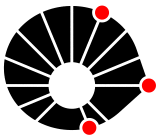

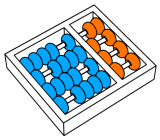

Impactos da utilização de *Feature Flags* na manutenção de código.

Lucas Antevere Santana *Breno Bernard Nicolas de França*

Relatório Técnico - IC-PFG-23-39
Projeto Final de Graduação
2023 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Impactos da má utilização de Feature Flags na manutenção de código.

Lucas Antevere Santana*‡ Breno Bernard Nicolas de França

Resumo

Este estudo investiga os potenciais impactos da utilização de *Feature Flags* na manutenção de código. O objetivo principal é compreender se o emprego dessas *flags* está associado a problemas na manutenção do *software*. Para alcançar esse objetivo, foi realizada uma avaliação de *Code Smells* em um repositório *open source*, seguida por uma análise estática e cruzamento de informações sobre os arquivos que fazem uso de *Feature Flags*. A identificação desses arquivos foi realizada por meio de um mapeamento usando expressões regulares no repositório estudado.

Os resultados apontam que o grupo de arquivos contendo *Feature Flags* apresentou uma maior densidade de *Code Smells* por arquivo em comparação com aqueles sem *flags*. No entanto, uma análise qualitativa revelou que nenhum *Code Smell* foi diretamente causado pelo uso dessas *flags*. Portanto, neste repositório específico, não foram observados impactos negativos na manutenção do código decorrentes da utilização de *Feature Flags*.

*Viviane Antevere Batagini
†Franciele Mendes Machado
‡Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

1 Introdução

A evolução constante no desenvolvimento de *software* demanda abordagens adaptativas para lidar com mudanças simultâneas e frequentes alterações nos requisitos e funcionalidades. Nesse contexto, as *Feature Flags*, ou "chaves de funcionalidade", surgem como uma ferramenta que possibilita aos desenvolvedores habilitar ou desabilitar funcionalidades em tempo de execução, sem a necessidade de alterações diretas no código-fonte, facilitando a integração contínua de novos trechos de código.

No cenário ágil, as *Feature Flags* viabilizam a incorporação incremental de lógica e trechos de código, permitindo a introdução de funcionalidades em desenvolvimento ou experimentais no código-base sem comprometer a estabilidade do sistema. Contudo, essa flexibilidade também implica na adição de novas lógicas condicionais e caminhos alternativos, o que pode complexificar o código.

Este estudo tem como objetivo explorar os impactos da utilização de *Feature Flags* e compreender se seu uso influencia na manutenção de código. Busca-se entender como essa implementação das *flags* pode afetar a qualidade e a facilidade de manutenção do software, caso a relação entre *Feature Flags* e manutenção de código se confirme.

A escolha desse tema é motivada pela crescente adoção de *Feature Flags* no mercado como estratégia de desenvolvimento ágil, aliada à importância crítica da manutenção de código em projetos de software. Entender os impactos gerados pela presença de *Feature Flags* é essencial para identificar boas práticas e evitar armadilhas que possam comprometer a integridade e a eficiência do código ao longo do tempo.

2 Fundamentação Teórica

2.1 *Feature Flags*

Feature Toggles, também conhecidos como *feature flags*, são mecanismos de desenvolvimento de *software* que proporcionam aos desenvolvedores a capacidade dinâmica de ativar ou desativar funcionalidades específicas durante a execução do programa. Essas *flags* são implementadas por meio de condicionais (*if's*), que determinam qual trecho de código será executado naquele momento [3].

O funcionamento desses mecanismos envolve a adição de uma camada ferramental no sistema, onde, a uma frequência determinada pelo desenvolvedor, o sistema acessa um serviço remoto para avaliar o estado de cada *flag*. Essa frequência pode ser ajustada para ocorrer a cada inicialização, em intervalos fixos de tempo ou durante a execução de trechos específicos de código.

O uso amplo dessa ferramenta no desenvolvimento de software acontece por conta da flexibilidade oferecida pelos *Feature Flags*, que habilita o time desenvolvedor a executar estratégias modernas e com risco reduzido na construção de novas funcionalidades, como a integração contínua e gradual de novos trechos de código.

Além disso, seu uso permite a aplicação de estratégias de lançamento de funcionalidades onde ocorre a sua ativação incremental para diferentes grupos de usuários, permitindo uma análise aprofundada do impacto antes da liberação completa e a rápida reversão de alterações indesejadas.

2.2 Má utilização de *Feature Flags*

Apesar do ganho de flexibilidade no desenvolvimento e lançamento de novas capacidades, as *feature flags* acrescentam integrações e implementações desses serviços, bem como a necessidade de implementar condicionais que preservam o código anterior. Essa necessidade acarreta em código adicional no sistema, que é suscetível às más práticas de desenvolvimento. Além disso, a má utilização das *feature flags* no código pode acarretar uma série de impactos negativos[2], comprometendo a manutenção e até mesmo a segurança do software.

O excesso de *feature flags* não gerenciadas pode resultar em complexidade desnecessária no código, tornando-o difícil de entender e manter. Isso pode levar a uma degradação da legibilidade e compreensão, aumentando a probabilidade de introdução de *bugs* e dificultando a colaboração entre membros da equipe.

Ainda, a má gestão das *feature flags* pode resultar em acumulação de código morto, ou seja, trechos de código relacionados a funcionalidades desativadas, mas que ainda estão presentes no código-base. Isso não apenas aumenta a complexidade, mas também contribui para um aumento no consumo de recursos e no tempo de compilação. Além disso, pode representar um risco de segurança, uma vez que código não utilizado pode conter vulnerabilidades que passam despercebidas.

Outro impacto negativo é a possibilidade de confusão na lógica de negócios do software. Se as *feature flags* não forem cuidadosamente gerenciadas, a presença de ramificações condicionais para diferentes funcionalidades pode tornar o código propenso a erros lógicos, prejudicando a consistência e a confiabilidade do sistema como um todo.

Assim, a má utilização de *feature flags* pode resultar em um código menos eficiente, difícil de manter e potencialmente propenso a problemas de segurança.

2.3 Code Smells

Para realizar uma análise da qualidade do código em um repositório, é necessário levantar e definir heurísticas que possam ser medidas e avaliadas. Para esse trabalho, optou-se pela medição da qualidade de código por meio da medição da quantidade de *Code Smells* presentes e se eles foram causados diretamente pela implementação das *Feature Flags*.

