

Em particular, duas funcionalidades da API são `/servidores` e `/servidores/remuneracao`, o que nos permite conceber o seguinte cenário, concluindo a primeira fase do *SigniFYIng APIs*.

> *Cenário.* João é programador profissional de uma organização não-governamental que pretende obter uma estimativa do atual gasto bruto do poder executivo federal. Parte da estratégia demanda o gasto com a folha de pagamentos relativa unicamente a servidores públicos. Quanto gasta o poder executivo federal anualmente com a remuneração de servidores públicos?

Passamos agora à segunda fase. Nesta fase, procuramos inferir o comportamento da API em função de elementos dela como o vocabulário, a sintaxe e o significado do cenário em que a API se encontra.

Em virtude das mensagens transmitidas pela interface durante a primeira fase de aplicação do método, entendemos que as funções `/servidores` e `/servidores/remuneracao`

não recebem argumentos obrigatórios como, por exemplo, um identificador de um registro. A função `/servidores/por-orgao`

é relativa a órgãos, o que parece implicar uma resposta organizada por órgão. Como essas funções não exigem um argumento obrigatório, elas provavelmente retornam um conjunto de registros.

Todas essas inferências precisam ser confirmadas na fase seguinte. A CDN que se aplica ao questionamento das inferências que fizemos é *Expressividade de papéis*[8] [16]: os elementos sintáticos da interface mostram-se precários quanto a seus papéis. Não sabemos ao certo se `/servidores` de fato nos dão uma lista de servidores. Inferimos que sim e que suas remunerações não estarão contidas na resposta, o que justifica a existência de `/servidores/remumeracao`, induzindo a premissa de que redundância é usualmente indesejável em APIs.

Passamos agora à terceira fase. O primeiro teste a ser efetuado é obter a resposta de `/servidores`. Ao enviar uma requisição HTTP do tipo `GET`, obtemos uma mensagem de erro claramente nos alertando de que há um argumento que é obrigatório. O argumento é logicamente composto por uma conjunção formada pelo número de uma página e um identificador de objeto, este último sendo o CPF de um servidor ou a matrícula do servidor no sistema SIAPE de um órgão em que o servidor esteja em exercício, ou a matrícula do servidor no sistema SIAPE de seu órgão de lotação.

O erro vem como uma surpresa em função de uma análise apenas da sintaxe da função. Classificamos a falha obtida como uma falha parcial. A causa característica da falha é classificada pelo sintoma de termos agora entendido o projeto da função `/servidores`, mas desejamos outra funcionalidade — *Rejeição intencional do recurso de design*[9] com a etiqueta de comunicabilidade *Não, obrigado*[10]. Aplica-se também a *falha parcial* caracterizado por *Recusa de funcionalidade não-suspeita*[11] de etiqueta de comunidade *Vai de outro jeito*[5], uma vez que acreditamos ainda ser possível alcançar o objetivo por outros meios. Isso conclui uma primeira iteração do método.

Na segunda iteração, refinamos o *frame de metacomunicação* para refletir o que aprendemos com a primeira iteração, exibindo o

[8] **Expressividade de papéis:** do inglês, *"role-expressiveness"*.
[9] **Rejeição intencional do recurso de design:** do inglês, *"Deliberate declination of design feature"*.
[10] **Não, obrigado:** do inglês, *"Thanks, but no, thanks"*.
[11] **Recusa de funcionalidade não-suspeita:** do inglês, *"Un-suspected declination of feature"*.

refinamento a seguir em negrito. Note que a parte modificada do *frame de metacomunicação* é a segunda [16], por isso omitimos a primeira.

> *Frame de metacomunicação.* [...] O conjunto de funcionalidades que lhe ofereço lhe permite obter diversas informações[...] **desde que você detenha uma identificação do servidor com seu CPF ou matrícula**. Quando um argumento é obrigatório,[...], **sendo necessário em outros casos consultar a documentação da API ou incrementalmente observar mensagens de erro** [...].

Passando à segunda fase do método, nesta segunda iteração, analisamos a função `/servidores/por-orgao`. Pelo nome do procedimento, pode ser que obtenhamos uma lista de servidores agregados por órgão, mas pode ser que tenhamos que especificar o órgão, dado o conhecimento que obtivemos na primeira iteração.

