IFFT: Uma ferramenta de alerta de código para alterações coordenadas

Aluno: Thiago Henrique Moreira Santos
Orientador: André Hora

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

thiago.santos@dcc.ufmg.br

Abstract. *Este relatório consiste na apresentação de detalhes da proposta e da implementação da ferramenta IFFT (IF-Flow-Then), uma ferramenta que auxilia os desenvolvedores a assegurarem que mudanças realizadas em certos blocos de código sejam sempre acompanhandas de modificações correspondentes nos arquivos associados. O aspecto técnico compreende uma lógica de "parsing"do código que detecta mudanças em blocos IFFT e a geração do aviso para o desenvolvedor, no caso de uma mudança não ser identificada no arquivo associado. Também foi implementado uma versão do código automática que é integrada ao repositório do projeto através do uso de "hooks"do Git.*

1. Introdução

1.1. Caracterização do Problema e Motivação

Em grandes projetos de software, onde múltiplas equipes trabalham em diferentes partes da "codebase", é comum haver interdependências entre arquivos de código. Manter o aspecto da consistência nessas interdependências é crucial para garantir a integridade do sistema no longo prazo. A falta de uma maneira eficaz para verificar essas mudanças pode levar a erros críticos (muitas vezes difíceis de detectar) e falhas no sistema.

1.2. Objetivos

Dado a importância do problema, a proposta do IFFT é de ser um "linter"que ajude os desenvolvedores nesse processo de identificar quando uma alteração em um bloco de código exige uma modificação correspondente em outro arquivo. Os principais objetivos do trabalho são:

- Desenvolver uma ferramenta que verifica as mudanças realizadas dentro de um commit e que alerte o desenvolvedor que determinada seção de código que ele editou tem relação com um outro código que esteja em outro local de modo as mudanças dele possívelmente precisam ser refletidas lá também.
- Integrar essa ferramenta com o "git hooks"para permitir um conforto ainda maior para os desenvolvedores, de modo que eles nem sequer precisem executar o linter, ele funcionará automaticamente de forma integrada com o Git toda vez que houver uma tentativa de commit.
- Permitir que a ferramenta seja configurável para se adequar as especificidades de cada projeto.

1.3. Estrutura do Trabalho

A partir deste ponto, o relatório seguirá com as seguintes seções:

IFFT Uma seção com detalhes da ferramenta e do seu funcionamento
Implementação Detalhes técnicos de organização e implementação da ferramenta
Conclusão e Sequência do Trabalho Conclusão e ideias para a próxima iteração

2. IFFT

Há alguns pontos que são importantes de serem elucidados em relação aos objetivos do IFFT. Primeiramente, **ele não tem como objetivo substituir práticas de testes e de refactoring**. O IFFT não foi pensado para ser usado em sincronizações triviais, por exemplo, manter um contador sincronizado entre dois arquivos. Para casos como esse, é recomendado fazer refatorações, o IFFT é mais indicado para casos em que mudanças em um arquivo precisem ser refletidas em outro, mas que essas mudanças sejam não triviais.

O IFFT é mais útil em **cenários onde o esforço de engenharia para escrever testes é muito maior do que o benefício que seria gerado por eles**. Deste modo, o IFTT também não é indicado para ser usado em partes críticas da lógica de negócio, pois nesses locais, é mais indicado ”pagar”o esforço de engenhaira para a testabilidade.

3. Implementação

3.1. Sintaxe do IFFT

Tendo esclarecido esses dois pontos acima, vamos seguir para mostrar mais sobre a ferramenta em si. A imagem a seguir mostra um pseudocódigo da sintaxe adotada para a ferramenta. Um bloco de código que queira adotar o suporte da ferramenta para mudanças sincronizadas deve estar dentro de um **bloco IFFT** que é definido a partir de um comentário na linguagem python como a ”Figura 1”mostra.