Code Smells referem-se a padrões de design ou práticas de programação que podem indicar a presença de problemas ou deficiências em um sistema de software. Estes não são defeitos ou *bugs* propriamente ditos, mas sim indicadores de possíveis áreas problemáticas que podem comprometer a qualidade e a manutenibilidade do código-fonte [1]. A detecção precoce de *code smells* é crucial para mitigar potenciais complicações no desenvolvimento e garantir a criação de software mais robusto e sustentável.

Um exemplo comum de *code smell* é a duplicação de código, onde trechos similares (ou idênticos) de lógica são replicados em diferentes partes do código, aumentando a probabilidade de inconsistências e dificultando futuras modificações. Outro *smell* é a complexidade excessiva, onde funções ou classes tornam-se intrincadas e difíceis de compreender, o que pode resultar em dificuldades na manutenção e introduzir oportunidades para *bugs*.

A identificação e correção de *code smells* são elementos cruciais da prática de refatoração, um processo iterativo que visa melhorar a estrutura interna do código sem alterar seu comportamento externo [1]. Ferramentas automatizadas de análise estática de código são frequentemente empregadas para detectar a presença de *code smells*, auxiliando os desenvolvedores na manutenção da qualidade do *software* ao longo do ciclo de vida do projeto.

3 Métodos

3.1 Linguagem de Programação

Como as *flags* são elementos utilizados dentro do código-fonte, a escolha da linguagem de programação alvo é um fator importante para uma condução de análises específicas sobre os impactos da sua má utilização na manutenção de código, sobretudo por aspectos técnicos de implementação e automação. É necessário conhecimento prévio da linguagem para entender como as *flags* podem ser integradas ao sistema, incluindo sua sintaxe de invocação e definição de regras de detecção para todas as ocorrências dentro do código-fonte, de forma que a análise seja precisa. Assim, optou-se pela linguagem *Swift*[1]. A razão para essa escolha reside na amplitude do uso dessa linguagem, especialmente em projetos voltados para o ecossistema iOS. *Swift* não só desfruta de uma posição proeminente no desenvolvimento de aplicativos móveis, mas também se estende para outras plataformas, tornando-se uma linguagem versátil e atual.

3.2 Repositório Estudado

Para definição do repositório a ser analisado, optou-se pela seleção do repositório do cliente *iOS* do *Mozilla Firefox*[2], um navegador web desenvolvido pela *Mozilla Foundation*. Essa organização foi fundada com o objetivo de garantir a abertura e acessibilidade na web por meio da promoção do desenvolvimento de um navegador de código aberto e desempenha um papel crucial no desenvolvimento de tecnologias que promovam a liberdade, privacidade e inovação online.

Com a colaboração e a participação ativa da comunidade, a *Mozilla Foundation* não apenas incentiva e promove soluções de software de código aberto, mas também lidera iniciativas para garantir um ecossistema digital saudável e ético.

A escolha desse repositório como objeto de análise atende alguns critérios estratégicos para o objetivo desse estudo. Em primeiro lugar, a natureza *open source* do projeto oferece transparência e acessibilidade aos detalhes de implementação. Isso é crucial para uma análise aprofundada, permitindo uma compreensão completa das estratégias adotadas no uso de *feature flags*. A transparência também contribui para a validade e replicabilidade do estudo, uma vez que outros pesquisadores podem examinar e verificar os resultados.

Por fim, o *Mozilla Firefox* é amplamente utilizado, e o cliente *iOS* representa uma parte significativa desse ecossistema, o que proporciona um ambiente diversificado para examinar a presença e impacto das *feature flags*. A escolha desse repositório é motivada pela sua relevância no cenário de desenvolvimento *iOS*. Ao explorar o cliente *iOS* do *Mozilla Firefox*, é possível obter conclusões que, possivelmente, beneficiarão a compreensão sobre o impacto das *feature flags* e também contribuirão para a prática no desenvolvimento de software para dispositivos *iOS*..

[1] `https://developer.apple.com/swift/`

[2] `https://github.com/mozilla-mobile/firefox-ios`

3.3 Identificação das *Feature Flags*

Para identificação sistemática das *flags* no código-fonte do cliente *iOS* do *Mozilla Firefox* se propõe a implementação de um script em *Python*, fazendo uso de expressões regulares (*Regex*) para detecção eficiente. O script seria projetado para analisar os arquivos, empregando *Regexs* específicas para identificar padrões indicativos de chamadas de *feature flags*. Essa abordagem permite uma varredura eficaz e adaptável aos diferentes estilos de implementação de *flags* presentes no código.

A combinação de *Python* e *regex* não apenas facilita a eficiência na identificação das *feature flags*, mas também proporcionou adaptabilidade, permitindo ajustes do *script* conforme necessário para lidar com diferentes nuances presentes no código-fonte. O resultado desse processo de identificação é uma listagem dos arquivos dentro do repositório e a contagem das *flags* encontradas em cada arquivo um deles.

Para a organização estruturada dessas informações, a proposta é pelo armazenamento em um banco de dados *SQL*, onde a tabela construída possuiria 2 colunas: uma representando o arquivo e outra representando a quantidade de *flags* nesse arquivo.

3.4 Detecção de *Code Smells*

Seguindo a investigação, a avaliação da qualidade do código é necessária para que se possa testar correlações entre a mesma e as *features flags*. Para isso, a estratégia definida é o uso do *SonarCloud*[3], uma ferramenta de análise estática de código que oferece uma abordagem sistemática e automatizada na detecção de *code smells*.

O *SonarCloud*, como ferramenta de análise estática, destaca-se por sua capacidade de identificar diversas classes de *code smells*, incluindo alta complexidade ciclomática, duplicação de código, códigos inseguros, módulos excessivamente grandes, entre outros. Essa variedade de categorizações proporciona uma visão detalhada das áreas passíveis de problemas de manutenção, contribuindo para uma compreensão mais completa da qualidade do código.

Outra característica que apoia a vantagem do uso dessa ferramenta é a API aberta do *SonarCloud*. Esta API possibilita o acesso direto aos resultados da análise, permitindo que consultas detalhadas possam também ser realizadas por dos *scripts* em *Python*, fazendo com que essa etapa seja uma continuação natural dos *scripts* que serão criados para a detecção das flags.