Passando à terceira fase do método, efetuando uma requisição `GET` na função `/servidores/por-orgao` obtemos a mensagem de erro "chave de API não informada". A API é pública e não se fala em chaves de acesso em qualquer contexto ou cenário. Classificamos essa iteração como *falha parcial* caracterizado por *Preciso aprender*[12] com a etiqueta de comunicabilidade *Socorro!*[13], o que nos leva à documentação. A documentação explicita que é preciso fornecer um argumento obrigatório chamado `pagina`, que nenhuma relação tem com chaves de acesso.

Sendo assim, iteramos o método uma terceira vez, sem modificações no cenário ou *frame de metacomunicação*, e submetemos uma requisição `GET` com HTTP *query string* contendo o argumento obrigatório `pagina=1`, quando finalmente obtemos uma resposta esperada do sistema.

A resposta nos informa um quantitativo de servidores por órgão, exatamente como desejávamos, mas não há qualquer identificação de servidores, o que é necessário para obter a remuneração. Sendo assim, apesar da resposta positiva, percebemos agora que o objetivo do cenário é inatingível e classificamos esta última iteração como *falha total* caracterizado por uma falha consciente, já que estamos cientes dele, e utilizamos a etiqueta de comunicabilidade *Desisto*[6], visto que o objetivo é inatingível.

Adicionalmente, consideramos usar o quantitativo de pessoas por órgão, o que a API nos forneceu com sucesso, de forma que usando uma média salarial pudéssemos pelo menos dar um limite inferior e superior, parcialmente satisfazendo o objetivo do cenário. Em uma segunda inspeção da resposta obtida pela API, notamos que o quantitativo total de servidores públicos do governo federal é de 170 pessoas, dentre todos os órgãos mencionados pela API, o que não reflete a realidade.

5.2 Análise da API de acordos de leniência

A API - Acordos de Leniência - nos possibilita consultar registros de acordos de leniência por nome ou CNPJ do sancionado, situação e período por meio de requisições `GET/acordos-leniencia`.

Iniciamos a primeira fase, que consiste em criar um frame de metacomunicação, como a seguir:

> *Frame de metacomunicação.* Entendo que você é um programador com um mínimo de experiência com o protocolo HTTP. Você vem a mim com o propósito de obter dados sobre acordos de leniência, o que está claro pelo próprio nome das minhas funções. Seu método preferido de obtenção de dados é via uma API HTTP. Por isso, projetei para você uma API HTTP para consumo dos dados.

As funcionalidades fornecidas pela API `/acordos-leniencia` possibilitam criar o cenário descrito a seguir, concluindo a primeira fase do *SigniFYIng APIs*.

> *Cenário.* Ana é uma programadora e foi contratada por uma empresa de advocacia para implementar uma interface que exiba os acordos de leniência sancionados há um ano, dada uma data específica.

Iniciamos a segunda fase observando que a API de acordo de leniência fornece as seguintes funções: `/cnpjSancionado`, `/situacao/nomeSancionado` `/dataInicialSancao` `/dataFinalSancao` e `/pagina`

Como definido no cenário na fase anterior, estou interessado em obter todos os acordos de leniência sancionados há um ano. Diante disto escolho a data 25/06/2019, então compreendo que para satisfazer o objetivo devo utilizar a função `/acordos-leniencia/dataInicialSancao`.

Agora, analisamos o vocábulo *dataInicialSancao*. A sintaxe da palavra, *dataInicialSancao*, sugere que seja inserido a data de início da aprovação do acordo de leniência. Observando semanticamente, ou seja, a resposta que se espera obter, é um conjunto de acordos de leniência sancionados na data 25/06/2019.

Realizada a análise da expressão, a próxima etapa é a cognição a respeito da função, ou seja, está claro pela sintaxe `/acordos-leniencia/dataInicialSancao` que a API mostrará todos os acordos com início na data de sanção escolhida, já que a remuneração "pertence" à hierarquia servidores, o que é transmitido pela sintaxe da URI.

Iniciamos a terceira fase, realizo uma requisição GET na função `/acordos-leniencia/dataInicialSancao` obtendo uma lista de acordos. Porém noto que não é possível encontrar a sintaxe adotada pelo o vocábulo *dataInicialSancao*. Em vez disso, a busca me retorna *dataInicioAcordo*, o que me leva a pensar se seria a mesma data de início de sanção, já que o vocábulo "sanção" [14] tem um significado diferente do vocábulo "acordo" [15].

