DOI: 10.5748/19CONTECSI/COM/ESD/6944

LABCODE ANALYZER: A SEMI-AUTOMATIC REFACTORING TOOL FROM STATIC CODE ANALYSIS OF LABVIEW APPLICATIONS

LABCODE ANALYZER: UMA FERRAMENTA SEMIAUTOMÁTICA DE REFATORAÇÃO A PARTIR DA ANÁLISE ESTÁTICA DE CÓDIGO DE APLICAÇÕES EM LABVIEW

Ederson Ramalho ; https://orcid.org/0000-0002-2941-2936
Instituto de Pesquisas Tecnológicas de São Paulo

Paulo Sérgio Muniz Silva
Instituto de Pesquisas Tecnológicas de São Paulo

LABCODE ANALYZER: A SEMI-AUTOMATIC REFACTORING TOOL FROM STATIC CODE ANALYSIS OF LABVIEW APPLICATIONS

LABCODE ANALYZER: UMA FERRAMENTA SEMIAUTOMÁTICA DE REFATORAÇÃO A PARTIR DA ANÁLISE ESTÁTICA DE CÓDIGO DE APLICAÇÕES EM LABVIEW

Abstract: Refactoring is a very important technique used throughout the software cycle, being responsible for increasing readability, scalability, and the ability to maintain an application. At the same time, this task is quite arduous depending on how and who produced the software. Clear examples of this scenario are the applications developed by end users, who are often unaware of the good practices applied to software development. Among the several specific languages used by these professionals is LabVIEW. LabVIEW applications are commonly used by technicians and engineers in industry for measurement, testing, and monitoring. This problem makes refactoring these applications a challenging task. This work presents a tool for static code analysis and refactoring in LabVIEW code with the objective of identifying, in a semi-automatic way, signs of software problems characterized as code smells in the analyzed codes and candidates to refactoring.

Keywords: LabVIEW, Code Smells, Technical Debt, Static Code Analysis, Refactoring.

Resumo: A refatoração é uma técnica muito importante utilizada ao longo do ciclo de software, sendo ela uma das responsáveis por aumentar a legibilidade, a escalabilidade e a capacidade de se manter uma aplicação. Ao mesmo tempo, essa tarefa é bastante árdua dependendo de como e de quem produziu o software. Exemplos claros desse cenário são as aplicações desenvolvidas por usuários finais, os quais, muitas vezes desconhecem as boas práticas aplicadas ao desenvolvimento de software. Dentre as diversas linguagens específicas utilizadas por esses profissionais está o LabVIEW. As aplicações em LabVIEW normalmente são utilizados por técnicos e engenheiros na indústria para medição, testes e monitoramento. Esse problema torna a refatoração dessas aplicações uma tarefa desafiadora. Este trabalho apresenta uma ferramenta para análise estática e refatoração de códigos em LabVIEW, com o objetivo de identificar e corrigir, de maneira semiautomática, indícios de problemas caracterizados como "code smells" nos códigos analisados e candidatos à refatoração.

Palavras-chave: LabVIEW, Cheiros de Código (code smells), Débito Técnico, Análise Estática de Código, Refatoração.

1. Introdução

É possível enumerar diversas definições para refatoração de software. Para Fowler (2019) por exemplo, a refatoração pode ser definida como a modificação feita internamente na estrutura de software, de maneira a deixá-lo mais legível e mais fácil de alterar, sem que essa mudança seja observável externamente. Além das melhorias, do ponto de vista de legibilidade e manutenibilidade, com a refatoração é possível alcançar melhorias de desempenho (Kaur & Singh, 2017). É preciso salientar também que, ao longo do ciclo de vida de software todo e qualquer projeto pode sofrer alterações, adição de novas funcionalidades e integração de novos membros ao time de desenvolvimento. Dessa forma, garantir a qualidade do código desenvolvido é fundamental na análise, alteração, manutenção e reuso desse artefato. Porém, nem sempre é possível garantir níveis ótimos de qualidade. Pressões por entregas, questões orçamentárias, escopos mal definidos e mudanças na equipe, podem diminuir a qualidade da aplicação desenvolvida (Makkar et al., 2021). Mesmo o software sendo entregue funcionando para o ambiente de produção, pode trazer uma longa lista de problemas potenciais que podem comprometer seu ciclo de vida no médio e longo prazo (Sobrinho et al., 2021). Outro aspecto importante é a herança do código por novos membros na equipe de desenvolvimento. Estima-se que os times que mantêm uma aplicação despendem 60% do tempo apenas para entender o que foi codificado (Mkaouer et al., 2014).

A refatoração torna-se uma tarefa ainda mais desafiadora quando se trata de aplicações desenvolvidas por usuários finais. Para Goodell (1998 apud Abdullah Mohd Zin, 2011), a programação realizada por usuários finais diz respeito ao desenvolvimento de aplicações computacionais feitas por profissionais que não possuem, necessariamente, experiência em linguagens de programação, ou mesmo que essa não seja sua atividade principal. Entretanto, existem alguns benefícios nessa abordagem. Ao mesmo tempo em que é possível reduzir o tempo de desenvolvimento, problemas relacionados à obtenção de requisitos podem ser mitigados, uma vez que, os próprios usuários estão na linha de frente do desenvolvimento. Além disso, minimizam-se problemas de comunicação entre analistas e usuários envolvidos no projeto (Amoroso & Cheney, 1992). O papel do usuário tem sido objeto de estudo há muito tempo. Scaffidi et al (2005) estimaram que, somente nos Estados Unidos, 90 milhões de pessoas fariam alguma atividade relacionada ao desenvolvimento como usuário final até o ano de 2012, seja ela em codificação de programas, manipulação de planilhas eletrônicas ou mesmo consultas a bancos de dados. Porém, esse modelo carrega alguns problemas. Muitos usuários finais não possuem uma base de engenharia de software, como por exemplo depuração, suporte a teste e mesmo gestão de mudança, os quais são itens básicos encontrados em ambientes de desenvolvimento de software profissional (Kuttal et al., 2014). A demanda por profissionais com múltiplas atribuições é cada vez mais latente. Quando esses profissionais são usuários de alguma aplicação computacional que afeta diretamente suas rotinas, é muito comum que eles sejam inseridos nesse contexto, seja para criar ou modificar artefatos de software, desde simples alterações como modificar uma interface de usuário, como até mesmo funcionalidades da aplicação. Assim, é muito comum o relato da alta incidência de erros em aplicações em que usuários finais participam do processo de desenvolvimento (Costabile et al., 2008).

Nessa categoria de programação por usuários finais está o LabVIEW. Essa plataforma de desenvolvimento foi criada nos anos 1980 com o objetivo de facilitar Engenheiros e

Cientistas, que não tinham conhecimento de programação de computadores, a desenvolver suas próprias aplicações de conexão com instrumentos, medição de sinais, aquisição de dados e testes de sistemas industriais (Jenning & Cueva, 2020). O LabVIEW apresenta uma abordagem totalmente gráfica para a construção do software. O modelo gráfico de programação torna o desenvolvimento mais intuitivo e facilita a compreensão de quem analisa o seu Diagrama de Blocos (nome dado ao ambiente onde o código é desenvolvido no LabVIEW). Além dessa característica, o LabVIEW possui um ambiente de desenvolvimento totalmente integrado e preparado para aplicações de teste e automação industrial, através de funções prontas para comunicação com equipamentos de aquisição de dados que podem coletar informações de sensores instalados em campo (Kerry & Snyder, 2010). Da mesma forma que em outros ambientes de programação por usuários finais, os desafios de refatoração de aplicações em LabVIEW também existem. Pesquisas apontam que alguns desenvolvedores, inclusive, encontram bastante dificuldade em interpretar um código gráfico escrito em LabVIEW por outra pessoa, preferindo em muitos casos escrever um novo código para aquela aplicação à sua maneira, em vez de procurar entender a aplicação que estão analisando (Henley & Fleming, 2016).