Uma vez que se obteve a detecção dos *code smells* por arquivos, pode-se armazená-los no mesmo banco *SQL* onde se armazenou a listagem de arquivos e flags. O mais apropriado será adicionar novas colunas representando os graus de severidades existentes de *smells* e em cada linha, que representa um arquivo no repositório, se armazena a quantidades de *smells* com aquela severidade naquele arquivo.

[3] `https://sonarcloud.io`

3.5 Análise Quantitativa

Em seguida, propõe-se partir para uma etapa de análise quantitativa do estado do repositório. A ideia é extrair análises fundamentais sobre a relação entre *feature flags* e *code smells* no código-fonte. Utilizando consultas SQL sobre a base estabelecida, serão realizados cruzamentos dos dados armazenados na tabela, contemplando informações sobre arquivos, *feature flags* e *code smells*.

Inicialmente, a análise buscará quantificar a presença de *flags* nos arquivos, fornecendo uma visão clara da extensão da utilização desses elementos na base de código. Esta métrica será crucial para compreender a distribuição espacial das *feature flags* e sua prevalência em diferentes módulos do projeto.

Em seguida, serão exploradas as relações entre a presença de *flags* e a ocorrência de smells. Será realizada uma avaliação da quantidade de *code smells* em arquivos que contêm *feature flags*, permitindo a identificação de possíveis áreas de concentração de problemas de manutenção associados ao uso dessas *flags*.

A análise também abordará a distribuição das severidades de *code smells* em arquivos que fazem uso de *feature flags*, considerando as categorias do *SonarCloud*:

- ***Blocker:*** Problemas críticos que exigem correção imediata, impedindo a compilação ou causando falhas graves.
- ***Critical:*** Problemas graves que devem ser corrigidos rapidamente, pois podem resultar em falhas ou comportamentos inesperados.
- ***Major:*** Problemas importantes que afetam a qualidade do código e devem ser tratados para garantir a estabilidade e eficiência.
- ***Minor:*** Problemas menores que não impactam diretamente a funcionalidade, mas melhorias são recomendadas para manter boas práticas.
- ***Info:*** Informações ou sugestões para melhorias, sem impacto na qualidade funcional, mas que podem aprimorar a legibilidade ou eficiência.

Este aspecto fornecerá conclusões valiosas sobre a gravidade dos problemas identificados, destacando se a presença de *feature flags* está correlacionada com *code smells* de severidades específicas.

Nessa etapa, fica clara a vantagem do escolha do SQL para o armazenamento das informações, já que a linguagem das consultas permitem construir consultas significativas de maneira rápida e iterativa, ou seja, conforme as análises forem provendo conclusões, novas análises não previstas podem ser escritas sob o mesmo banco de dados.

3.6 Análise Manual

Finalmente, a última etapa do estudo, consolida e contextualiza as conclusões obtidas por meio das análises quantitativas. A abordagem proposta consiste em direcionar a atenção para os principais arquivos e cenários identificados nas etapa anterior, oferecendo uma validação qualitativa dos resultados.

Os arquivos que emergiram como pontos de interesse durante a análise quantitativa serão minuciosamente examinados conforme as correlações estabelecidas entre a presença de *feature flags* e de *code smells*. Esta etapa permitirá uma avaliação mais aprofundada se a causa dos *code smells* foram de fato a presenta da *feature flag* ou não, ajudando a levantar os potenciais impactos negativos da má utilização de *feature flags* na manutenção do código.

Esta abordagem manual acrescenta uma camada de validação crítica, permitindo a identificação de nuances que podem não ter sido plenamente capturadas pelas análises automáticas e quantitativas.

4 Proposta de Solução

4.1 Arquitetura

A proposta de arquitetura da solução executada é fundamentada em um sistema de *scripts* Python3, onde cada *script* desempenha uma função específica na implementação e execução da metodologia definida. Essa abordagem modular permite uma estrutura flexível, onde cada componente é responsável por uma etapa do processo de análise permitindo que sua atualização/execução seja feita de maneira independente, desde que as informações necessárias para sua execução já existam.

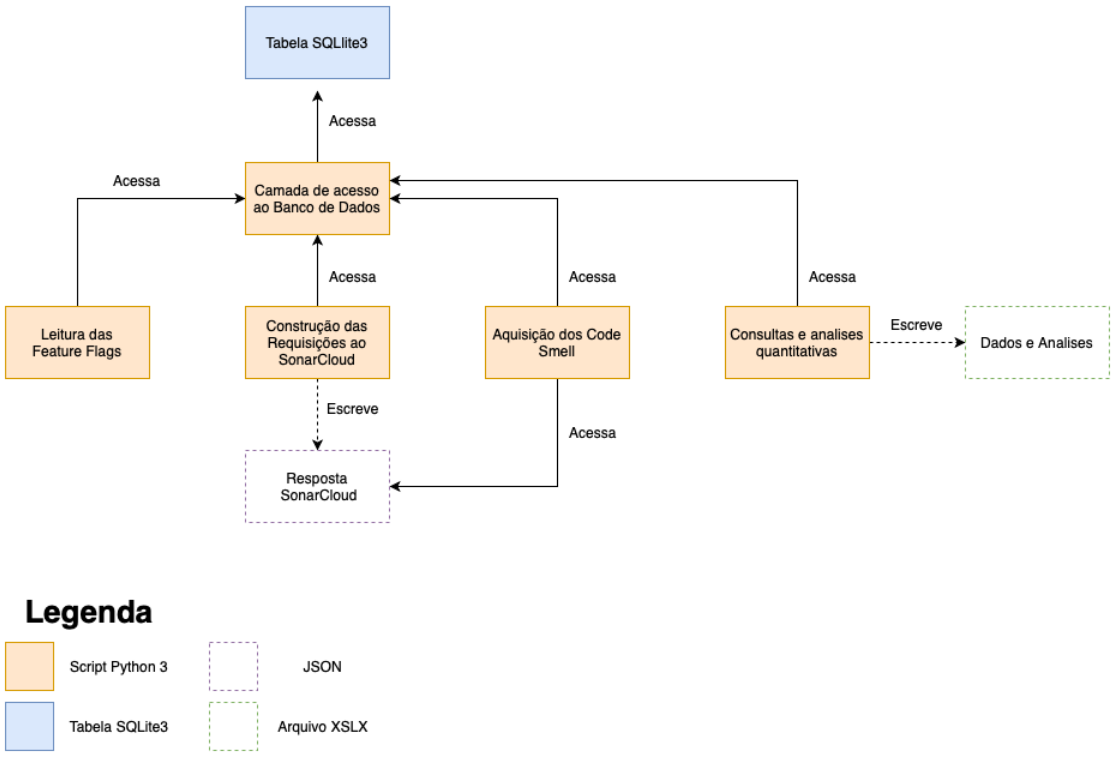

Figura 1: Diagrama arquitetural da solução proposta

Ao lado do *Python3*, foi definido de um banco de dados *SQLite3*, uma escolha baseada na simplicidade e portabilidade do sistema, permitindo armazenar as informações localmente em um arquivo e dispensando a necessidade da construção de todo uma camada de serviço para acesso e hospedagem desse banco. Essa decisão facilita a manipulação e consulta dos dados, essenciais para os processos seguintes.