Com o intuito de entender melhor se, de fato, os vocábulos significam a mesma coisa, efetuo o seguinte teste: na url da requisição, troco a palavra *dataInicialSancao* por *dataInicioAcordo*. Após a execução realizada, percebo que retorna os mesmos valores. Ou seja, de fato *dataInicialSancao* e *dataInicioAcordo* são adotados com o mesmo significado nesse contexto. Classificamos essa iteração como *falha parcial* caracterizado por *Preciso descobrir*[16] com a etiqueta de comunicabilidade *Por que não funciona?*[17],

Isso conclui então a primeira iteração do método. Inicio uma nova iteração, refinando o frame de metacomunicação.

> *Frame de metacomunicação.*[...] que fornece um conjunto de funções que lhe dá múltiplas formas de obter acordos de leniência, como `/acordos-leniencia/dataInicialSancao`,

[12] **Preciso aprender:** do inglês, *"Need to learn"*.
[13] **Socorro!:** do inglês, *"Help!"*.
[14] "Ato pelo qual um chefe de Estado aprova e confirma uma lei."[17]
[15] "Combinação ajustada entre duas ou mais pessoas."[18]
[16] **Preciso descobrir:** do inglês, *"Need to discover"*.
[17] **Por que não funciona?:** do inglês, *"Why doesn't it?"*.

> no qual poderá obter acordos de leniência relativas a data inicial de sanção.

Inicio a segunda fase do método, nela já conseguimos identificar a CDN de consistência, já que as diferentes sintaxes apresentadas geraram a dúvida de se ambas produziriam o mesmo resultado.

Inicio a terceira fase do método em busca de identificar as potenciais falhas no cenário descrito. Agora, inicio um segundo teste. Observo que é retornado da função uma lista de acordos sancionados que contêm datas que, aparentemente, iniciam no dia 25 e consequentemente todas as sanções realizadas dessa data para frente. Porém, antes de executar o este, eu havia assumido que a função retornaria apenas os acordos sancionados no dia 25. O que me fez duvidar se a busca também retorna datas após o dia 25 ou apenas datas com início no dia 25. Para confirmar a informação, faço os seguintes procedimentos:

(1) Escolho uma nova *dataInicioAcordo* a partir de um dos acordos de leniência retornados na chamada à API com *dataInicioAcordo* = '25/06/2019' e que seja maior que a consultada, por exemplo, '26/06/2019'
(2) Faço uma outra requisição ao serviço a partir desta nova data posterior para ver se este mesmo acordo também é retornado

Ao alterar a *dataInicioAcordo* para '26/06/2019' vejo que o mesmo acordo de leniência, escolhido no procedimento 1, foi retornado na consulta, mas não recebi nenhum acordo que se iniciasse antes da data especificada. Assim, concluo que a requisição está realmente me retornando os acordos que se iniciam na data solicitada e, consequentemente, nas datas posteriores a ela.

Após a execução desse teste, já posso associar a situação como uma *falha parcial*, o que leva a situação *Preciso conhecer*[18] com etiqueta de comunicabilidade *O que é isso?*[19], o que caracteriza a dúvida de qual a resposta será obtida visto que a sintaxe do vocábulo escolhido, *dataInicialSancao*, não me deixou claro qual seria a resposta que a ser obtida. Por exemplo, a utilização do vocábulo *datasDeSançõesIniciadasApartirDe* deixaria muito mais claro o que a função faz.

Após isso, finalizamos a terceira iteração já que não foi observada, na terceira fase, nenhuma outra falha no cenário de uso.

6 RESULTADOS DAS ANÁLISES DE COMUNICABILIDADE

Após a aplicação do *SigniFYIng APIs* como relatado nas seções 5.1 e 5.2, foi possível identificar as principais rupturas de comunicabilidade que a API de servidores e de acordo de leniência possuem.

Como resultados temos a CDNs e as etiquetas de comunicabilidade. As CDNs levam em conta a interpretação do texto e o perfil cognitivo da linguagem de programação. Já as etiquetas de comunicabilidade [16] vinculam a atividade analítica de falhas às obtidas na primeira e na segunda etapa do método.