3.1.1. Melhoria adotada

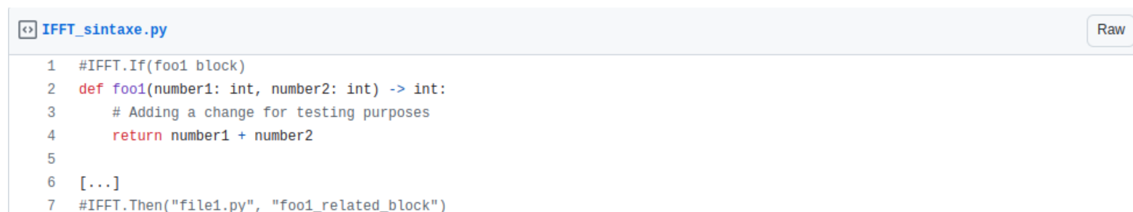

Figura 1. Sintaxe do IFFT

Inicialmente, a ideia para essa sintaxe adotada seria de que o parser teria que tentar encontrar essas declarações de abertura e fechamento dos blocos com um ”match perfeito”das strings. Quando comecei a fazer alguns testes percebi que essa abordagem poderia facilmente gerar erros pelo fato de que o desenvolvedor poderia escrever os comentários de forma levemente diferente da que eu especifiquei, por exemplo, colocando um espaço extra depois do caracter que marca o início do comentário em Python. A melhoria que eu fiz nessa parte foi, ao invés de tentar dar um ”match”exato no parser, passei

a usar uma **expressão regular** que dá muito mais liberdade para o desenvolvedor escrever as aberturas e fechamentos dos blocos IFFT. A ”Imagem 2”mostra como essa mudança foi feita no código.

ifft-sintaxe-improvement.py   Raw

```python
# Antes:
for line_number, line in enumerate(lines):
      if line.strip().startswith("#IFFT.If"):
          [...]

      elif line.strip().startswith("#IFFT.Then"):
          [...]
# Depois:
ifft_if_pattern = re.compile(r'#\s*IFFT\.If', re.IGNORECASE)
ifft_then_pattern = re.compile(r'#\s*IFFT\.Then\(\s*"([^"]+)"\s*,\s*"([^"]+)"\s*\)', re.IGNORECASE)
```

Figura 2. Melhoria no matching dos blocos

3.2. IFFT Parser

A lógica de ”parsing”da ferramenta funciona da seguinte forma: uma vez que o programa é executado manualmente ou no modo automático (que será mencionado na subsessão de melhorias) um script é executado dentro do diretório do projeto e da início à lógica de parsing. Para tornar esse processo mais eficiente, eu faço um passo prévio de identificar quais arquivos foram modificados até o momento pelo usuário, isso é feito com o método **get_modified_files** (Figura 3). Note que, apesar de simples, no caso do modo automático essa ação é não trivial e por isso no modo automático esse método é implementado como uma subrotina que é executada antes do commit. Essa diferença foi a explicação de um problema muito incômodo que tive no desenvolvimento do projeto no modo automático, quando a pessoa fazia um commit a ferramenta simplesmente terminava sua execução. Depois de horas debugando, descobri que a ação de commit estava sendo realizada antes da verificação automática que implementei, e quando ela rodava já não haviam mais arquivos na ”unstaged”área, porque os arquivos já tinha sido comitados. Por isso além do método mais trivial de simplesmente checar os arquivos que tinham diffs, tive que implementar uma solução mais robusta (solução da figura 3) que especifica uma subrotina.

get-modified-files.py   Raw

```python
def get_modified_files():
    result = subprocess.run(['git', 'diff', '--cached', '--name-only', '--diff-filter=ACM'], capture_output=True, text=True)
    return result.stdout.splitlines()
```

Figura 3. Método auxiliar que faz o primeiro filtro de arquivos

Tendo os arquivos modificados em mãos, podemos passar para o próximo nível de granularidade que é checar as linhas, vamos gerar um conjunto com todas as linhas modificadas em cada arquivo. Na verdade, esse conjunto tem um mapeamento do arquivo para as linhas modificadas, isso para cada arquivo que passou no primeiro filtro (de ter sofrido uma modificação), isso é feito pelo método **get_modified_lines** (Figura 4).