4.2 Análise de *Code Smells*

Primeiramente, é necessário o *fork* no repositório do cliente *iOS* do *Mozilla Firefox*, estabelecendo uma versão fixada do código-fonte para a aplicação dos métodos propostos. Essa etapa é essencial para preservar a integridade do projeto original, já que o projeto está em constante mudança, enquanto se realiza a análise e implementação da solução.

Após o *fork*, a análise estática do código-fonte é conduzida por meio da integração com o *SonarCloud*. Essa plataforma estabelece uma conexão direta com o repositório no *GitHub*, permitindo a execução automatizada da análise de código. O *SonarCloud* verifica uma variedade de aspectos, desde a estrutura geral do código até detalhes específicos de implementação, identificando possíveis *code smells*, vulnerabilidades e áreas que demandam atenção.

Ao realizar a análise, o *SonarCloud* lista e categoriza as questões encontradas, como *Code Smells*, em um formato acessível e visual no painel da plataforma. Os itens são apresentadas de maneira clara, fornecendo informações detalhadas sobre cada problema identificado, incluindo a descrição do *Code Smell*, sua severidade e a localização exata no código-fonte.

Além disso, todas as informações expostas via plataformas são acessíveis via API aberta, permitindo a construção de *scripts* que acessem e trabalhem essas informações

Ao abranger o repositório inteiro, a análise não se limita apenas aos arquivos que contêm *flags*, mas examina a totalidade do código. Isso permite uma avaliação abrangente da qualidade do código-base, possibilitando a identificação de padrões, tendências e áreas críticas que transcendem o uso específico de *Feature Flags*. Essa abordagem ampla é crucial para traçar comparações significativas, tanto em análises quantitativas quanto qualitativas, e proporciona uma visão holística da integridade e eficiência do código-fonte.

4.3 Mapeamento das *Feature Flags*

Após a análise do código-fonte, a próxima etapa consiste na extração e armazenamento das *feature flags*. O sistema de *feature flags* no repositório em questão é gerenciado por uma classe denominada *LegacyFeatureFlagsManager*, que acessa o serviço de *flags* e verifica seus valores por meio da chamada do método *isFeatureEnabled(featureID:)*.

A *LegacyFeatureFlagsManager* é implementada como um *singleton*, garantindo que todos os objetos do sistema acessem a mesma instância da classe, proporcionando uma implementação centralizada e coesa. Para facilitar as chamadas dessa classe no código, foi criada uma interface, conhecida como *protocols* em *Swift*, denominada *FeatureFlaggable*. Essa interface inclui uma implementação padrão da variável *featureFlags*, do tipo *LegacyFeatureFlagsManager*, assegurando que qualquer classe que a adote tenha acesso fácil ao *singleton* sem a necessidade de referenciá-lo explicitamente, bastando chamar a variável.

Dessa forma, torna-se fácil identificar uma chamada de *feature flag* ao buscar pela *string* *featureFlags.isFeatureEnabled()* ou simplesmente *.isFeatureEnabled()* no código-fonte, conforme apresentado no Algoritmo 1.

```
47 /// Used as the main way to find out whether a feature is active or not, checking
48 /// either just for the build, the build and user preferences, or just user
```

```
/// preferences (supported by Nimbus defaults).
public func isFeatureEnabled(_ featureID: NimbusFeatureFlagID,
                                checking channelsToCheck:
    FlaggableFeatureCheckOptions
) -> Bool {
    let feature = NimbusFlaggableFeature(withID: featureID, and: profile)

    let nimbusSetting = feature.isNimbusEnabled(using: nimbusFlags)
    let userSetting = feature.isUserEnabled(using: nimbusFlags)

    switch channelsToCheck {
    case .buildOnly:
        return nimbusSetting
    case .buildAndUser:
        return nimbusSetting && userSetting
    case .userOnly:
        return userSetting
    }
}
```

Algoritimo 1: Método do serviço de Feature Flags para avaliação dos valores

Para realizar essa busca, optou-se por utilizar expressões regulares (*regex*), escolha motivada pela necessidade de lidar com padrões complexos e variáveis presentes nas chamadas de *feature flags*, e fornecendo uma ferramenta flexível para identificar e extrair informações específicas de *strings*, como no caso das chamadas de *feature flags* no código-fonte.

Com a *regex* definida, foi desenvolvido um *script* que percorre os arquivos em busca das ocorrências. Ao final de sua execução, o *script* gera uma lista de arquivos e a quantidade de *flags* associadas a cada um. Antes de encerrar sua execução, os dados são armazenados em uma tabela *SQLite3*, com uma coluna para os arquivos contendo o caminho relativo em relação à raiz do projeto e outra coluna representando a quantidade de *flags* associadas.

```
regex_patterns = [
    r'\.isFeatureEnabled\('
]

matching_files_with_counts = find_files_with_regex_patterns(repo_path,
    '../trabalho/firefox-ios', regex_patterns, ignored_dirs)

if matching_files_with_counts:
    print(f"{len(matching_files_with_counts)} arquivos correspondentes aos
    padroes de regex:")
    for file_name, total_matches in matching_files_with_counts.items():
        print(f"Arquivo: {file_name}, Total de Correspondencias:
    {total_matches}")
    save_to_sqlite(matching_files_with_counts)
else:
    print("Nenhum arquivo corresponde aos padroes de regex.")
```

Algoritimo 2: Código para mapeamento dos arquvios com flags

É importante ressaltar que essa solução e, mais especificamente, essa expressão regular funciona somente para o repositório analisado, considerando que essa solução visou somente a realização da análise deste caso.