1.1 Contextualização do Problema

Dentre as características já mencionadas para aplicações desenvolvidas por usuários finais, as aplicações em LabVIEW têm grande potencial de ter problemas com relação à sua legibilidade, manutenibilidade e escalabilidade ao longo do tempo. Pela facilidade em se criar protótipos com essa ferramenta, aplicações de medição e testes de equipamentos, por exemplo, podem ser construídas com muita facilidade e muito rapidamente. Entretanto, para aplicações maiores, que tendem a operar por muito tempo e com possibilidade de agregar novas funcionalidades, a necessidade da elaboração de um bom planejamento é essencial para o sucesso do projeto (*LabVIEW Development Guidelines*, 2003). Quando tais cuidados não são tomados, fica cada vez mais difícil manter tais aplicações. Essa tarefa é ainda mais desafiadora quando quem mantém o código não foi quem o desenvolveu, podendo ser novos membros do time de desenvolvimento ou mesmo consultores contratados com a finalidade de resolver problemas encontrados no desenvolvimento. Mesmo sendo uma plataforma que já está no mercado há mais de 30 anos e com muitos usuários nos campos de engenharia e ciências, o LabVIEW possui poucas ferramentas para análise e alteração de código (Henley & Fleming, 2016).

Por um lado, as aplicações desenvolvidas por usuários finais, em muitos casos, visam resolver um problema pontual. Nesses casos, há pouca preocupação com eventuais necessidades de alterações, melhorias ou tolerância às falhas que possam ocorrer ao longo do uso do software. Além disso, a refatoração de códigos dessa categoria fica prejudicada por conta da falta de ferramentas que auxiliem na detecção e correção desses problemas nos códigos analisados.

Por outro lado, as aplicações desenvolvidas em LabVIEW que não levam em consideração a importância da legibilidade do código, a escalabilidade e sua manutenção, precisam de uma ferramenta de refatoração que se baseie nos principais indícios de problemas de codificação, para que auxilie os desenvolvedores na gestão, análises e correções desses problemas em suas aplicações.

1.1.1 Questão de Pesquisa e Objetivo

Esse artigo procura dar uma resposta inicial à seguinte questão de pesquisa: Como desenvolver uma ferramenta para efetuar a refatoração de códigos em LabVIEW, a partir da análise de indícios de problemas potenciais de programação nos códigos analisados?

Para tanto, estabelece como objetivo principal de pesquisa a definição dos principais indícios de problemas de programação em LabVIEW que fundamentem a implementação de uma ferramenta de refatoração, de tal modo que possa auxiliar a análise e correção desses problemas.

As demais seções deste artigo são divididas da seguinte forma: a próxima seção apresenta um resumo dos principais aspectos da fundamentação conceitual na qual esse trabalho se baseia; a seção 3 apresenta os resultados da pesquisa exploratória de trabalhos relacionados ao problema em pauta, o tratamento da questão de pesquisa e a descrição da proposta de uma ferramenta que auxilie na análise e correção de problemas de codificação em LabVIEW: o LabCODE Analyzer. As duas últimas seções apresentam, respectivamente, as conclusões do estágio atual da pesquisa e sugestões de trabalhos futuros.

2. Análise Estática e Refatoração da Base de Código

Cada vez mais é possível encontrar, nos mais diferentes setores econômicos, profissionais que fazem uso de ferramentas computacionais para auxiliá-los em suas atividades. Aplicações para cálculos complexos, planilhas eletrônicas, modelagem e interface com os mais diversos tipos de sensores são alguns dos exemplos em que tais profissionais desenvolvem software. Eles são popularmente conhecidos como desenvolvedores ou programadores usuários finais (Kuttal et al., 2014). As aplicações desenvolvidas em LabVIEW também se enquadram nesse cenário. Essa ferramenta normalmente é utilizada para o desenvolvimento de sistemas de testes, medição e monitoramento em vários segmentos industriais (Henley & Fleming, 2016).

Como em outras linguagens, os problemas de programação também estão presentes no LabVIEW. A falta de cuidado durante a construção das estruturas do software podem resultar em modelos que, ao longo do tempo, se tornam difíceis de manter, escalar ou, no pior dos casos, simplesmente ler (*LabVIEW Development Guidelines*, 2003). A característica de legibilidade é especialmente importante para desenvolvedores usuários finais, pois facilita a análise do código de programação e o entendimento do que ele realmente está fazendo. Essas estruturas decorrentes daquela falta de cuidado são conhecidas na Engenharia de Software como "Maus Cheiros de Código" (uma tradução livre do inglês *Code Smells*)[1], as quais violam algum princípio fundamental, ou boas práticas, de desenvolvimento e tem o potencial de impactar e comprometer negativamente a qualidade do software desenvolvido (Zhao & Gray, 2019).

Para tentar melhorar a qualidade de aplicações desenvolvidas, tanto no ambiente de desenvolvimento do LabVIEW quanto naqueles oferecidos para outras linguagens, existem ferramentas de análise estática de código. Essas ferramentas identificam estruturas problemáticas na base de código identificando-as para o desenvolvedor (Szoke et al., 2015). O LabVIEW também possui uma ferramenta com essa funcionalidade: o *VI Analyzer Tool*. Essa ferramenta será abordada com mais detalhes adiante.

Por conveniência, neste artigo grafa-se o termo em inglês, como é comumente referenciado na comunidade de software.

O processo de detecção dos chamados *code smells* é importante para que o desenvolvedor de uma nova aplicação, ou o responsável pela manutenção de uma aplicação já desenvolvida, tenha condições de tomar a melhor decisão sobre o que fazer para solucionar o problema detectado. Caso opte-se por alterar a estrutura com problema, o programa será refatorado. Para Fowler (2019), a refatoração é a modificação de uma estrutura interna de um programa de computador com o intuito de torná-lo mais legível e, consequentemente, com menor custo para ser alterado, sem que seu comportamento observável, seja alterado.

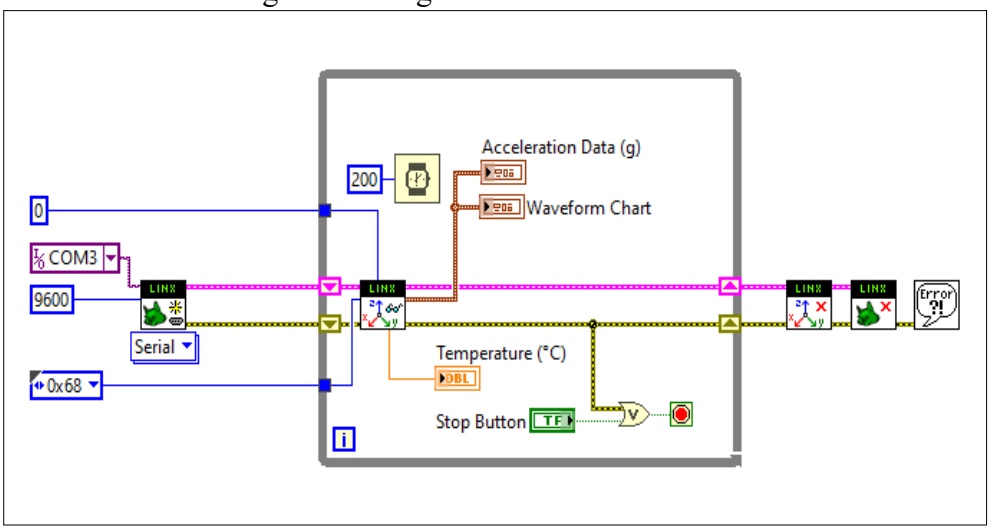
2.1 Desenvolvimento de Software em LabVIEW

O LabVIEW é um ambiente de programação gráfico. O desenvolvedor constrói todo o código de maneira gráfica. O ambiente de programação do código em LabVIEW é conhecido como Diagrama de Blocos. Os principais itens que existem em um diagrama de blocos são:

- Fios: por onde trafegam os dados. Existem vários tipos de dados e a cor do fio representa o tipo de dado que trafega pelo fio.
- Terminais: são as entradas e saídas de dados. No LabVIEW, as entradas são comumente conhecidas como Controles e as saídas como Indicadores. Além dos controles, as entradas também podem ser fornecidas por constantes.
- Nós: são blocos onde os fios entram e saem do programa. São responsáveis por executar alguma tarefa. Podem ser comparados nas linguagens de programação textual às chamadas de funções.