6.1 Análise da API de servidores

Na primeira análise da API de servidores, o método visto em 5.1 nos proporcionou encontrar os seguintes resultados:

A CDN relacionada nesse cenário foi a *Expressividade de papéis*[8], ou seja, os elementos sintáticos da interface mostram-se precários quanto a seus papéis. Não sabemos ao certo se `/servidores`, de fato, nos dá uma lista de servidores. Inferimos que sim e que suas remunerações não estarão contidas na resposta, o que justifica a existência de `/servidores/remuneracao`.

As etiquetas de comunicabilidade encontradas foram *Não, obrigado*[10], *Vai de outro jeito*[5], *Socorro!*[13] e *Desisto*[6]. A etiqueta *Não, obrigado*[10] está associada à situação *Rejeição intencional do recurso de design*[9], uma *falha parcial*. A causa desse falha é classificada pelo fato de que desejamos outra funcionalidade da API, embora tenhamos compreendido a funcionalidade de `/servidores`. A etiqueta *Vai de outro jeito*[5] está associada à situação *Recusa de funcionalidade não-suspeita*[11], uma *falha parcial*, uma vez que acreditamos ser possível alcançar o objetivo por outros meios. A etiqueta *Socorro!*[13] está associada à situação *Preciso aprender*[12], uma *falha parcial*, devido a um entendimento incorreto da documentação. A documentação explicita que é preciso fornecer um argumento obrigatório chamado `pagina`, que deve aparecer na *query string* da URI e que nenhuma relação tem com chaves de acesso. Por fim, a etiqueta *Desisto*[6] está associada à situação *Falha consciente da tarefa*[7], que caracteriza uma *falha total*, visto que o objetivo é inatingível. A resposta nos informa um quantitativo de servidores por órgão, exatamente como desejávamos, mas não há qualquer identificação de servidores, o que é necessário para obter a remuneração. Sendo assim, apesar da resposta esperada, percebemos agora que o objetivo do cenário é inatingível.

6.2 Análise de acordo de leniência

Já na segunda análise da API vista em 5.2, o método nos proporcionou a encontrar os seguintes resultados:

A CDN encontrada foi *Consistência*[20], apresentando semânticas semelhantes que são expressas em formas sintáticas semelhantes, o que leva o usuário à dúvida se elas de fato produziriam o mesmo resultado.

As etiquetas de comunicabilidade encontradas foram *Por que não funciona?*[17] e *O que é isso?*[19]. A etiqueta *Por que não funciona?*[17] é associada à situação *Preciso descobrir*[16], que caracteriza uma *falha parcial* pois não fica claro para o usuário se *dataInicioAcordo* é o mesmo que *dataInicioSanção*. Já a etiqueta *O que é isso?*[19] é associada à situação *Preciso conhecer*[18], o que caracteriza uma *falha parcial*, visto que a sintaxe do vocábulo escolhido, *dataInicialSanção*, não esclarece claro qual seria a resposta a ser obtida.

6.3 Análise dos tipos de falhas

Na Seção 5 aplicamos o *SigniFYIng APIs* e ilustramos através da criação dos cenários de uso advindos da técnica fornecida. Isso possibilitou a identificação de situações como, por exemplo, na API servidores, o usuário desejar outra funcionalidade da API, apesar de ter compreendido a funcionalidade dada. Tais situações que são obtidas na ferramenta geram diferentes tipos de falhas no software.

Observando os tipos de falhas que foram encontrados em cada API, obteve-se na API de servidores o total de 3 *falhas parciais* e 1 *falha total*, enquanto que na API de acordos de leniência houveram 2 *falhas parciais*. A obtenção da *falha parcial* significa que o usuário

[18] **Preciso conhecer:** do inglês, *"Need to Know"*.
[19] **O que é isso?:** do inglês, *"What's this?"*.
[20] **Consistência:** do inglês, *"Consistency"*.

pode perder ou recusar a interação que o *designer* esperava que ele se envolvesse, apesar dele ter atingido seu objetivo fazendo de outra maneira. Já na *falha total*, temos a situação que o usuário perde completamente a comunicação do *designer* e falha em atingir o seu objetivo esperado [16].

A gravidade da falha é identificada de acordo com o seu tipo, assim como a sua relevância de gravidade, ou seja, a gravidade de uma *falha temporária* é menor que uma *falha parcial*; a *falha parcial* é menos grave que uma *falha total*. Da análise realizada, nota-se que o único tipo de falha que não foi observado no estudo aplicado foi a *falha temporária*, no qual o mesmo é definido quando o usuário perde temporariamente a compreensão da API.