E por fim, o passo final (Figura 5) é passar esse enorme conjunto para um outro método chamado **scan_file** que será responsável por encontrar as linhas que foram modificadas dentro de blocos IFFT e construir uma estrutura que contém informações como, conteúdo do bloco, arquivo associado, rótulo do bloco no arquivo associado e conteúdo que foi modificado dentro do bloco.

get-modified-lines.py

```python
def get_modified_lines(repo: str, filename: str, auto_mode: argparse.Namespace) -> set:
    modified_lines = set()
    diff_text = ""

    if not auto_mode:
        diff_text = repo.git.diff(None, filename)

    else:
        diff_text = repo.git.diff('HEAD', filename)

    diff_lines = diff_text.split('\n')
    line_number = 0

    for line in diff_lines:
        if line.startswith('@@'):
            line_number = int(line.split()[2].split(',')[0].replace('+', ''))
        elif line.startswith('+') and not line.startswith('+++'):
            modified_lines.add((line_number, line[1:].strip()))
            line_number += 1
        elif line.startswith(' '):
            line_number += 1

    return modified_lines
```

Figura 4. Obtém as linhas modificadas dentro dos arquivos que sofreram alguma modificação

scan-files.py

```python
def scan_files(project_path: str = dir_path_mock_project, auto_mode: argparse.Namespace = None) -> dict:
    results_dict = {}
    try:
        repo = Repo(project_path)
    except NoSuchPathError:
        return results_dict
    except InvalidGitRepositoryError:
        return results_dict
    except Exception as e:
        return results_dict

    modified_files = []
    if not auto_mode:
        modified_files = [item.a_path for item in repo.index.diff(None)]
    else:
        modified_files = get_modified_files()

    for filename in modified_files:
        if filename.endswith(".py"):
            modified_lines_set = get_modified_lines(repo, filename, auto_mode)
            results_dict[filename] = scan_file(project_path, filename, modified_lines_set)

    return results_dict
```

Figura 5. Gera uma estrutura com todos os dados finais de interesse, em especial, as linhas modificada, que estão dentro de um bloco IFFT

Existe um passo intermediário em que o programa verifica se o arquivo associado existe, mas resolvi omitir esta etapa da descrição anterior visando focar nas partes mais importantes. Depois disso, a ferramenta exibe na saída padrão as informações com formato a depender se o desenvolvedor fez a mudança no arquivo associado ou não, no primeiro caso, nenhuma recomendação é feita para o programador, caso contrário, o programa faz o alerta de que uma edição foi feita dentro de um bloco IFFT e que portanto o programador deve checar se deve fazer uma mudança no arquivo associado.

3.3. IFFT Automode

Ainda dentro dos aspectos de implementação, ao fim do projeto, percebi que ainda havia margem para melhorar a experiência do programador com a ferramenta. Na forma nativa da ferramenta, o usuário deveria ter que executar manualmente a ferramenta pelo terminal (que foi a única superfície que fiz para essa primeira parte) todas as vezes que ele quisesse fazer essa verificação. Foi então que pensei que talvez fosse interessante fazer essa ação de forma automática, fazendo com que a verificação seja feita toda vez que um commit

for feito. Fazendo dessa forma, o programador não precisaria ser preocupar com nada e poderia seguir sua rotina como de costume.

3.3.1. O que são os Hooks no Git

Hooks no Git são scripts que executam automáticamente em momentos específicos do ciclo de funcionamento de um repositório que usa git. Eles permitem que ações personalizadas sejam realizadas em resposta a eventos específicos, como a **criação de um commit** ou uma ação de **push** para o repositório remoto. Existem dois tipos principais de Hooks:

Hooks do lado do cliente são usados principalmente para tarefas de verificação, como validar a mensagem de commit ou verificar se todos os testes passam antes de permitir um commit.

Hooks do lado do servidor são usados para impor políticas de repositório, como garantir que os commits sigam determinado padrão antes de serem aceitos no repositório remoto.

