UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
CENTRO MULTIDISCIPLINAR DE ANGICOS
DEPARTAMENTO DE CIÊNCIAS EXATAS E TECNOLOGIA DA INFORMAÇÃO
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

KALIELSON MATHEUS DE SOUZA

UTILIZAÇÃO DE *LÍNTER* COMO OPÇÃO PARA AUTOMATIZAR A REFATORAÇÃO DE SISTEMAS LEGADOS

ANGICOS - RN
2024

KALIELSON MATHEUS DE SOUZA

UTILIZAÇÃO DE *LÍNTER* COMO OPÇÃO PARA AUTOMATIZAR A REFATORAÇÃO DE SISTEMAS LEGADOS

Trabalho de Conclusão de Curso apresentado a Universidade Federal Rural do Semi-Árido como requisito para obtenção do título de Bacharel em Sistemas de Informação, sob orientação do Prof. Dr. Francisco de Assis Pereira Vasconcelos de Arruda.

ANGICOS

Outubro / 2024

SS729u Souza, Kalielson.
UTILIZAÇÃO DE LÍNTER COMO OPÇÃO PARA AUTOMATIZAR A REFATORAÇÃO DE SISTEMAS LEGADOS / Kalielson Souza. - 2024.
46 f. : il.

Orientador: Francisco Arruda.
Monografia (graduação) - Universidade Federal Rural do Semi-árido, Curso de Sistemas de Informação, 2024.

1. análise . 2. estática. 3. sugestões. 4. débito . 5. técnico. I. Arruda, Francisco , orient. II. Título.

Ficha catalográfica elaborada por sistema gerador automáto em conformidade com AACR2 e os dados fornecidos pelo) autor(a).
Biblioteca Campus Angicos
Bibliotecário: Helder Romero Maia Duarte
CRB: 15/673

O serviço de Geração Automática de Ficha Catalográfica para Trabalhos de Conclusão de Curso (TCC´s) foi desenvolvido pelo Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo (USP) e gentilmente cedido para o Sistema de Bibliotecas da Universidade Federal Rural do Semi-Árido (SISBI-UFERSA), sendo customizado pela Superintendência de Tecnologia da Informação e Comunicação (SUTIC) sob orientação dos bibliotecários da instituição para ser adaptado às necessidades dos alunos dos Cursos de Graduação e Programas de Pós-Graduação da Universidade.

KALIELSON MATHEUS DE SOUZA

UTILIZAÇÃO DE *LÍNTER* COMO OPÇÃO PARA AUTOMATIZAR A REFATORAÇÃO DE SISTEMAS LEGADOS.

Trabalho de conclusão de curso apresentado a Universidade Federal Rural do Semi-Árido como requisito para obtenção do título de Bacharel em Sistemas de Informação, sob orientação do Prof. Dr. Francisco de Assis P. V. de Arruda.

Defendida em: 25/10/2024

BANCA EXAMINADORA

Prof. Dr. Francisco de Assis Pereira Vasconcelos de Arruda (UFERSA)
Orientador-Presidente

Profa. Dra. Joêmia Leilane Gomes de Medeiros (UFERSA)
Membro Examinador

Profa. Dra. Thatiana Cunha Navarro Diniz (UFERSA)
Membro Examinador

RESUMO

Este trabalho apresenta a utilização de línters como ferramenta para automatizar a refatoração de sistemas por meio da análise estática do código-fonte. A análise estática é discutida como uma abordagem eficaz para melhorar a qualidade do código, permitindo a identificação de padrões inadequados e a proposição de ajustes de forma eficiente. A implementação do línter em um sistema de testes demonstrou sua capacidade de detectar problemas que, sem essa ferramenta, exigiriam maior esforço manual. Além disso, foram abordadas as necessidades de realização de testes em bases de código maiores para validar a eficácia da abordagem. Para trabalhos futuros, sugere-se a modularização do linter, permitindo sua adaptação a diferentes contextos de desenvolvimento, bem como a exploração de novas métricas e regras de refatoração para uma análise mais abrangente do código. Conclui-se que a utilização de análises estáticas, combinadas com uma abordagem automatizada, pode resultar em um sistema mais confiável e de fácil manutenção..