Após aplicar a técnica nas APIs de servidores e de acordo de leniência, verificamos que a ferramenta é simples de ser utilizada e que nos permitiu identificar, além das etiquetas de comunicabilidade e das CDNs, os principais tipos de falhas ocorridas na comunicação das referidas APIs.

7 RESULTADOS DAS ANÁLISES DE TESTABILIDADE

O projeto de casos de teste, assim como a qualidade da suíte de testes, depende de quanto o testador conseguiu entender eficazmente o que a API está comunicando. Uma vez que o *SigniFYIng APIs* possibilita a identificação de falhas de comunicação em uma API, podemos então associar essas falhas como um fator que irá diminuir a testabilidade da API e, consequentemente, do software criado com a sua utilização.

A Tabela 1 sintetiza o resultado das análises de testabilidade da documentação das APIs de acordo de leniência e de servidores com o *SigniFYIng APIs* . Nessa Tabela, a coluna *Situação* corresponde às situações descritas na Seção 5.1 e na Seção 5.2. A coluna *Falha sem o SigniFYIng APIs* lista as falhas associadas à cada uma dessas situações sem o uso prévio da ferramenta. Sem essas informações, a equipe de teste criaria os casos de teste listados na coluna *Caso de Teste*, que levariam as falhas listadas na coluna *Falha do Teste sem o SigniFYIng APIs* . Por exemplo, um caso de teste possível mediante a esse cenário seria a consulta de acordos de leniência com dado de entrada igual a $dataInicialSancao$ e dado de saída igual a 10/07/17. Nesse exemplo o caso de teste falharia nos seguintes cenários:

(1) A primeira falha seria que o teste criado não encontraria na resposta nenhum parâmetro $dataInicialSancao$, pois como visto na análise, o parâmetro não se manteve com o mesmo nome, obtendo *null* como resposta.
(2) A segunda falha seria que o caso de teste criado não obteria como resposta apenas acordos relativos a data 01/10/20, mas uma lista contendo também as datas posteriores, levando assim a falha do teste.

Tais falhas relatadas teriam sido evitadas se o testador tivesse conhecimento prévio delas. Em mãos dessas informações o testador teria criado para as duas situações relatadas anteriormente casos de testes mais adequados. Na primeira situação, descrita no item 1, o teste projetado, como mostrado na coluna *Testes projetados com o SigniFYIng APIs* já consideraria que o retorno esperado seria $dataInicioAcordo$ ao invés do retorno da $dataInicialSancao$. Já na segunda situação, descrita no item 2, o teste já consideraria uma lista de datas ao invés de uma data apenas.

Outro ponto observado é que além de oferecer informações de qual deve ser a saída esperada para um caso de teste previamente existente, foi possível também ampliar o conjunto inicial. Um exemplo é a situação "Consultar todos os servidores", descrita também na Tabela 1. Uma vez que o testador sabe que precisa informar um novo dado de entrada, ele pode aplicar o critério Análise do Valor Limite, da técnica de teste funcional [19], por exemplo, para criar três novos casos de testes que considerem (i) a primeira página ($pagina = 1$), (ii) a última página ($pagina = max$, sendo max igual ao número máximo de páginas) e (iii) a página posterior à última página ($pagina = max + 1$), ou seja, uma página inexistente.

Nota-se, entretanto que em algumas situações não é possível melhorar o conjunto de casos de teste inicial. Um exemplo é situação "Consultar /servidores/por-orgao e /servidores" que, apesar do testador ter em mãos a falha obtida, ele ainda continuaria incapaz de projetar um caso de teste adequado pois, para isso, é essencial saber qual é a saída esperada. Entretanto, esses resultados ainda poderiam ser utilizados pela equipe desenvolvedora do serviço para melhorar a documentação de suas APIs.

8 CONCLUSÕES E TRABALHOS FUTUROS

Em vista do aumento da complexidade que os sistemas têm tomado, nas diversas esferas da sociedade, observa-se a exigência de serem feitas integrações entre serviços que se comunicam por meio de APIs. O desafio de construir uma API bem projetada e que seja bem compreendida pelo consumidor do serviço ainda é de difícil abordagem, visto que as formas de documentações de APIs não é universal, prejudicando as atividades de garantia de qualidade. Diante disso, nota-se a importância de se ter um método ou soluções sólidas que garanta a qualidade de desenvolvimento do software que utiliza um serviço.