4.4 Cruzamento Feature Flags x Code Smells

Com a listagem dos arquivos e suas respectivas *feature flags* armazenadas na tabela *SQLite3*, foi desenvolvido um *script* que extrai essa lista e constrói uma requisição ao *SonarCloud* para coleta das informações sobre *code smells* nesses arquivos, gerando arquivos JSON que consolidam os resultados da análise estática.

```
def divide_chunks(l, n):
    # looping till length l
    for i in  stringstylerange(0,  stringstylelen(l), n):
        yield l[i:i + n]

def save_json_to_file(data, file_path):
    with  stringstyleopen(file_path, "w") as json_file:
        json.dump(data, json_file, indent=4)

def get_code_smells_from_sonar_cloud(chunks, sonarcloud_url,
    sonarcloud_api_token):
    i = 0

    for files_list in chunks:
        i += 1
        output_file = f"./results/sonarcloud/reponse_{i}.json"

        # Remova os dois componentes mais a raiz do caminho
        file_names = [f"lucasasantana_firefox-ios:{file_path}" for
    file_path in files_list]

        # URL da API do SonarCloud para obter os code smells do arquivo
        keys = ','.join(file_names)
        api_url = f"{sonarcloud_url}/api/issues/search"
        api_url +=  f"?componentKeys={keys}&types=CODE_SMELL"

        # Requisicao à API do SonarCloud
        response = requests.get(api_url, headers={"Authorization": f"Bearer
    {sonarcloud_api_token}"})

        if response.status_code == 200:
            save_json_to_file(response.json(), output_file)
            print(f"Resposta da API do SonarCloud salva em '{output_file}'")

        else:
            print(f"Falha ao obter code smells para o arquivos
    {file_names}. Path: {file_names} Status code: {response.status_code}")
            # Print da mensagem de erro completa
            print(f"Mensagem de erro: {response.text}")
```

Algoritimo 3: Código leitura dos Code Smells via Sonar

Posteriormente, um segundo *script* foi elaborado para ler esses arquivos *JSON* e identificar a quantidade de *code smells* para cada arquivo. Esse processo inclui a agrupamento dos *code smells* por severidade, resultando na criação de colunas correspondentes a cada nível de severidade dos *smells*.

```
def count_code_smells(values: Issues):
    smells = {}
    for issue in values.issues:
        if not (issue. stringstyletype == TypeEnum.CODE_SMELL):
            continue

        component = issue.component.replace('lucasasantana_firefox-ios:',
    '')
        severity = issue.severity.value.lower()

        if not component in smells:
            smells[component] = {case.value.lower(): 0 for case in
    IssueSeverity}

        smells[component][severity] += 1

    return smells
```

Algoritimo 4: Código da leitura JSON da resposta do Sonar

Após a execução dos *scripts*, o que se obtém é uma tabela que junta os dados de nome de arquivo, quantidade de *fetures flags* e *code smells*, estas agrupadas por severidade. Esses resultados permitem a construção dos resultados números que servirão de direcionamento para as analises qualitativas.

4.5 Construção dos Resultados Quantitativos

Para extrair os resultados numéricos necessários para as análises subsequentes, desenvolveu-se mais um *script Python* dedicado. Este *script* acessa a tabela *SQLite3* por meio de consultas *SQL* estrategicamente construídas e compila os números pertinentes ao estudo. As exatas consultas serão abordadas na discussão sobre os experimentos realizados.

```
def calculate_smells_by_severity():
    # Excel file name
    workbook = get_file()
    sheet = create_or_load_sheet('smells_by_severity', workbook)

    # Query to calculate the percentage of files with feature flags
    query_sql = """
    SELECT
        CASE WHEN number_of_flags > 0 THEN 'Possui flags' ELSE 'ãNo possui
    flags' END AS flag_category,
        COUNT(DISTINCT file_name) AS distinct_file_count,
        SUM(blocker) AS blocker_count,
        SUM(critical) AS critical_count,
        SUM(info) AS info_count,
        SUM(major) AS major_count,
        SUM(minor) AS minor_count
    FROM flags
    GROUP BY flag_category;
    """
    (results, _) = fetch_all(query_sql)

```

```
21      # Add the result to the sheet
22      sheet.append(['Flag Status', 'Quantidade de Arquivos', 'blocker',
        'critical', 'info', 'major', 'minor'])
23      for result in results:
24          sheet.append(result)
25
26      # Save the Excel file
27      save(workbook)
```

Algoritimo 5: Exemplo de função para construção de resultados numéricos

O *script* não apenas executa as consultas necessárias, mas também as armazena de forma organizada em um arquivo XLSX. Cada análise é representada em uma aba separada, proporcionando uma visão clara e estruturada dos resultados numéricos obtidos. Isso não apenas simplifica a interpretação dos dados, mas também facilita a referência durante as análises subsequentes.

5 Avaliação Realizada

5.1 Quantidade de arquivos com Feature Flags

A primeira análise realizada foi quantificar a presença de *feature flags* nos arquivos do repositório. Para realizar essa análise, elaborou-se uma *query SQL* que retorna a quantidade de arquivos que contêm *feature flags* e aqueles que não as possuem.

```
# Cria a aba no arquivo excel
workbook = get_file()
sheet = create_or_load_sheet('flags', workbook)

# Conta a quantidade de arquivo com e sem flags
query_sql = """
SELECT
    CASE WHEN number_of_flags > 0 THEN 'Possui flags' ELSE 'ãNo possui
    flags' END AS flag_category,
    COUNT(DISTINCT file_name) AS file_count,
FROM flags
GROUP BY flag_category;
"""
(results, _) = fetch_all(query_sql)

# Adiciona os resultados a planilha
sheet.append(['Flag Status', 'Quantidade de Arquivos'])
for result in results:
    sheet.append(result)

# Salva o resultado
save(workbook)
```

Algoritimo 6: Código para contagem de arquivos

Essa consulta permite uma visão inicial da distribuição das *feature flags* nos arquivos do repositório e estabelece uma base sólida para compreender a extensão do uso desses elementos no código-fonte.

Possui Flags	**Arquivos**
Não possui	610
Possui	37

Tabela 1: Quantidade de arquivos com e sem Feature Flags

Com esse resultado, é observar que as incidência de *flags* não é ampla no código, incidindo apenas em 5,72% dos arquivos.