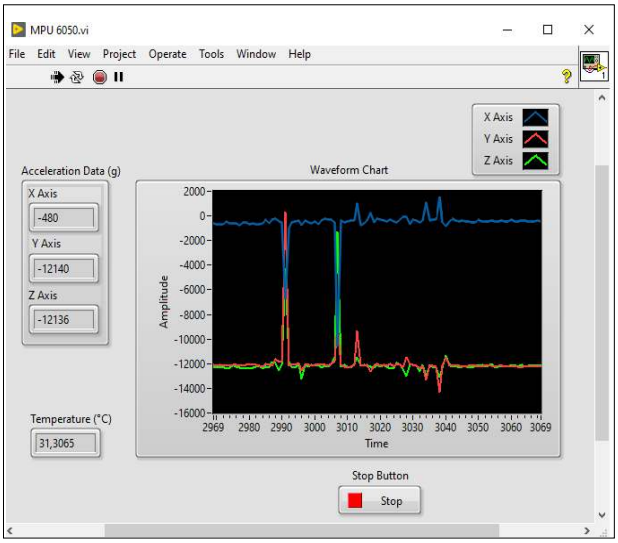
Além do Diagrama de Blocos, o programador desenvolve paralelamente ao código a interface de usuário que será utilizada pela aplicação. O ambiente para esse desenvolvimento é chamado de Painel Frontal. Nesse ambiente, os controles e indicadores mencionados são mostrados ao usuário para que ele possa inserir dados (Controles) e ler as saídas (Indicadores) (National Instruments, 2021). As Figuras 1 e 2 apresentam esses elementos.

Figura 1 - Diagrama de Blocos em LabVIEW.



Fonte - Elaborado pelo Autor.

Figura 2 - Painel Frontal do LabVIEW.



Fonte - Elaborado pelo Autor.

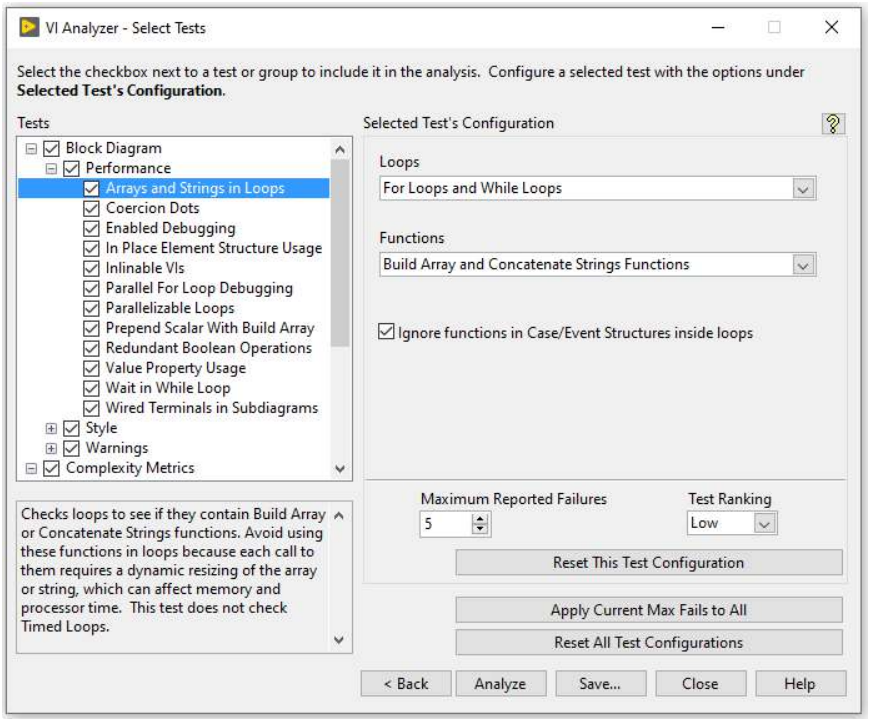
2.2 Análise Estática de Código

As ferramentas de análise estática de código possuem a capacidade de analisar o programa desenvolvido e indicar possíveis indícios de problemas de programação em uma determinada linguagem. Tais problemas, na maioria dos casos, não impedem o funcionamento da aplicação, e no curto prazo podem até ser benéficos para aumentar a produtividade e atender prazos curtos de entrega. Entretanto, no médio e longo prazos esses problemas podem resultar uma grande quantidade de trabalho extra, incorrendo no que é denominado de débito técnico (Sobrinho et al., 2021). Dessa forma, essas ferramentas podem ser muito úteis, tanto ao longo do desenvolvimento, quanto em sua manutenção.

Quando se trata de qualidade no desenvolvimento de software, existem alguns conceitos que que já estão consagrados, tanto na indústria como na academia. Um caso típico são os princípios SOLID (Martin, 2018). Esses princípios propõem soluções, fundamentadas nos conceitos da orientação por objetos, para que os programas sejam mais inteligíveis, flexíveis e, sobretudo, manuteníveis. Alguns autores relacionam alguns problemas de código em orientação por objetos com estruturas no código de LabVIEW. Zhao et al. (2021), mostra, por exemplo, que classes muito extensas em orientação por objetos se assemelham a funções (Sub VIs) muito grandes construídas no LabVIEW. Da mesma, forma, ele indica que o *code smell* em orientação por objetos conhecido como "Lista longa de parâmetros" (*Long Parameter List*) é similar a um arquivo de LabVIEW (VI) com uma grande quantidade de controles.

Para o LabVIEW, em particular, existe uma ferramenta com a função de detectar problemas no código estaticamente, ou seja, analisando sua estrutura e não o seu comportamento. Essa ferramenta é o VI Analyzer Tool, ilustrado na Figura 3. Essa ferramenta possui mais de 60 testes divididos em subgrupos. Esses testes inspecionam o código e reportam problemas que impactam o estilo e o desempenho do programa em questão (Blume, 2007).

Figura 3 – Tela de Configuração do VI Analyzer Tool.



Fonte: Elaborado pelo Autor.

Essa ferramenta é muito útil para a detecção de problemas de código, porém, ela não sugere quais alterações poderiam ser feitas para resolver o código problemático. Tampouco existe, no ambiente do LabVIEW, uma ferramenta acoplada a ela que permita a refatoração do código de modo automático ou semiautomático, caso o desenvolvedor opte por efetuar alterações.

2.3 Refatoração Automática e Semiautomática de Código

A refatoração é um processo de manutenção para reestruturar um código com o intuito de melhorar alguns atributos de software, entre os quais, pode-se destacar a legibilidade e a usabilidade (Baqais & Alshayeb, 2020). A refatoração facilita as tarefas de manutenção e melhora o projeto da estrutura do código, enquanto preserva a funcionalidade do software. A literatura propõe várias abordagens de refatoração automática, em sua maioria utilizando métricas de software como, por exemplo, parâmetro para estimar as melhorias de qualidade nos programas em que a refatoração é aplicada (Mkaouer et al., 2014). Enquanto a refatoração automática modifica o software automaticamente, existe também a abordagem semiautomática, em que o desenvolvedor pode revisar o código antes de tomar a decisão quanto ao tipo de refatoração, ao parâmetro necessário para a modificação que será feita ou mesmo à melhor solução de refatoração, pois em alguns casos existe mais de uma opção para resolver o problema a ser refatorado (Baqais & Alshayeb, 2020). Entretanto, existe uma lacuna entre as ferramentas de refatoração automática e seu uso. Negara (2013 apud Verebi, 2015), conduziu um estudo empírico que mostrou que tais ferramentas de refatoração automática são subutilizadas e aplicadas apenas em pequenas modificações de código.

No caso do LabVIEW, a situação não é muito diferente. A refatoração manual é uma tarefa tediosa. Henley e Fleming (2016) relatam em seu trabalho que muitos desenvolvedores preferem desenvolver um novo programa a ter que modificar algo que já existia. Outro relato, agora de um gestor de equipe nesse mesmo artigo, define a alteração de um código em

LabVIEW como um problema de “1 milhão de dólares”, e que seu time dispende horas tentando entender e modificar um código até que decidem descartá-lo e começar algo novo. Desse modo, ter uma ferramenta que faça análise estática da base de código, identifique *code smells* e permita ou guie a efetivação de alterações de forma automática ou semiautomática facilitaria muito o dia a dia de desenvolvedores e mantenedores de código.

3. Resultados atuais da pesquisa

A primeira parte desta seção apresenta os principais trabalhos relacionados a esse tema que foram coletados a partir de uma pesquisa exploratória da literatura, como etapa preliminar de uma revisão sistemática da literatura a ser realizada com critérios de busca mais acurados. Um critério importante adotado nessa pesquisa foi o de desconsiderar as extensões de análise estática e de refatoração já incorporadas nativamente em ambientes de desenvolvimento bem conhecidos da comunidade de desenvolvedores, como Visual Studio Code, Eclipse, NetBeans, IntelliJ IDEA, etc. A parte final, esboça as principais características de uma ferramenta para análise estática e refatoração semiautomática de códigos em LabVIEW.