Como primeira contribuição, este trabalho propõe o uso da ferramenta *SigniFYIng APIs* como método para investigar a capacidade da ferramenta em identificar, na documentação de APIs, os fatores que possam vir a alterar a testabilidade e, com isso, possibilitar uma melhora na qualidade do software produzido. Com o intuito de verificar se a ferramenta atingia tal propósito, realizamos um estudo de caso com duas APIs do governo federal: a API de acordo de leniência e a API de servidores. O conhecimento que é adquiridos da documentação da API, a partir do uso do *SigniFYIng APIs* , possibilitou a apropriação das falhas de comunicabilidade encontradas para antecipar a avaliação de defeitos nos casos de teste ocasionados pela má compreensão da API. Observamos que o processo proposto mostrou indícios de sucesso para algumas situações de uso para o testador, já que o mesmo teria o conhecimento prévio das falhas que poderiam ser evitadas ao projetar os casos de teste. Já em outra situação, observou-se que, apesar do testador ter em mãos as falhas, não teria como projetar um caso de teste adequado pois não se sabe qual seria a saída esperada. Diante desses resultados, obtivemos indícios de que a ferramenta *SigniFYIng APIs* pode ser utilizada como apoio na condução da atividade de teste de software. Dado o escopo amplo da ferramenta, inúmeros outros estudos ainda são necessários, sendo esta uma primeira contribuição.

Consideramos neste trabalho que o processo proposto é utilizado pela equipe de desenvolvimento e de testes do software que consome a API. No entanto, os resultados obtidos por meio do processo

Situação	Falha	Caso de Teste	Falha do teste sem o Signifying API	Testes projetados com o SignifYIng API
Consultar *dataInicialSancao*	Variável *dataInicialSancao* não encontrada obtenho *dataInicioAcordo*.	**Entrada:**get/*dataInicialSancao* =10/07/17 **Saída esperada:** 10/07/17	O teste falha pois ele esperava como retorno 10/07/17 mas obteve a data *dataInicioAcordo*.	Testador projetaria casos de teste esperando como retorno a *dataInicioAcordo*.
Consultar acordos de leniencia feitos na data 10/07/2017	Informar *dataInicialSancao*= 10/07/17, mas recebeu outras além da solicitada.	**Entrada:** get/*dataInicialSancao* =10/07/17 **Saída esperada:** 10/07/17	O teste falha pois ele esperava como retorno 10/07/17 mas obteve uma lista de datas.	O testador projetaria casos de teste que validasse se o retorno é uma lista de datas.
Consultar /servidores/por-orgao e /servidores	Variável /servidores/por-orgao e /servidores não são especificadas os seus retornos.		O testador não seria capaz de projetar os casos de teste pois não sabe qual será a saída esperada.	O testador não seria capaz de projetar os casos de teste pois não sabe qual será a saída esperada.
Consultar todos os servidores	Aparece uma mensagem de erro de argumento obrigatório solicitado. Solicitado o número da página.	**Entrada:** get/servidor **Saída esperada:** lista de todos os servidores	O teste não consegue ser executado.	O testador projetaria casos de testes considerando o número da página como entrada e obteria como resultado todos os servidores dessa página. Ele também projetaria casos de teste que considerassem a primeira e últimas páginas, além de uma página inexistente
Consultar todos os servidores por orgão	Aparece a mensagem de erro "chave de API não informada".	**Entrada:** get/servidor/por-orgao **Saída esperada:** lista com todos os servidores de um determinado orgão	O teste não consegue ser executado	O testador projetaria o caso de teste de forma a informar também a chave de API.
Consultar o quantitativo de servidores por orgão necessários para obter a remuneração	A requisição get/servidor/por-orgao&pagina=1 não retorna a identificação dos servidores.	**Entrada:** get/servidor/por-orgao &pagina=1 **Saída esperada:** listagem dos servidores com suas remunerações	O teste falha pois ele esperava uma lista contendo a identificação dos servidores.	O testador projetaria casos de teste sem esperar a identificação dos servidores.