Palavras-chave: análise estática; sugestões; débito técnico;

ABSTRACT

This work presents the use of linters as a tool to automate the refactoring of systems through static code analysis. Static analysis is discussed as an effective approach to improve code quality, enabling the identification of inadequate patterns and the efficient proposition of adjustments. The implementation of the linter in a test system demonstrated its ability to detect issues that would require greater manual effort without this tool. Furthermore, the need for testing on larger codebases was addressed to validate the effectiveness of the approach. For future work, it is suggested to modularize the linter, allowing its adaptation to different development contexts, as well as exploring new metrics and refactoring rules for a more comprehensive analysis of the code. It is concluded that the use of static analyses, combined with an automated approach, can result in a more reliable and maintainable system.

Keywords: refactoring ; technical debt ; static code analysis;

LISTA DE QUADROS

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

Com a popularização dos *softwares* e a dependência gerada por eles no mercado, ocorreu um impulso gigantesco na quantidade de código e de programas de computadores produzidos ao longo dos anos. Com tal impulso, foi natural que surgissem técnicas de desenvolvimento de que representam abstrações dos *softwares* desenvolvidos, e são utilizadas comumente para otimizar o seu processo de produção. Tão natural quanto o surgimento dessas técnicas foi o caminhar destas para uma maior especialização.

Dentro desse amplo conceito que abrange a engenharia de *software*, envolvendo uma variedade de técnicas e ferramentas para o desenvolvimento de sistemas de tecnologia da informação, a área de foco específico deste trabalho é a qualidade de *software*. Inicialmente, a definição precisa do que seria qualidade apresenta desafios, uma vez que o próprio processo de definir qualidade deriva de uma especialização dos métodos internamente empregados pelas equipes de desenvolvimento.

A adaptação desses procedimentos para melhorar a qualidade do código produzido, embora possuam similaridades com outros contextos onde a engenharia de software está empregada e os quais estão intrinsecamente ligados ao contexto do que está sendo empreendido, está tão entrelaçada com os objetivos do projeto que a originou, que moldar seus requisitos se torna tão específico para aquela empreitada que sua aplicação pode perder sua relevância ao tentar definir qualidade em um cenário distinto; assim, para contornar essa situação, diversos especialistas em qualidade de *software* têm adotado uma abordagem de avaliar a qualidade por meio de métricas primárias que possam ser medidas e reproduzidas em diferentes projetos. Essas métricas são designadas como primárias devido ao fato de que, à primeira vista, representam as características comuns identificáveis entre projetos de *software* de diferentes naturezas.

Essas métricas primárias podem variar em nome ou descrição, mas de acordo com as pesquisas realizadas nesse trabalho e em consulta a literatura especializada, vê-se comumente as seguintes referências, por vezes repetidas: a adequação, a qual se refere o alinhamento do que se está implementado na funcionalidade e o que a mesma tem como apropriado; a eficiência de desempenho, a qual engloba todos os recursos utilizados para manter o funcionamento pleno de todas as funcionalidades; a usabilidade, que valida o nível de conhecimento necessário tanto para utilizar a aplicação quanto para entender seu funcionamento; a confiabilidade, que verifica se os resultados gerados pelas funcionalidades estão de acordo com o especificado; a

manutenibilidade, que como parâmetro de qualidade engloba os quesitos relacionados a como o produto de *software* se relaciona com as alterações vindouras a sua entrega, sejam elas correções, ou implementações de novas funcionalidades. Cada uma dessas métricas representam parâmetros do *software*, produzindo estimativas que podem ser utilizadas como uma forma de assegurar a sua qualidade.

Considerando as métricas e sua interpretação, compreende-se que o processo de identificação desses pontos pode ser complexo e custoso, semelhante a qualquer análise de código. Por exemplo, a métrica de manutenibilidade abrange o conceito da refatoração do código fonte. A refatoração consiste em reestruturar ou remodelar o código-fonte existente, sem alterar seu comportamento externo, visando a melhoria da legibilidade, aumento da manutenibilidade, a simplificação e a redução da quantidade de problemas/bugs.