3.1 Pesquisa Exploratória da Literatura

Herbold et al. (2009) abordam a importância da inspeção e verificação de programas durante o uso de ferramentas de análise estática de código. Essas ferramentas possibilitam a localização de problemas de codificação, especialmente estruturas internas que violam as boas práticas de programação: os denominados *code smells*. Os autores indicam a presença de uma lacuna entre os analisadores estáticos de código e as ferramentas de refatoração automática. Por conta disso, propõem uma prova de conceito de uma ferramenta que analisa problemas de códigos em Java reportados por duas ferramentas, as quais são *plugins* dos ambientes de desenvolvimento Eclipse e NetBeans. As ferramentas são conhecidas como *“FindBugs”* e *“PMD”*. A ferramenta desenvolvida pelos autores, chamada *“AddFix”*, incluía não só aspectos automáticos, ou seja, analisava os problemas e efetuava correções automaticamente, como também aspectos semiautomáticos, em que era necessário selecionar a opção de refatoração mais adequada para o problema reportado. Para ambos os casos, a ferramenta se mostrou eficiente para a remoção dos problemas de código identificados.

Chambers e Scaffidi (2013) destacam a dificuldade em se otimizar o desempenho de códigos desenvolvidos em LabVIEW. Os autores ressaltam que a melhor forma de se evitar problemas de desempenho é evitar estruturas que prejudiquem o desempenho da aplicação. Através de entrevistas com especialistas, eles conseguiram elencar 13 problemas comuns que prejudicam o desempenho de uma aplicação em LabVIEW, reproduzidos na Tabela 1. A partir desse levantamento, os autores desenvolveram uma aplicação que localizava essas estruturas com problemas para que os desenvolvedores tivessem ciência daquilo que poderia impactar o desempenho do código.

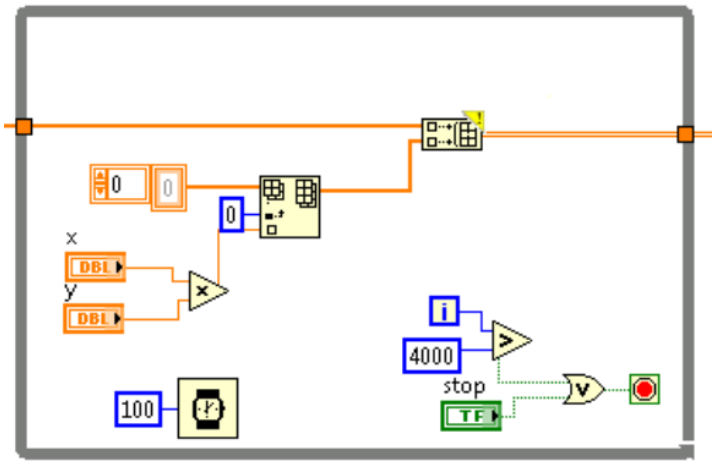
Tabela 1 – Problemas comuns reportados por especialistas que prestam suporte a usuários do LabVIEW

Problem (ᴾ = performance-related)	Description	# who reported
Too Many Variables ᴾ	Too many variables in a VI can lead to race conditions, The suggested limit ranges from 2 to 5 variables per VI—more than this raises the risk of inconsistent use.	8
Build Array inside Loop ᴾ	When a build-array node is inside of a loop, it builds a new copy of the array every iteration. This can cause slow performance and memory issues.	6
Multiple Array Copies ᴾ	When an array is forked and passed to multiple nodes, a new copy of the array is made—and large arrays forking multiple times can cause memory issues	5
No Wait in a loop ᴾ	If there is no wait inside of a loop, it could cause synchronization issues and also usually makes front panel elements unresponsive	4
Unconnected Front Panel elements	When elements on the front panel are not connected to anything in the block diagram, this can lead to unexpected behavior or confusing VIs	3
Redundant operations ᴾ	When there is large data, such as a large array, having operations that are redundant can waste time and memory in the VI (e.g., adding 0 to each element of an array)	3
No Constraint on Queues ᴾ	When there are no constraints on a queue, the queue can grow infinitely large. This can cause performance problems due to memory and loop synchronization issues.	3
Infinite While Loop ᴾ	A loop that runs infinitely (sometimes due to the lack of any rule for termination) may cause issues with expected behavior and execution time.	3
Non Reentrant VI's ᴾ	Non-reentrant VIs have a data space shared between multiple calls. If a non-reentrant VI is in a structure that can be parallelized, blocking causes poor execution speed.	2
String Concatenation in Loop ᴾ	When string concatenation occurs inside of a loop, it builds a new copy every iteration. This can cause slow performance and memory issues.	2
Sequence instead of State machine ᴾ	Sequence structures remove a lot of the power of LabVIEW, limiting the compiler's optimization options and potentially reducing readability.	2
Multiple Nested Loops	Multiple nested loops reduce readability.	2
Only Loop Once	If a loop only iterates once, then having the loop structure reduces readability	2

Fonte: Chambers e Scaffidi (2013).

O protótipo da ferramenta foi avaliado por 13 usuários do LabVIEW que analisaram um determinado código, com e sem a ferramenta proposta, buscando uma solução para os problemas nele contidos. Os resultados foram julgados satisfatórios, pois 92% dos usuários conseguiram solucionar o problema proposto utilizando a ferramenta, enquanto apenas 60% conseguiram solucioná-lo sem a ferramenta. A Figura 13 ilustra a identificação de um desses problemas realizada pelo protótipo.

Figura 4 – Protótipo insere uma marcação amarela sobre a função *Build Array*.

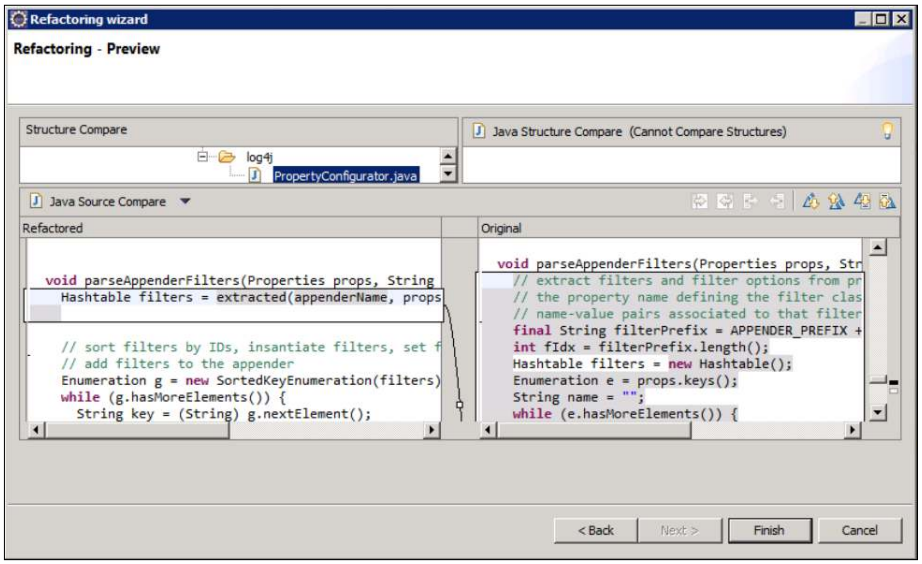


Fonte: Chambers e Scaffidi (2013).

Para Szoke et al. (2015), a refatoração contínua do código é uma maneira eficiente de manter a qualidade do código, além de impedir seu processo de degradação caso tais cuidados não sejam tomados. Entretanto, além de ressaltarem que o processo de refatoração não é algo simples, devido à necessidade de análise da melhor alternativa de refatoração por parte do desenvolvedor, essa mudança pode incorrer em alguns riscos. Assim, o código modificado precisa ser submetido a testes que garantam que nenhuma funcionalidade da

aplicação não foi comprometida. O artigo propôs uma ferramenta de refatoração automática chamada *"FaultBuster"* - ilustrada na Figura 5 - para códigos em Java que utilizavam o conceito de refatoração contínua. O processo de refatoração era integrado à IDE de desenvolvimento e analisava o código periodicamente à medida que ele era construído. A ferramenta tinha capacidade de encontrar e corrigir cerca de 40 problemas de código diferentes.

Figura 5 – Ferramenta *FaultBuster* mostrando o antes e depois do trecho do código refatorado.