Tabela 1: Análise das falhas nos casos de teste

também poderiam ser utilizados pela equipe de desenvolvimento da API. Nesse caso, uma vez identificadas situações em que não é possível projetar casos de testes, o produtor do serviço poderia melhorar a documentação para melhorar a testabilidade. Trabalhos futuros incluem a definição de um processo sistemático para tal. Outros trabalhos futuros incluem a condução de mais estudos de caso, considerando outras APIs e em diferentes cenários para refinar a processo proposto.

REFERÊNCIAS

[1] Anna Maria Eilertsen and Anya Helene Bagge. Exploring api: Client co-evolution. In *Proceedings of the 2nd International Workshop on API Usage and Evolution*, WAPI '18, page 10–13, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357548. doi: 10.1145/3194793.3194799.

[2] Luiz Marques Afonso, Renato F de G Cerqueira, and Clarisse Sieckenius de Souza. Evaluating application programming interfaces as communication artefacts. *System*, 100:8–31, 2012.

[3] Ferdian Thung. Api recommendation system for software development. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 896–899, 2016.

[4] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, USA, 9th edition, 2010. ISBN 0137035152.

[5] Robert Watson, Mark Stamnes, Jacob Jeannot-Schroeder, and Jan H. Spyridakis. Api documentation and software community values: A survey of open-source api documentation. In *Proceedings of the 31st ACM International Conference on Design of Communication*, SIGDOC '13, page 165–174, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450321310. doi: 10.1145/2507065.2507076. URL https://doi.org/10.1145/2507065.2507076.

[6] IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990. doi: 10.1109/IEEESTD.1990.101064.

[7] Wei-Tek Tsai, Jerry Gao, Xiao Wei, and Yinong Chen. Testability of software in service-oriented architecture. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 2, pages 163–170, 2006. doi: 10.1109/COMPSAC.2006.167.

[8] Clarisse Sieckenius de Souza, Carla Faria Leitão, Raquel Oliveira Prates, Sílvia Amélia Bim, and Elton José da Silva. Can inspection methods generate valid new knowledge in hci? the case of semiotic inspection. *International Journal of Human-Computer Studies*, 68(1):22–40, 2010. ISSN 1071-5819. doi: https://doi.org/10.1016/j.ijhcs.2009.08.006. URL https://www.sciencedirect.com/science/article/pii/S1071581909001128.

[9] Adriana Lopes, Édson César Oliveira, Tayana Conte, and Clarisse Sieckenius de Souza. Directives of communicability: Towards better communication through software models. In *2019 IEEE/ACM 12th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 45–48, 2019. doi: 10.1109/CHASE.2019.00019.

[10] Luiz Marques Afonso. *Communicative dimensions of application programming interfaces (APIs)*. PhD thesis, Pontificia Universidade Católica do Rio de Janeiro, Rio de Janeiro, RJ, Brazil, 2015.

[11] Clarisse Sieckenius de Souza and Carla Faria Leitão. *Semiotic Engineering Methods for Scientific Research in HCI*. Morgan & Claypool Publishers, 2009.

[12] Donald A Norman. Cognitive engineering. *User centered system design*, 31:61, 1986.

[13] Michael Meng, Stephanie Steinhardt, and Andreas Schubert. How developers use api documentation: An observation study. *Commun. Des. Q. Rev*, 7(2):40–49, August 2019. doi: 10.1145/3358931.3358937. URL https://doi.org/10.1145/3358931.3358937.

[14] Api rest do portal da transparência do governo federal, 2020. URL http://www.transparencia.gov.br/swagger-ui.html. Acessado em: 11/06/2020.

[15] Swagger: Api documentation & design tools for teams, 2020. URL https://swagger.io/. Acessado em: 11/06/2020.

[16] Clarisse Sieckenius De Souza, Renato Fontoura de Gusmão Cerqueira, Luiz Marques Afonso, Rafael Rossi de Mello Brandão, and Juliana Soares Jansen Ferreira. *Software Developers as Users*. Springer, 2016.

[17] Priberam. Dicionário priberam da língua portuguesa, 2020. URL https://dicionario.priberam.org/sancao. Acessado em: 11/06/2020.

[18] Priberam. Dicionário priberam da língua portuguesa, 2020. URL https://dicionario.priberam.org/acordo. Acessado em: 11/06/2020.

[19] Marcio Delamaro, Mario Jino, and Jose Maldonado. *Introdução ao teste de software*. Elsevier Brasil, 2013.