5.2 Densidade de Code Smells

Após analisar a quantidade de arquivos com e sem *feature flags*, realizou-se um experimento para avaliar a densidade de code smells nos dois grupos. Para isso, foi também calculado o total de *code smells*, independentemente de sua severidade, em ambos os conjuntos de

arquivos. Em seguida, a densidade de *code smells* por arquivo foi obtida pela divisão do número total de *smells* pelo número de arquivos em cada grupo.

```
# Cria a aba no arquivo excel
workbook = get_file()
sheet = create_or_load_sheet('flags', workbook)

# Conta a quantidade de arquivo com e sem flags
query_sql = """
SELECT
    CASE WHEN number_of_flags > 0 THEN 'Possui flags' ELSE 'ãNo possui flags' END AS flag_category,
    COUNT(DISTINCT file_name) AS file_count,
    SUM(blocker + critical + info + major + minor) AS total_code_smells
FROM flags
GROUP BY flag_category;
"""
(results, _) = fetch_all(query_sql)

# Adiciona os resultados a planilha
sheet.append(['Flag Status', 'Quantidade de Arquivos', 'Total Code Smells', 'Densidade de Smells'])
for result in results:
    sheet.append(result)

# Salva o resultado
save(workbook)
```

Algoritimo 7: Código para medição de densidade de *code smells*

A partir dessa consulta, foi possível observar que o total de *Code Smells* foi de 688 nos arquivos sem *flags* e 37 nos arquivos com *flags*. Isso revela uma densidade de *smells* de 1,13 *smells/arquivo* nos arquivos sem *flags* e 3,08 *smells/arquivo* nos arquivos com *flags*.

Possui Flags	**Arquivos**	**Code Smells**	**Smells/Arquivos**
Não possui	610	688	1,13
Possui	37	114	3,08

Tabela 2: Densidade de Code Smells em arquivos com e sem Feature Flags

Os resultados revelaram uma densidade de *code smells* quase três vezes maior nos arquivos com *feature flags* em comparação com os arquivos sem essas *flags*. Essa diferença significativa destaca uma correlação entre a presença de *feature flags* e a ocorrência de *code smells* no código-fonte.

5.3 Densidade de *Code Smells* por Severidade

Em seguida, estende-se a análise agora estudando a densidade de *smells* por severidade. O objetivo por trás dessa análise é entender se a maior incidência de *code smells* acontece por um aumento não apenas de *smells* de severidade menos críticas, já que essas não causam tanto impacto na manutenção do código como nas mais severas.

Severidade	**Possui Flag**	***Code Smells***	***Smells*/Arquivos**
blocker	Não	2	0,003
	Sim	0	0
critical	Não	131	0,215
	Sim	29	0,784
info	Não	16	0,026
	Sim	2	0,054
major	Não	166	0,272
	Sim	56	1,514
minor	Não	373	0,611
	Sim	27	0,730

Tabela 3: Densidade de *Code Smells*, agrupado por severidade, em arquivos com e sem *Feature Flags*.

Podemos observar que categorizando os *code smells*, a densidade de *smells* ainda é maior nos arquivos com *flags* em praticamente todas as categorias, apenas excluindo a categoria *blocker* que é mais severa de todas.

Com isso, pode-se dizer que o aumento da incidência de *smells* nos arquivos que possuem *feature flags* é consistente através das diferentes severidades de *smells*.

5.4 Análise Qualitativa de Arquivos

Com a potencial correlação entre *flags* e *Code Smells* nesse repositório, se faz necessário uma avaliação qualitativa de se os *smells* são de fato causados pelas *features flags*. Para isso, foi listado quais os arquivos com *flags* e mais *code smells* presentes.

O resultado mostra que a incidência de *smells* desse grupo de arquivos está em sua maior parte concentrada nos 5 primeiros arquivos da lista ordenada pela quantidade de *smells*. Além disso, quando olhamos para a severidade dos *smells*, também vemos que esses mesmos arquivos também concentram a incidência de *smells* de maior severidade (*critical* e *major*).

Portanto, é seguro dizer que esses arquivos têm uma participação decisiva na densidade de *smells* observada anteriormente. Se os *smells* nesses arquivos forem ou não causa direta da presença de *feature flags*, poderemos ter uma conclusão sobre se elas impactaram ou não a qualidade e nível de manutenção de código nesse repositório.

5.4.1 Análise Detalhada de *BrowserViewController.swift*

Nesse arquivo, temos duas chamadas do serviço de *flag* no código:

1. *prepareURLOnboardingContextualHint()*

```
610 private func prepareURLOnboardingContextualHint() {
611     guard contextHintVC.shouldPresentHint(),
612             featureFlags.isFeatureEnabled(.contextualHintForToolbar,
    checking: .buildOnly)
```

Número	Arquivo	Flags	Smells	Critical	Major
1	BrowserViewController.swift	2	32	13	15
2	LegacyTabManager.swift	1	21	8	10
3	SettingsTableViewController.swift	2	13	0	2
4	HistoryPanel.swift	1	11	3	1
5	TabDisplayManager.swift	2	10	1	7
6	SearchViewController.swift	1	6	1	5
7	MainMenuActionHelper.swift	4	5	1	3
8	HistoryHighlightsViewModel.swift	1	4	0	4
9	JumpBackInViewModel.swift	1	3	0	3
10	TelemetryWrapper.swift	4	3	2	1
11	HistoryPanelViewModel.swift	1	2	0	1
12	HomePageSettingViewController.swift	7	1	0	1
13	JumpBackInDataAdaptor.swift	1	1	0	1
14	TabScrollController.swift	1	1	0	1
15	ShareExtensionHelper.swift	1	1	0	1

Tabela 4: Listagem de Arquivos com Smells

```
613         else { return }
614
615         contextHintVC.configure(
616             anchor: urlBar,
617             withArrowDirection: isBottomSearchBar ? .down : .up,
618             andDelegate: self,
619             presentedUsing: { self.presentContextualHint() },
620             andActionForButton: {
        self.homePanelDidRequestToOpenSettings(at: .customizeToolbar) },
621             overlayState: overlayManager)
622     }
```