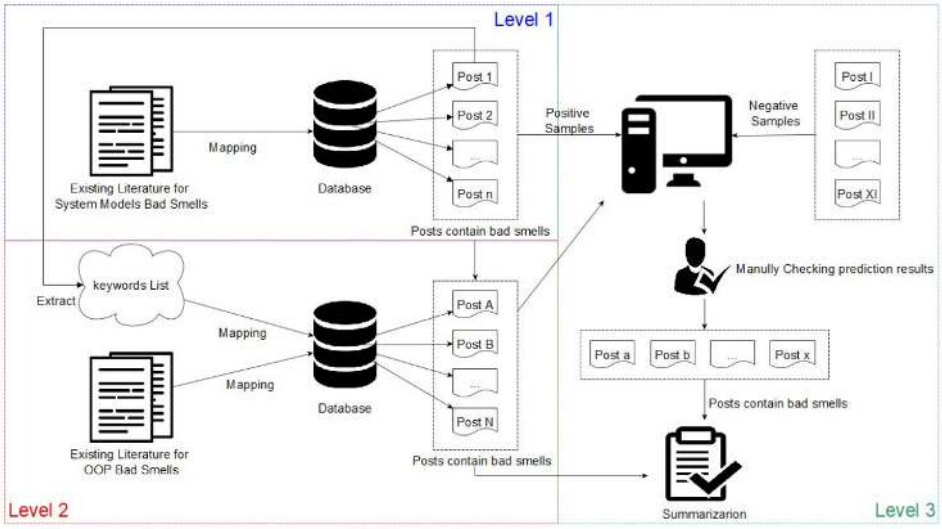


Fonte: Szoke et al. (2015).

A ferramenta *FaultBuster* foi um projeto apoiada pelos governos húngaro e da União Europeia, tendo sido implementado por seis empresas. Foram analisados pela ferramenta diversos projetos, totalizando um número superior a 5 milhões de linhas de código, nos quais foram detectados e corrigidos mais de 11.000 problemas de programação.

Zhao e Gray (2019) propõem uma ferramenta para sumarização de problemas de código em LabVIEW, chamado BESMER (*Bad Smell Summarization from End Users*), ilustrado na Figura 6. Essa ferramenta faz uma busca em fóruns de discussão na Internet por palavras-chaves vinculadas a problemas de codificação existentes na literatura (Nível 1). Após essa busca, a ferramenta relaciona as palavras mais mencionadas nos fóruns com termos comuns de problemas de software conhecidos em programação orientada por objetos (Nível 2). A ferramenta cria uma marcação como uma amostra positiva nas postagens que se relacionam a problemas de software e uma marcação de amostra negativa para o contrário. Ao final, os pesquisadores verificam se as postagens realmente se relacionam com a lista de problemas de codificação existentes na literatura (Nível 3).

Figura 6 – Os três níveis do mecanismo de busca da ferramenta BESMER.



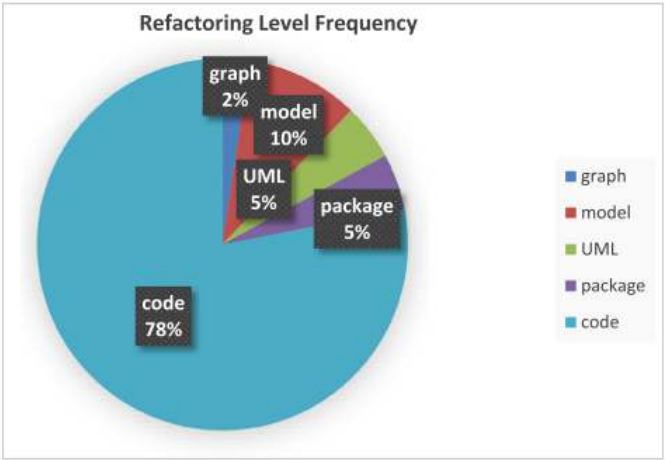
Fonte: Zhao e Gray (2019).

Devido à pouca literatura existente sobre problemas de codificação em LabVIEW, foram utilizadas na ferramenta BESMER os *code smells* catalogados por Chambers e Scaffidi (2013), mostrados anteriormente na Tabela 1. Nesse trabalho de Zhao e Gray (2019) foi possível verificar a presença de 6 dos 13 problemas existentes na literatura postados em fóruns sobre LabVIEW.

Baqais e Alshayeb (2020) publicaram uma revisão sistemática da literatura sobre refatoração automática de código. As principais evidências elencadas na revisão foram as seguintes:

- O nível de refatoração acontece majoritariamente no código. Os demais campos em que ela acontece se dividem em pacotes, diagramas, modelos e grafos, como mostrados na Figura 7.

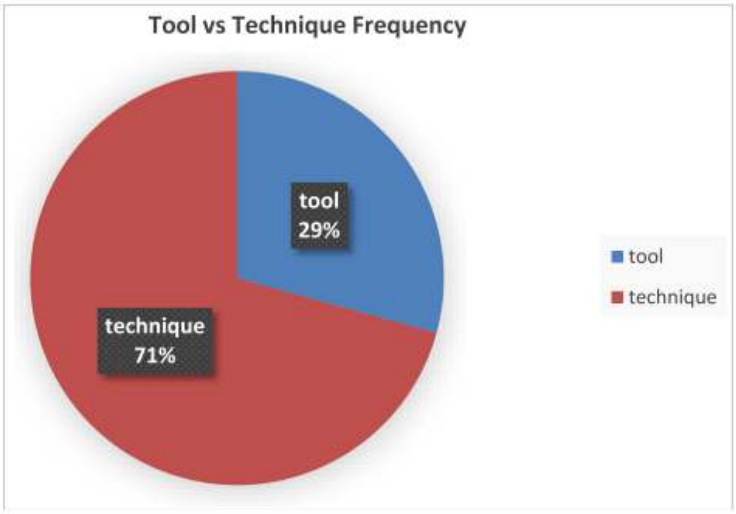
Figura 7 – Nível de Frequência da refatoração.



Fonte: Baqais e Alshayeb (2020).

- Cerca de dois terços dos trabalhos avaliados nessa revisão focalizam técnicas de refatoração automática. O outro um terço apresenta alguma ferramenta para execução desse tipo de refatoração, como mostrado na Figura 8.

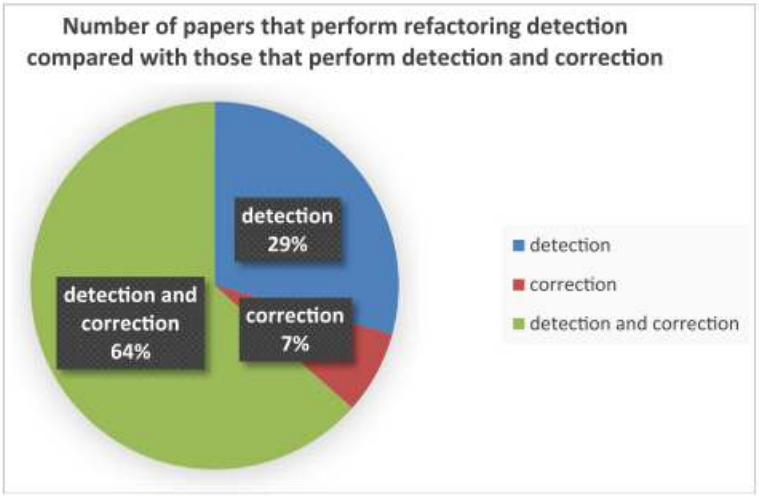
Figura 8 – Quantidade artigos que abordam técnica ou ferramenta de refatoração.



Fonte: Baqais e Alshayeb (2020).

- A grande maioria dos artigos (64%) focalizam a detecção e correção nos processos de refatoração de código. Outros 29% tratam a detecção e os demais 7% somente no processo de correção, como pode ser visto na Figura 9.

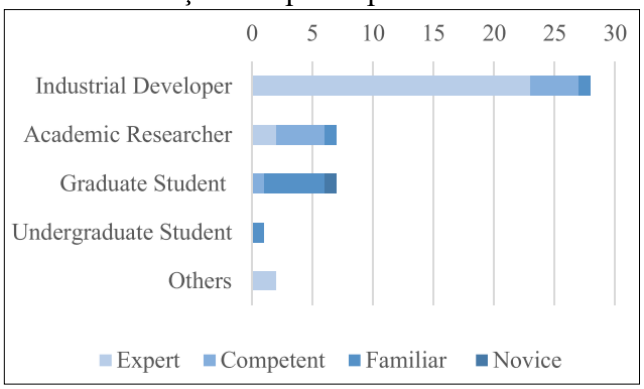
Figura 9 – Foco dos artigos quanto ao processo de refatoração.