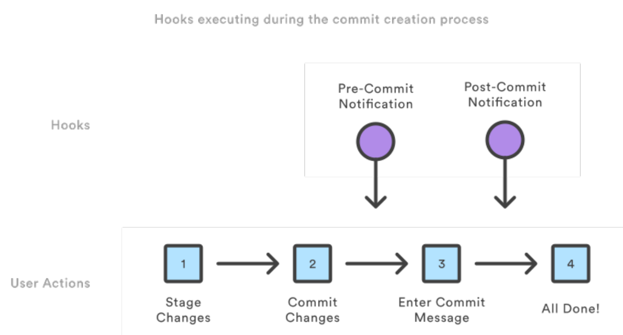

Figura 6. Diagrama dos Hooks do Git

3.3.2. Integração do IFFT com Hooks do Git

Para automatizar a execução do IFFT, integrei a ferramenta com o **Hook de pre-commit** do Git. O Hook de pre-commit é um script que é executado antes que um commit seja criado. Se o script retornar um código de saída diferente de zero, o commit será abortado, permitindo a execução de verificações e validações antes que o commit seja efetuado de fato.

Em termos de implementação propriamente dito, tive que criar um script no formato que os pre-commit são feitos e o que esse script faz é abortar a ação de commit instantaneamente se forem detectadas mudanças em blocos IFFT que não foram refletidas nos arquivos associados, em outras palavras, se a saída do IFFT recomendar alguma edição.

Criei um template do arquivo de pré-commit para facilitar o trabalho de configuração para o desenvolvedor. Tudo que ele terá que fazer é trocar alguns poucos campos para referenciar o projeto dele e a configuração estará quase pronta. O único passo

que vai restar é ele ativar a opção do ”modo automático”num arquivo de configuração que criei para a ferramenta. O arquivo de configuração se chama **ifft_config.json**, atualmente ele tem apenas a config *”auto_mode”* que pode ser definida como ”true”caso o desenvolvedor queira ligar o modo automático, mas para o futuro pretendo colocar outros parâmetros para deixar a ferramenta mais personalizável. Depois de definida essa configuração, está tudo pronto e desenvolvedor pode seguir sua rotina de desenvolvimento normalmente e sempre que ele for fazer um commit a ferramenta será executada préviamente de forma automática.

4. Boas práticas de desenvolvimento de software

Um dos objetivos desse projeto mais técnico que idealizei foi o de praticar aspectos e atributos que são importantes na carreira de um engenheiro de software, muitos deles foram coisas que aprendi durante a graduação e alguns outros são aspectos mais genéricos, mas que ainda assim são indispensáveis. Nessa sessão vou comentar de forma mais detalhada sobre esses aspectos.

4.0.1. Criação de um repositório para o desenvolvimento do projeto

Todo o desenvolvimento do projeto está sendo feito utilizando-se de um serviço de repositórios remoto (Github) que utiliza um **sistema de controle de versão** chamado Git. Isso traz diversos benefícios como a facilidade de acessar diferentes versões do código (recurso que precisei recorrer por algumas vezes durante o desenvolvimento), permite o desenvolvimento de forma incremental dado que posso sempre me concentrar em apenas uma parte específica do código em cada commit e, por fim, pode me ajudar a ter ajuda da comunidade de software livre para continuar desenvolvendo a ferramenta.

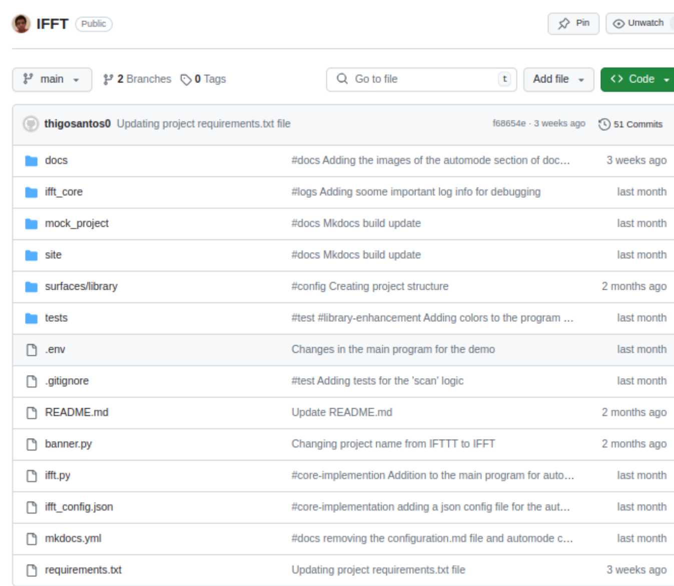

Figura 7. Imagem do repositório remoto utilizado para o projeto

4.0.2. Página de issues

A página de issues foi um grande diferencial nesse trabalho em termos organizacionais. Lá pude manter todas as ideias novas que ia tendo na medida em que trabalhava no desenvolvimento da ferramenta e, principalmente, pude registrar os problemas, os chamados **bugs**, que iam surgindo na medida que a ferramenta ia aumentando de complexidade.