Algoritimo 8: Trecho com uso de flag no BrowserViewController.swift

2. *creditCardAutofillSetup()*

```
1709 private func creditCardAutofillSetup(_ tab: Tab, didCreateWebView
        webView: WKWebView) {
1710     let userDefaults = UserDefaults.standard
1711     let keyCreditCardAutofill = PrefsKeys.KeyAutofillCreditCardStatus
1712
1713     let autofillCreditCardStatus = featureFlags.isFeatureEnabled(
1714         .creditCardAutofillStatus, checking: .buildOnly)
1715     if autofillCreditCardStatus {
1716         let creditCardHelper = CreditCardHelper(tab: tab)
1717         tab.addContentScript(creditCardHelper, name:
        CreditCardHelper.name())
1718         creditCardHelper.foundFieldValues = { [weak self] fieldValues,
        type, frame in
1719             guard let tabWebView = tab.webView,
1720                   let type = type,
```

```
1721                    userDefaults.object(forKey: keyCreditCardAutofill)
    as? Bool ?? true
1722            else { return }
1723
1724            switch type {
1725            case .formInput:
1726                TelemetryWrapper.recordEvent(category: .action,
1727                                             method: .tap,
1728                                             object:
    .creditCardFormDetected)
1729                self?.profile.autofill.listCreditCards(completion: {
    cards, error in
1730                    guard let cards = cards, !cards.isEmpty, error ==
    nil
1731                    else {
1732                        return
1733                    }
1734                    DispatchQueue.main.async {
1735
    tabWebView.accessoryView.reloadViewFor(.creditCard)
1736                        tabWebView.reloadInputViews()
1737                    }
1738                })
1739            case .formSubmit:
1740                self?.showCreditCardAutofillSheet(fieldValues:
    fieldValues)
1741                break
1742            }
1743
1744            tabWebView.accessoryView.savedCardsClosure = {
1745                DispatchQueue.main.async { [weak self] in
1746                    // Dismiss keyboard
1747                    webView.resignFirstResponder()
1748                    // Authenticate and show bottom sheet with select
    a card flow
1749
    self?.authenticateSelectCreditCardBottomSheet(fieldValues:
    fieldValues,
1750
    frame: frame)
1751                }
1752            }
1753        }
1754    }
1755 }
```

Algoritimo 9: Trecho com uso de flag no BrowserViewController.swift

Ao delimitar as linhas 610-622 e 1709-1755, pode-se realizar uma busca por *Code Smells* nas linhas específicas do arquivo, baseando-se na na estrutura da resposta fornecida pelo *SonarCloud*. Ao analisar o *JSON* retornado, podemos extrair informações detalhadas sobre cada *Code Smell*, incluindo a localização exata no código-fonte já que *SonarCloud* identifica a posição de cada *Code Smell* no arquivo, indicando a linha específica em que ocorre.

Portanto, construindo um *script* em *Python* que lê as respostas em *JSON* do *SonarCloud* ainda armazenadas, podemos encontrar *Code Smells* nessas funções.

```
# Carregando o JSON
data = load_data_from_json(json_file_paths)

# Componente analisado
component = "Client/Frontend/Browser/BrowserViewController.swift"

# Range das funcoes com feature flags
list_of_ranges = [(610, 622), (1709, 1755)]

# Filtrando issues com base no componente e no range
filtered_issues = [issue for issue in data["issues"] if (
    issue["component"] == component and
     stringstyleany(
        lin_min <= issue["textRange"]["startLine"] <= lin_max and
        lin_min <= issue["textRange"]["endLine"] <= lin_max
        for lin_min, lin_max in list_of_ranges
    )
)]

# Estrutura de dados para contagem das issues por severidade
severity_count = {
    "MAJOR": 0,
    "MINOR": 0,
    "INFO": 0,
    "CRITICAL": 0,
    "BLOCKER": 0
}

for issue in filtered_issues:
    severity = issue["severity"]
    severity_count[severity] += 1

# Printing the results
print(f"Quantidade de Smells Encontrados: {len(filtered_issues)}")
print("Quantidade por severidade:")
for severity, count in severity_count.items():
    print(f"{severity}: {count}")
```

Algoritimo 10: Código para Contagem

Ao executar o *script*, obteve-se a seguinte resposta:

```
Quantidade de Smells Encontrados: 0
Quantidade por severidade:
MAJOR: 0
MINOR: 0
INFO: 0
CRITICAL: 0
BLOCKER: 0
```

Algoritimo 11: Resposta da execução do código Python

Portanto, dos 32 *Code Smells* encontrados no arquivo, nenhum deles ocorreu na função onde se encontram as *flags*. Além disso, as *flags* estão sendo utilizadas em locais de validação (um *if* e um *guard*) utilizados de maneira localizada, dessa forma nenhum *smell* secundário (em outro trecho do mesmo código) pode ter sido gerado por essa utilização. Portanto, é seguro dizer que nesse arquivo as *flags* não são a causa dos *Code Smells*.

5.4.2 Análise de *LegacyTabManager.swift*

Nesse arquivo, temos apenas uma chamadas do serviço de *flag* no código:

```
825 private func viableTabs(isPrivate: Bool = false) -> [Tab] {
826     if !isPrivate, featureFlags.isFeatureEnabled(.inactiveTabs, checking:
    .buildAndUser) {
827         // only use active tabs as viable tabs
828         // we cannot use recentlyAccessedNormalTabs as this is filtering
    for sponsored and sorting tabs
829         return InactiveTabViewModel.getActiveEligibleTabsFrom(normalTabs,
    profile: profile)
830     } else {
831         return isPrivate ? privateTabs : normalTabs
832     }
833 }
```

Algoritimo 12: Trecho com uso de flag no LegacyTabManager.swift

Executando o mesmo script para esse componente e conjunto de linhas temos o seguinte resultado

```
Quantidade de Smells Encontrados: 0
Quantidade por severidade:
MAJOR: 0
MINOR: 0
INFO: 0
CRITICAL: 0
BLOCKER: 0
```

Algoritimo 13: Resposta da execução do código Python

Com isso vemos que as *flags* também não são a causa dos *Smells* nesse arquivo.