Fonte: Baqais e Alshayeb (2020).

Zhao et al. (2021) apresentam um estudo realizado a partir de um *survey* (45 participantes) com o intuito de entender a percepção dos desenvolvedores relacionada aos *code smells*. Além disso, procuraram verificar essa percepção em diferentes cenários e com diferentes níveis de experiência dos participantes do estudo, como mostra a Figura 10. A partir das perguntas de pesquisa no artigo e o resultado do *survey*, os autores elaboraram algumas recomendações a fim de minimizar a ocorrência de code *smells* nos códigos em LabVIEW.

Figura 10 – Área de atuação dos participantes e nível de conhecimento.



Fonte: Zhao et al. (2021).

Os problemas de código foram denominados *model smells*, uma vez que os autores relacionam códigos gráficos a modelos de programação. O Quadro 1 reproduz os problemas de código abordados no *survey*.

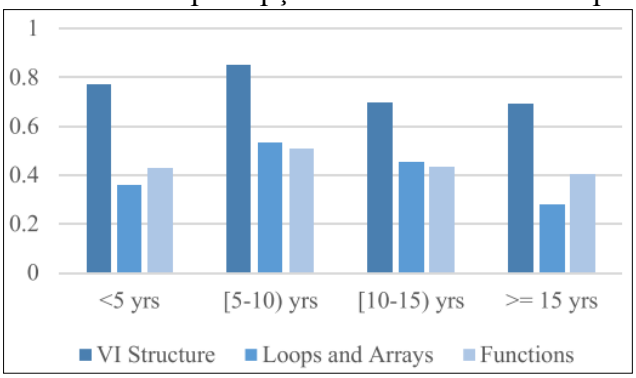
Quadro 1 – Problemas de código abordados no *survey*.

Category	Model Smells
VI Structure	1.1. Too many controls; 1.2. Large VI; 1.3. Deficient SubVI; 1.4. Deeply Nested Hierarchy; 1.5. Dead code; 1.6. Duplicated code; 1.7. Unorganized wires; 1.8. Unconnected front panel element.
Loops and Arrays	2.1. Build array inside a loop; 2.2. No wait in a while loop; 2.3. Infinite loop; 2.4. Loop once; 2.5. Concatenate strings in a loop; 2.6. Multiple nested loops; 2.7. Fix-length array resizing.
Functions	3.1. Stacked Sequence structure; 3.2. Non-reentrant VI; 3.3. Excessive property nodes; 3.4. Redundant operations; 3.5. Excessive data copies.

Fonte: Zhao et al. (2021).

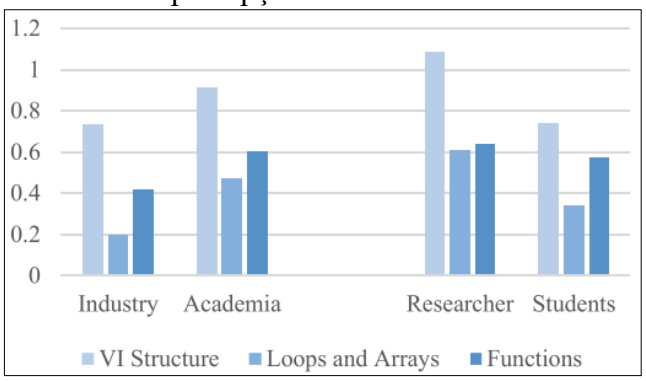
A pesquisa mostrou que a percepção quanto aos problemas de codificação se assemelha entre profissionais com menos de 5 e mais de 15 anos de experiência. Profissionais entre 5 e 15 anos têm uma percepção semelhante (Figura 11). Em relação ao segmento profissional, todas as áreas têm percepção semelhante (Figura 12).

Figura 11 – Valor de percepção média baseada na experiência.



Fonte: Zhao et al. (2021).

Figura 12 – Valor de percepção média baseada na área de atuação.



Fonte: Zhao et al. (2021).

Popoola et al. (2021) destacam que sistemas de versionamento de código, como Git por exemplo, são fontes de dados relevantes para avaliar a qualidade do software desenvolvido ao longo das várias versões que são criadas. Os autores descrevem uma aplicação que analisou 10 repositórios com códigos em LabVIEW NXG (versão do LabVIEW que foi descontinuada em 2021). Os códigos em LabVIEW NXG podem ser representados em arquivos XML (Figura 13). Dessa forma, a ferramenta analisa os arquivos XML procurando por estruturas de código problemáticas e os reporta. A ferramenta executou essa análise em todas as versões efetivadas (*commits*) pelos desenvolvedores e as dividiu em primeira versão, segunda versão e demais versões, analisando a incidência de *code smells* ao longo da evolução do desenvolvimento. Devido à dificuldade em se implementar uma grande quantidade de *code smells* para serem analisados pela ferramenta, foram selecionados sete *code smells* entre os trabalhos de Zhao e Gray (2019) e Chambers e Scaffidi (2013), usados na ferramenta que analisou os códigos nos repositórios. Toda a vez que um destes *code smells* fosse localizado nas versões analisadas, eles seriam reportados pela ferramenta.

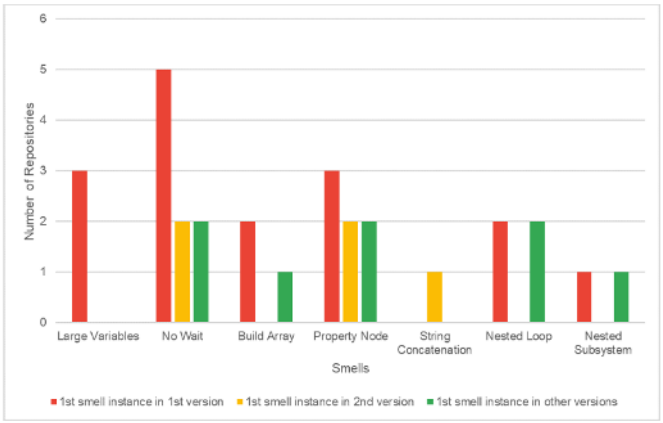
Figura 13 – Trecho de arquivo XML representando o código em LabVIEW.

```
<BlockDiagram Id="12">
    <DataAccessor Id="16" Label="19">
    <Terminal DataType="Double" Direction=
    "Output" />
  </DataAccessor>
  <NodeLabel AttachedTo="16" Id="19">
    <p.Text>Input 1</p.Text>
  </NodeLabel>
  <DataAccessor Id="21" Label="24" >
    <Terminal DataType="Double" Direction=
    "Output"/>
  </DataAccessor>
  <NodeLabel AttachedTo="21" Id="24">
    <p.Text>Input 2</p.Text>
  </NodeLabel>
  <DataAccessor Id="26" Label="29" >
    <Terminal DataType="Double" Direction=
    "Input"/>
  </DataAccessor>
  <NodeLabel AttachedTo="26" Id="29">
    <p.Text>Output</p.Text>
  </NodeLabel>
  <Add Id="30" Terminals="o=33, c0t0v=31, c1t0v
    =32" />
  <Wire Id="31" Joints="N(16:Value)| N(30:
  c0t0v)" />
  <Wire Id="32" Joints="N(21:Value)| N(30:
  c1t0v)" />
  <Wire Id="33" Joints="N(30:o)|N(26:Value)" />
</BlockDiagram>
```

Fonte: Popoola et al. (2021).

O estudo preliminar sugere que os problemas de codificação são inseridos nas primeiras versões do código e frequentemente persistem ao longo de novas versões, com se pode observar na Figura 14. Isso poderia ser uma indicação da necessidade de uma ferramenta de refatoração para análise e correção do código.

Figura 14 – *Code Smells* persistentes em várias versões.



Fonte: Popoola et al. (2021).