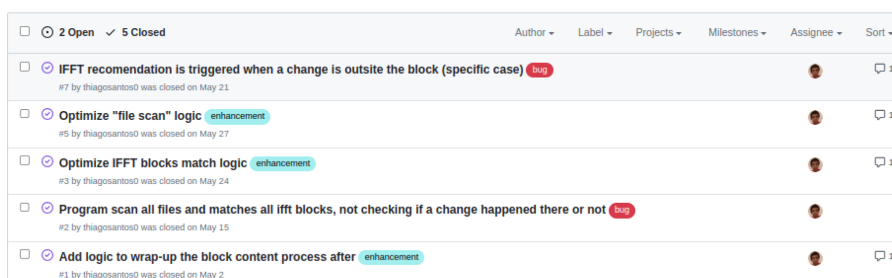

Figura 8. Issues resolvidas durante o desenvolvimento

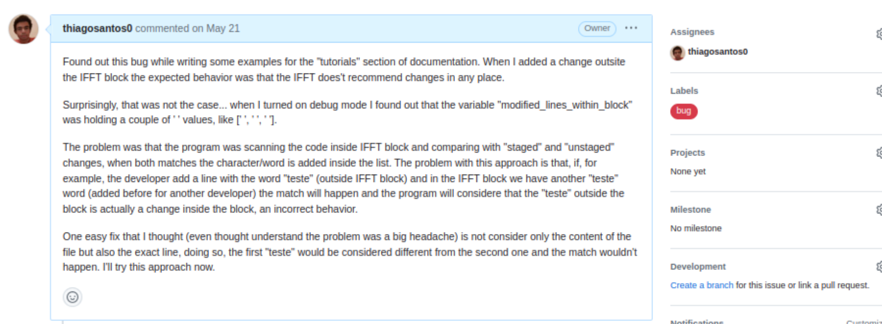

Figura 9. Descrição de um bug encontrado e como ele foi solucionado

4.1. Aplicação da ideia proposta pelo Manifesto Ágil

Apesar de ter tido várias ideias de ”features”para a ferramenta, optei por utilizar de uma das principais ideias do Manifesto Ágil, que é a da rápida prototipação e validação da ideia de forma geral. Deste modo, meu foco nessa primeira parte do projeto foi fazer a ferramenta funcionar em apenas um projeto específico chamado de **mock project**. Esta foi uma simulação de um projeto que usaria o IFFT que criei, muito mais simplificado do que seria um projeto de verdade e então meu foco foi construir a ferramenta de modo que atendesse esse projeto simples e que eu pudesse validar a ideia. Depois de averiguar a viabilidade e potencial do protótipo, fica mais simples começar a trabalhar em expansões como suporte a outras superfícies como extensões para editores de texto, atrelar ele ao pipeline de CI/CD do Github actions e também features novas como a possibilidade de escrever blocos IFFT aninhados. Esses e outros planos que tenho para a ferramenta serão comentados com mais detalhes na seção de ”Conclusão e Sequência do Trabalho”.

4.2. Documentação

Outro aspecto que dei bastante importância nesse projeto, que é comum em ser feito em projetos sérios de desenvolvimento de software, é o aspecto da documentação. Tentei tomar um grande cuidado para documentar bem os métodos que foram escritos, fornecendo

descrições detalhadas do funcionamento, exemplos de entrada e saída, também reservei sessões na documentação onde é mostrado alguns casos de usos da ferramenta tanto no modo manual quanto no modo automático da ferramenta.

Para escrever a página de documentação do trabalho, fiz o uso de uma biblioteca muito interessante chamada **mkdocs**, ela tem algumas features bem interessantes e uma fácil integração com a linguagem Python. Uma dessas features interessantes é a de criação facilitada das referências para os métodos, tudo que precisei fazer foi documentar os métodos logo após as assinaturas, usando **docstrings** e isso era automáticamente redirecionado para a página de referência da documentação, facilitando bastante o processo de escrita dela. Por fim, foi feito o **deploy** da página de documentação e atualmente ela pode ser acessada via este **link**.