5.4.3 Análise de *SettingsTableViewController.swift*

Nesse arquivo, temos duas chamadas do serviço de *flag* no código:

1. *displayBool()*

```
343 func displayBool(_ control: UISwitch) {
344     if let featureFlagName = featureFlagName {
345         control.isOn = featureFlags.isFeatureEnabled(featureFlagName,
    checking: .userOnly)
346     } else {
347         guard let key = prefKey, let defaultValue = defaultValue else
    { return }
348         control.isOn = prefs?.boolForKey(key) ?? defaultValue
```

```
349     }
350 }
```

Algoritimo 14: Trecho com uso de *flag* no SettingsTableViewController.swift

2. *displayBool()*

```
358 override func displayBool(_ control: UISwitch) {
359     if let featureFlagName = featureFlagName {
360         control.isOn = featureFlags.isFeatureEnabled(featureFlagName,
    checking: .userOnly)
361     } else {
362         guard let key = prefKey, let defaultValue = defaultValue else
    { return }
363 
364         Task { @MainActor in
365             let isSystemNotificationOn = await isSystemNotificationOn()
366             control.isOn = (userDefaults?.bool(forKey: key) ??
    defaultValue) && isSystemNotificationOn
367         }
368     }
369 }
```

Algoritimo 15: Trecho com uso de *flag* no SettingsTableViewController.swift

Executando o mesmo *script* para esse componente e conjunto de linhas temos o seguinte resultado

```
Quantidade de Smells Encontrados: 0
Quantidade por severidade:
MAJOR: 0
MINOR: 0
INFO: 0
CRITICAL: 0
BLOCKER: 0
```

Algoritimo 16: Resposta da execução do código Python

Com isso vemos que as *flags* também não são a causa dos *Smells* nesse arquivo.

5.4.4 Análise de *HistoryPanel.swift*

Nesse arquivo, temos uma chamada do serviço de *flag* no código:

```
56 var shouldShowSearch: Bool {
57     guard viewModel.featureFlags.isFeatureEnabled(.historyGroups, checking:
    .buildOnly) else {
58         return false
59     }
60 
61     return state == .history(state: .mainView) || state == .history(state:
    .search)
62 }
```

Algoritimo 17: Trecho com uso de *flag* no HistoryPanel.swift

Executando o mesmo *script* para esse componente e conjunto de linhas temos o seguinte resultado

```
Quantidade de Smells Encontrados: 0
Quantidade por severidade:
MAJOR: 0
MINOR: 0
INFO: 0
CRITICAL: 0
BLOCKER: 0
```

Algoritimo 18: Resposta da execução do código Python

Com isso vemos que as *flags* também não são a causa dos *Smells* nesse arquivo.

5.4.5 Análise de *TabDisplayManager.swift*

Nesse arquivo, temos das chamadas do serviço de *flag* no código:

```
92  var shouldEnableGroupedTabs: Bool {
93      return featureFlags.isFeatureEnabled(.tabTrayGroups, checking:
        .buildAndUser)
94  }
95
96  var shouldEnableInactiveTabs: Bool {
97      return featureFlags.isFeatureEnabled(.inactiveTabs, checking:
        .buildAndUser)
98  }
```

Algoritimo 19: Trecho com uso de *flag* no TabDisplayManager.swift

Executando o mesmo *script* para esse componente e conjunto de linhas temos o seguinte resultado

```
Quantidade de Smells Encontrados: 0
Quantidade por severidade:
MAJOR: 0
MINOR: 0
INFO: 0
CRITICAL: 0
BLOCKER: 0
```

Algoritimo 20: Resposta da execução do código Python

Com isso vemos que as *flags* também não são a causa dos *Smells* nesse arquivo.

5.4.6 Análise de *HomePageSettingViewController.swift*

A análise desse arquivo não consiste em fazer a correlação entre as *flags* e os seus *Code Smells*, mas sim em levantar que esse arquivo possui a chamada ao serviço de *flags* em 7 pontos de seu código, sendo o arquivo com maior incidência.

Se a correlação entre *Code Smells* e *Feature Flags* fosse causada pela causalidade entre os dois itens, esse arquivo deveria possuir o maior ou um dos maiores índices de *Code Smells*. Porém, ele apenas possui 1 *Code Smell*, o que, junto com a não causalidade dos outros

arquivos, reforça que as *Feature Flags* não estão impactando na incidência de *Code Smells* nesse repositório.

6 Conclusão

Após uma análise dos arquivos do repositório, com foco nas *features flags* e sua relação com *code smells*, algumas conclusões importantes podem ser destacadas:

1. **Maior Incidência de *Code Smells* em Arquivos com *Flags***

   Foi observado que os arquivos que continham *features flags* apresentaram uma maior incidência de *code smells* em comparação com aqueles sem *flags*. Isso sugeriu uma possível correlação entre a presença de *flags* e a ocorrência de *code smells*.

2. ***Flags* não Causaram *Code Smells* Diretamente**

   Porém, ao se imergir em uma análise qualitativa, indicou que as *flags* em si não são a causa direta dos *code smells* nos arquivos. Um dos arquivos que mais possui *feature flags*, o *HomePageSettingViewController.swift*, apenas tem 1 *Code Smell*.

3. **Arquivos com Mais *Code Smells* devido ao Core do Aplicativo**

   Os arquivos que fazem parte do core do funcionamento do aplicativo, contendo mais código e regras de negócio, apresentaram uma maior quantidade de *code smells*. Isso sugere que a complexidade inerente às funcionalidades essenciais do aplicativo contribui significativamente para a presença de *code smells*.

Diante dessas observações, a conclusão geral é a seguinte: **nesse repositório, *Features Flags* não impactaram significativamente na manutenção do código**. Apesar da maior incidência de *code smells* em arquivos com *features flags*, não se pode afirmar que as *flags* foram a principal causa desses problemas. Outros fatores, como a complexidade das funcionalidades centrais, parecem ter contribuído de maneira mais substancial para a presença de *code smells*.

A conclusão apresentada não é conclusiva para determinar se *features flags*, em geral, impactam ou não na manutenção do código. A análise se baseou em um único repositório, e a generalização para outros contextos requereria estudos adicionais em diferentes ambientes e projetos.

Referências

[1] Martin Fowler and Kent Beck. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*, 1997.

[2] Rezvan Mahdavi-Hezaveh, Nirav Ajmeri, and Laurie Williams. Feature toggles as code: Heuristics and metrics for structuring feature toggles. *Information and Software Technology*, 145:106813, 2022.

[3] Jens Meinicke, Chu-Pan Wong, Bogdan Vasilescu, and Christian Kästner. Exploring differences and commonalities between feature flags and configuration options. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, pages 233–242, 2020.