3.2 Projeto da Ferramenta *LabCODE Analyzer*

O principal objetivo da presente pesquisa é o projeto e implementação da ferramenta *LabCODE Analyzer* que visa analisar códigos em LabVIEW para indicar problemas de codificação (*code smells*) e sugerir, de modo semiautomático, refatoração do código que possa afetar o desempenho, o funcionamento, a manutenibilidade, a legibilidade, a escalabilidade e o reuso de uma aplicação nos médio e longo prazos. Devido à dificuldade de se implementar todos os *code smells* encontrados na literatura pesquisada, numa primeira etapa será adotada a mesma estratégia utilizada por Popoola et al. (2021), em que foram

escolhidos sete problemas mais comuns encontrados em aplicações LabVIEW. Além de analisar os códigos em LabVIEW, o *LabCODE* tem a capacidade de explicar a razão pela qual o problema indicado pode prejudicar a aplicação, e apresentar uma sugestão para a refatoração do trecho de código que causa o problema, características ausentes na ferramenta *VI Analyzer Tool* do LabVIEW. Essas características permitem a alteração do código para a correção do problema, bem como a colocação de uma marcação indicando que aquele *code smell* poderá ser ignorado em futuras análises, caso o desenvolvedor entenda ser necessário a utilização do código considerado como *code smell*. Em outras palavras, o desenvolvedor tem a alternativa da aceitação da dívida técnica decorrente do *code smell* detectado.

Os problemas de codificação analisados no LabCODE são os seguintes:

- Uso de Variáveis Locais: as variáveis locais são constructos poderosos no LabVIEW. Como dito no início deste trabalho, as entradas e saídas no LabVIEW são conhecidas, respectivamente, como controles e indicadores. Sempre que é necessário ler ou escrever em um controle ou indicador, em um trecho do programa que não esteja próximo desses terminais, o uso de uma variável local pode ser interessante. Entretanto, utilizar esse recurso de forma descontrolada pode incorrer em situações indesejadas, como, por exemplo as condições de corrida em cenários de concorrência (*race condition*), são situações em que ocorre uma escrita indesejada em um desses terminais e estes reproduzem valores errados na aplicação. Além disso, o uso excessivo desse recurso aumenta o uso de memória, podendo comprometer o desempenho da aplicação.

- Estruturas de Repetição (*For Loop* e *While Loop*) sem funções de temporização: todas as vezes em que se utiliza uma estrutura de repetição no LabVIEW, a plataforma tenta executar as iterações imediatamente uma após a outra. Esse fator faz com que a CPU dedique todo o seu processamento na execução dessas iterações, não deixando que ela execute outras funções. Para contornar essa característica, utiliza-se uma função de temporização dentro da estrutura de *loop*, permitindo que, a cada iteração, a CPU tenha um tempo para que possa executar outras funcionalidades da aplicação.

- Função *Build Array* em estruturas de repetição: a função *Build Array* é utilizada para concatenar múltiplos vetores ou adicionar um novo elemento em um vetor n-dimensional. O problema associado a essa função é que, quando ela está dentro de um *loop*, ela cria uma cópia do vetor a cada nova iteração. À medida que essa aplicação está em uso, esse fator pode sobrecarregar o uso de memória do computador e assim diminuir o desempenho da aplicação.

- Função *String Concatenation* em estruturas de repetição: o problema com essa função é semelhante ao da função *Build Array*. A cada iteração do *loop* a aplicação cria uma cópia da *string* que é concatenada, podendo aumentar o consumo de memória e reduzir o desempenho da aplicação.

- Uso excessivo de *Property Nodes*: os *Property Nodes* são recursos muito úteis, utilizados para alterar propriedades de controles, indicadores e da própria interface de usuário, por exemplo. Entretanto, quando utilizado em excesso pode reduzir o desempenho da aplicação.

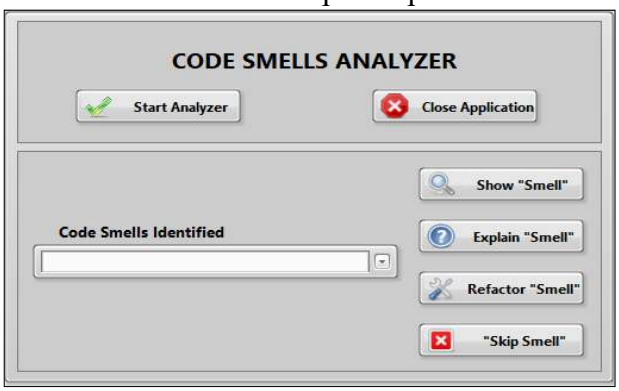
- Múltiplos *loops* aninhados: esse é um problema comum em qualquer linguagem de programação. À medida que se tem mais estruturas de repetição dentro de outras estruturas de repetição, aumenta-se a complexidade ciclomática da aplicação, o que pode dificultar o entendimento do software por quem mantém a referida aplicação.

- Hierarquia de subsistemas aninhados em profundidade: esse problema diz respeito ao uso de várias chamadas de funções dentro de outras funções. É aproximadamente similar ao problema de *loops* aninhados, dificultando o entendimento da aplicação.

Para o caso do *LabCODE*, será considerado um *code smell* sempre que houver mais de dois *loops* aninhados e mais de duas SubVIs (funções em LabVIEW) aninhadas na aplicação.

A Figura 15 apresenta a interface de usuário da versão atual do LabCODE, ainda em seu estágio de protótipo.

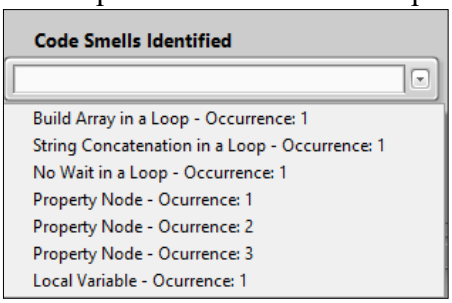
Figura 15 – Interface de usuário do protótipo do *LabCODE Analyzer*.



Fonte: Elaborado pelo Autor.

Para iniciar o *LabCODE*, o usuário clica no botão *Start Analyzer* para selecionar o arquivo em LabVIEW que será analisado. Após isso, a aplicação gera uma lista de problemas identificados pelo analisador de código estático e os coloca no indicador *Code Smells Identified*, como ilustrado na Figura 16.

Figura 16 – Lista de problemas identificados pelo *LabCODE*.



Fonte: Elaborado pelo Autor.

As principais ações do *LabCODE* são as seguintes:

- *Show Smell*: clicando nesse botão, o *LabCODE* abre o programa que está sendo analisado e mostra o problema para o usuário diretamente no código.

- *Explain Smell*: essa função abre uma janela explicativa que mostra o significado do problema, identificado como *code smell*, e quais as implicações em mantê-lo.

- *Refactor Smell*: clicando nesse botão, o *LabCODE* mostra as opções de refatoração disponíveis para aquele problema, deixando que o usuário faça a escolha antes de a ferramenta sugerir a refatoração do código.

- *Skip Smell*: nessa opção, o *LabCODE* cria uma marcação no código analisado para que o problema indicado nessa análise não seja mais listado em análises futuras.

4. Conclusões

O primeiro ponto importante é responder à pergunta de pesquisa: *como desenvolver uma ferramenta para efetuar a refatoração de códigos desenvolvidos em LabVIEW de forma sistematizada, a partir da análise de indícios de problemas potenciais de programação nos códigos analisados?* Verificou-se, a partir da pesquisa exploratória da literatura correlata que o ponto de partida para qualquer ferramenta de análise e refatoração de código estático é uma definição bem clara dos principais problemas na linguagem de programação em questão que possam impactar sua qualidade, manutenibilidade, legibilidade e escalabilidade. As pesquisas de Chambers e Scaffidi (2013) e mais tarde de Zhao e Gray (2019) mostraram uma preocupação em identificar quais são esses problemas no contexto do LabVIEW. Outro ponto importante é a definição do grau de automatismo da ferramenta. Uma ferramenta totalmente automática, que analise o código e já faça as modificações necessárias pode, em alguns casos, dificultar o entendimento dos desenvolvedores e até mesmo inserir novos problemas. Além disso, algumas refatorações necessitam que haja alguma parametrização, de maneira que muitas vezes a abordagem manual ou semiautomática pode ser a escolha mais apropriada.

Um ponto importante confirmado ao longo da pesquisa é que nem sempre a presença de *code smells* no programa deve ser considerada um problema, isto é, nem sempre determinadas aceitações de dívida técnica são más decisões. Na verdade, é comum os desenvolvedores aceitarem-nas para obter a melhor solução diante de restrições que se apresentam no ciclo do desenvolvimento. Isso afeta a decisão sobre o grau de automatismo de uma ferramenta de análise de código. Uma ferramenta semiautomática permite ao desenvolvedor ser o responsável final sobre a decisão de se refatorar ou não o código.