Reference

This part of the project documentation focuses on an **information-oriented** approach. You can find all helper methods defined by the "ifft_core" module here. You'll find detailed information about the method signatures, return types, exceptions, and examples.

The module contains the following functions:

- `get_modified_lines(repo, filename) - Get a set of modified lines for the given filename.`
- `scan_file(project_path, filename, modified_lines_set) - Scan the file for IFFT blocks and return the results.`
- `scan_files(project_path, dir_path_mock_project) - Scan the repository for modified Python files and return the results in a dictionary.`
- `validate_associated_file(associated_file_name)` - Validade if the associated file specified in IFFT block exists.

**Note: ** For the examples in this documentation, the plus sign (+) indicates a modified line.

Table of contents

**Note: ** For the examples in this documentation, the plus sign (+) indicates a modified line.
ifft_parser
get_modified_lines
scan_file
scan_files
validate_associated_file

`get_modified_lines(repo, filename)`

Get a set of modified lines for the given filename.

Example

Parameters:

Name	Type	Description	Default
`repo`	`str`	A string corresponding to the repository.	*required*
`filename`	`str`	A string corresponding to the filename.	*required*

Returns:

Name	Type	Description
`set`	`set`	A set of modified lines.

Source code in `ifft_core/ifft_parser.py`

`scan_file(project_path, filename, modified_lines_set)`

Scan the file for IFFT blocks and return the results.

Example

Parameters:

Figura 10. Imagem da seção de referência da documentação com os métodos

4.3. Testes

Um aspecto importante da engenharia de software que tentei trazer para este trabalho foram os testes. Apesar de não ter escrito muitos deles, escrevi testes que me ajudaram muito no desenvolvimento da parte crítica da aplicação, que é o parser. Inclusive, alguns dos bugs que estão listados na página de issues só puderam ser encontrados pelos testes dado que muitas vezes isso não poderia ser percebido durante um teste manual do sistema. E mesmo nos casos em que o problema era percebido, ficava mais difícil o processo de depuração.

4.3.1. Projeto escrito completamente em língua inglesa

Por fim, um aspecto que considero de menor importância, mas ainda assim relevante, foi que escrevi o projeto todo em língua inglesa. A língua inglesa é amplamente utilizada quando se pensa no cenário global de engenharia de software, projetos de software livre são majoritariamente escritos em inglês e portanto, ter prática com este idioma nos abre possibilidades de contribuições no futuro, sem contar que grandes empresas normalmente terão times globais, com pessoas trabalhando de todas as partes do mundo e nesses casos, o inglês é normalmente usado como *proxy* para a colaboração de modo geral.

5. Conclusão e Sequência do Trabalho

Em termos dos objetivos que tracei na proposta do trabalho, a maior parte deles foram alcançados com sucesso. No estado atual, a ferramenta está funcional, apesar de ser apenas para um projeto específico, que era o objetivo principal definido e eu já consigo mostrar o funcionamento dela, o que será feito em uma **demonstração** no vídeo de apresentação da ferramenta, uma vez que a apresentação esteja concluída, ela estará disponível neste **link**. E meu outro grande objetivo também foi alcançando com sucesso que era de exercitar os conceitos que aprendi durante a graduação, em especial, das matérias relacionadas ao âmbito da Engenharia de Software, que é minha área de maior interesse.

Estou muito satisfeito com o caminho que o projeto está tomando e por isso pretendo continuá-lo na segunda parte do trabalho de conclusão de curso. Dentre as várias coisas que penso em fazer para a segunda iteração de desenvolvimento da ferramenta se destacam:

5.1. Generalização do suporte da ferramenta

Como foi dito anteriormente, para que eu pudesse validar a ideia, primeiramente construi a ferramenta para funcionar apenas em um projeto específico de testes. Claro que alguns passos mais importantes já foram desenvolvidos pensando um pouco nessa ampliação que seria feita no futuro como por exemplo, o template que criei para a configuração do "modo automático"da ferramenta, ele é genérico e possivelmente sofrerá poucas alterações. No que diz respeito a prestar suporte para qualquer projeto que esteja utilizando git, precisarei fazer alguns testes para descobrir quais passos precisarei dar para alcançar esse objetivo. Este será um dos principais objetivos da próxima etapa do projeto.