Outro ponto importante a ser destacado é a aparente inexistência de ferramentas para a plataforma LabVIEW que, além de analisar os códigos estaticamente, também efetuem a refatoração ou, ao menos, orientem a desenvolvedor das alternativas de alteração de código. Tanto a ferramenta disponível no próprio LabVIEW, o *VI Analyzer Tool*, quanto os protótipos apresentados nos trabalhos de Chambers e Scaffidi (2013) e Popoola et al. (2021), apenas localizam os problemas de codificação. Dessa forma, a principal contribuição da ferramenta proposta – o *LabCODE Analyzer* - é oferecer ao desenvolvedor a possibilidade de executar a correção do código de maneira semiautomática através das alternativas possíveis de refatoração para o problema reportado.

Por fim, é importante destacar que o estágio atual da pesquisa refletido nesse artigo possui algumas limitações. Uma delas refere-se à natureza do método da pesquisa bibliográfica que foi aplicado. Por ainda se tratar de uma pesquisa exploratória da literatura pertinente, não é possível avaliar se os principais trabalhos relevantes sobre os conceitos relacionados à pesquisa foram cobertos. De todo modo, essa limitação será tratada na continuidade da pesquisa em uma revisão sistemática da literatura, cujos critérios de busca serão calibrados pelos resultados dessa pesquisa exploratória. Com relação à ferramenta proposta LabCODE, devido à dificuldade de implementação encontrada até aqui, é provável, como apontado, que ela não poderá cobrir todos os *code smells* identificados na literatura pesquisada até aqui.

5. Trabalhos Futuros

Do ponto de vista conceitual, planeja-se um estudo mais aprofundado das relações entre análise estática de código e táticas de refatoração, especialmente em aplicações realizadas com linguagem de programação gráfica. Para tanto, prevê-se uma revisão sistemática da literatura relacionada, bem como de soluções de implementação de ferramentas de refatoração existentes aplicáveis ao caso em pauta. O objetivo desse estudo é o de tornar mais acurados os algoritmos de detecção dos tipos de *code smells*.

Os resultados desse estudo serão incorporados na evolução da versão da ferramenta LabCODE, cujas características da versão inicial de seu protótipo experimental foram aqui descritas. Uma vez implementada a ferramenta, ela será submetida a uma investigação empírica de sua capacidade operacional para analisar e corrigir problemas de codificação em LabVIEW. Vislumbram-se duas possibilidades: um estudo de caso ou um experimento.

No estudo de caso, para a seleção do caso devem ser considerados usuários iniciantes na programação em LabVIEW que atuam em organizações que empregam esse ambiente de desenvolvimento. Para a unidade de análise, o estudo de caso deverá selecionar e propor um conjunto de projetos submetidos aos participantes. Esses projetos apresentam problemas variados de *code smells* de diferentes graus de complexidade.

No experimento, os participantes devem ter o mesmo perfil dos participantes da alternativa de estudo de caso. Ao grupo de controle, submete-se um problema exemplo sem o uso da ferramenta, para que os participantes identifiquem os problemas de codificação. Ao grupo experimental, submete-se o mesmo problema, nesse caso utilizando a ferramenta LabCODE para análise e refatoração do código para fins de comparação.

Nas duas alternativas, a coleta e análise de dados é feita por um questionário para a avaliação da ferramenta, impressões, sugestões de melhorias e outros fatores selecionados.

Referências

Abdullah Mohd Zin. (2011). Block-Based Approach for End-User Software Development. *Asian Journal of Information Technology*, *10*(6), 249–258. https://doi.org/10.3923/ajit.2011.249.258

Amoroso, D. L., & Cheney, P. H. (1992). Quality End User-Developed Applications: Some Essential Ingredients. *ACM SIGMIS Database*, *23*(1), 1–11. https://doi.org/10.1145/134347.134350

Baqais, A. A. B., & Alshayeb, M. (2020). Automatic software refactoring: a systematic literature review. *Software Quality Journal*, *28*(2), 459–502. https://doi.org/10.1007/s11219-019-09477-y

Blume, P. A. (2007). *The labVIEW style book* (1st ed.). Pearson Education.

Chambers, C., & Scaffidi, C. (2013). Smell-driven performance analysis for end-user programmers. *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, 159–166. https://doi.org/10.1109/VLHCC.2013.6645261

Costabile, M. F., Mussio, P., Provenza, L. P., & Piccinno, A. (2008). End users as unwitting software developers. *Proceedings - International Conference on Software Engineering*, 6–10. https://doi.org/10.1145/1370847.1370849

Fowler, M. (2019). *Refatoração Aperfeiçoando o design de códigos existentes* (R. Prates (Ed.); 2nd ed.). Novatec Editora.

Henley, A. Z., & Fleming, S. D. (2016). Yestercode: Improving code-change support in visual dataflow programming environments. *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, *2016-Novem*, 106–114.

https://doi.org/10.1109/VLHCC.2016.7739672

Herbold, S., Grabowski, J., & Neukirchen, H. (2009). Automated refactoring suggestions using the results of code analysis tools. *1st International Conference on Advances in System Testing and Validation Lifecycle, VALID 2009*, *October*, 104–109. https://doi.org/10.1109/VALID.2009.12

Kaur, G., & Singh, B. (2017). Improving the quality of software by refactoring. *Proceedings of the 2017 International Conference on Intelligent Computing and Control Systems, ICICCS 2017*, *2018-Janua*, 185–191. https://doi.org/10.1109/ICCONS.2017.8250707

Kerry, E., & Snyder, D. (2010). Comparing LabVIEW graphical code to text-based alternatives for use in test applications. *AUTOTESTCON (Proceedings)*, 245–246. https://doi.org/10.1109/AUTEST.2010.5613603

Kuttal, S. K., Sarma, A., & Rothermel, G. (2014). On the benefits of providing versioning support for end users: An empirical study. *ACM Transactions on Computer-Human Interaction*, *21*(2). https://doi.org/10.1145/2560016

LabVIEW Development Guidelines (1st ed.). (2003). National Instruments. https://www.ni.com/pdf/manuals/321393d.pdf

Makkar, P., Sikka, S., & Malhotra, A. (2021). A Multi-Objective Approach for Software Quality Improvement. *Journal of Physics: Conference Series*, *1950*(1). https://doi.org/10.1088/1742-6596/1950/1/012068

Martin, R. C. (2018). *Arquitetura Limpa - O Guia do Artesão para Estrutura e Design de Software* (1st ed.). Alta Books.

Mkaouer, M. W., Kessentini, M., Bechikh, S., Ó'Cinnéide, M., & Deb, K. (2014). *Software refactoring under uncertainty*. 187–188. https://doi.org/10.1145/2598394.2598499

Popoola, S., Zhao, X., & Gray, J. (2021). Evolution of Bad Smells in LabVIEW Graphical Models. *Journal of Object Technology*, *20*(1), 1–15. https://doi.org/10.5381/jot.2021.20.1.a1

Scaffidi, C., Shaw, M., & Myers, B. (2005). Estimating the numbers of end users and end user programmers. *Proceedings - 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, *2005*, 207–214. https://doi.org/10.1109/VLHCC.2005.34

Sobrinho, E. V. D. P., De Lucia, A., & Maia, M. D. A. (2021). A Systematic Literature Review on Bad Smells-5 W's: Which, When, What, Who, Where. *IEEE Transactions on Software Engineering*, *47*(1), 17–66. https://doi.org/10.1109/TSE.2018.2880977

Szoke, G., Nagy, C., Fulop, L. J., Ferenc, R., & Gyimothy, T. (2015). FaultBuster: An automatic code smell refactoring toolset. *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation, SCAM 2015 - Proceedings*, 253–258. https://doi.org/10.1109/SCAM.2015.7335422

Verebi, I. (2015). A model-based approach to software refactoring. *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings*, 606–609. https://doi.org/10.1109/ICSM.2015.7332524

Zhao, X., & Gray, J. (2019). BESMER: An approach for bad smells summarization in systems models. *Proceedings - 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion, MODELS-C 2019*, 304–313. https://doi.org/10.1109/MODELS-C.2019.00047

Zhao, X., Gray, J., & Riché, T. (2021). A Survey-Based Empirical Evaluation of Bad Smells in LabVIEW Systems Models. *Proceedings - 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021*, 177–188. https://doi.org/10.1109/SANER50967.2021.00025