5.2. Criação de uma interface gráfica para a versão de terminal

Logo em sequência de prioridade, quero construir uma interface gráfica para a versão de terminal (versão principal do IFFT). Eu tive essa ideia depois que eu me vi incomodado

com o fato de que a saída do programa era dada num formato pouco visual e em forma de texto simples no terminal, além disso eu me lembrei das páginas de relatório de cobertura de frameworks de testes e pensei em fazer algo similar. Isso não só vai tornar a saída da aplicação muito mais agradável visualmente, mas também mais simples de ser entendida, sem contar que abre mais margem para que possa mostrar outras informações relevantes. Achei essa ideia tão interessante que eu até mesmo comecei a trabalhar numa versão, muito rudimentar é claro, do que seria essa interface, abaixo segue duas imagens que traz como os resultados da ferramenta são exibidos atualmente (no terminal) e o protótipo que fiz de uma interface:

```
thiagosan@thiagosan-Aspire-A514-54:~/Área de Trabalho/IFFT/mock_project$ git commit -m "commit test"
Match: <re.Match object; span=(0, 44), match='#IFFT.Then("file1.py", "foo1_related_block")'>
Modified lines within the block: ['def foo4(number1: int, number2: int) -> int:', 'return number1 / number2', '', '']
Should also modify: file1.py
Block label: foo1_related_block
IFFT check detected changes in the blocks.
Changes detected in IFFT blocks. Do you want to continue with the commit? (y/n):
```

Figura 11. Imagem da saída do IFFT, caso em que a ferramenta faz o alerta

IFFT Blocks in Project

app.py

IFFT Block

```
#IFFT.If(foo1 block)
def foo1(number1: int, number2: int) -> int:
    # Adding a change for testing purposes
    return number1 + number2

def foo2(number1: int, number2: int) -> int:
    return number1 - number2

def foo3(number1: int, number2: int) -> int:
    return number1 * number2

def foo4(number1: int, number2: int) -> int:
    return number1 / number2

# Adding a change for testing purposes

def foo5(number1: int, number2: int) -> int:
    return number1 ** number2

def foo6(number1: int, number2: int) -> int:
    return number1 % number2

def foo7(number1: int, number2: int) -> int:
    return number1 // number2
```

Modified Lines within the Block:

```
def foo3(number1: int, number2: int) -> int:
return number1 * number2

def foo4(number1: int, number2: int) -> int:
return number1 / number2
```

Associated File: "file1.py"

Associated Label: "foo1_related_block"

Figura 12. Imagem do protótipo de interface que será desenvolvido na segunda iteração do projeto

5.3. Desenvolvimento de features novas mais sofisticadas

Um outro aspecto que eu gostaria de trazer para a ferramenta são features mais complexas como por exemplo permitir que o desenvolvedor possa fazer blocos IFFT aninhados.

Estas são mudanças mais desafiadoras porque envolve mexer com a parte crítica da ferramenta e a principal preocupação deve ser a de acabar inserindo erros onde não havia anteriormente, os testes serão essenciais para ajudar a combater esse tipo de problema.

5.4. Suporte a novas superfícies

Por fim, quero expandir o IFFT para que ele possa funcionar de outras formas, como por exemplo através de uma extensão que poderia ser usada no VSCode e também gostaria de disponibilizá-lo a partir de um plugin que pudesse ser inserido no pipeline de CI/CD do Guthub Actions de um projeto.

Referências

[1] Git Hooks by Atlasian. https://www.atlassian.com/br/git/tutorials/git-hooks

[2] Making a python l. https://python.plainenglish.io/asts-and-making-a-python-linter-from-scratch-79e66e8d99b8

[3] Git Hooks. https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks

[4] Pydriller. https://github.com/ishepard/pydriller

[5] GitPython. https://gitpython.readthedocs.io/en/stable/