Para melhorar a métrica manutenibilidade por meio da refatoração, existem ferramentas auxiliares que fazem a leitura e análise do código fonte de forma estática. Nesse sentido, surgiram ferramentas específicas de análise estática, usadas especificamente para verificação do estilo de codificação e a conformidade com um conjunto de regras predefinidas. Essas ferramentas são conhecidas como *línters*. Essas ferramentas funcionam de maneira semelhante aos corretores ortográficos, destacando inconformidades de acordo com a sintaxe da linguagem de programação utilizada. Além de *línters*, outras ferramentas como perfilhadores (*profilers*), ferramentas de revisão de código e métricas de código complementam a análise estática, mas que fogem ao escopo deste trabalho.

Os *línters* se concentram na análise estática de código e são amplamente utilizados no desenvolvimento de *software*, desempenhando um papel fundamental na melhoria da qualidade do código-fonte. Os *línters* podem ser utilizados para identificar erros, problemas de estilo, inconsistências e possíveis vulnerabilidades no código-fonte. Ao examinar o código em busca de conformidade com diretrizes de estilo e boas práticas de programação, o *línter* contribui significativamente para a manutenibilidade, legibilidade e confiabilidade. No entanto, o escopo dos *línters* não se limita apenas a isso; eles podem ser expandidos para verificar aspectos de estilo, como espaços inadequados, indentação ou o uso de aspas simples e compostas, abrangendo uma ampla gama de elementos no código.

Adicionalmente, os *línters* desempenham funções cruciais além da detecção de erros e inconsistências de estilo. Eles realizam análises de performance, identificando trechos de código que podem ser otimizados, e verificam padrões que representam vulnerabilidades de

segurança. Também avaliam a complexidade do código, destacando métodos que necessitam de refatoração. Além disso, analisam dependências desatualizadas ou não utilizadas, asseguram que o código está devidamente documentado e aderente às convenções do projeto.

Diante do exposto, este trabalho busca explorar o uso de *línters* como uma ferramenta valiosa na avaliação da qualidade do *software*, especialmente no contexto da manutenibilidade, e justifica sua relevância no campo da engenharia de *software*, apresentada a seguir.

1.1 JUSTIFICATIVA

Na prática, a grande maioria dos sistemas produzidos tem ou terão em algum ponto do seu processo de desenvolvimento algum tipo de débito técnico vestigial. Débito técnico é o compromisso assumido ao optar por soluções rápidas, e menos ideais durante o desenvolvimento de *software*, o que pode incluir atalhos no código, falta de documentação e testes insuficientes. Embora permita um progresso inicial rápido, pode resultar em maior complexidade, bugs e dificuldades futuras na manutenção e evolução do sistema.

O débito técnico ocorre pelo fato da exigência crescente de um desenvolvimento de *software* cada vez mais complexos, e produzidos a velocidades maiores. O débito é agravado ainda por outros fatores, tais quais: priorização de prazos ao invés da qualidade, falta de planejamento ou planejamento otimista, mudanças constantes de requisitos ou aderência aos requisitos do cliente sem priorização, ausência de testes ou testes insuficientes e ainda a pressão pelo uso de novas tecnologias ainda imaturas. É compreensível que em algum ponto dessa trajetória a logística por trás desses processos precise fazer concessões, o que podem afetar a qualidade do *software* desenvolvido.

O débito técnico afeta a qualidade do código, a usabilidade ou a eficiência, pois foi adquirido para atender a prazos apertados ou requisitos imediatos. Muitas vezes sendo esse aspecto relacionado a qualidade de manutenção, a qual corresponde a característica adquirida do *software* que o permite ser modificado ou adaptado. Essa escolha favorece os prazos e preza as entregas de curto prazo, porém dificulta em um segundo instante o aprimoramento desse mesmo sistema, criando pontos de código nos quais as escolhas empregadas na arquitetura geraram um trabalho de escrita e reescrita do código, resultando em um sistema com a complexidade acima do necessário. Essa *bola de neve* dentro das organizações é empurrada para o futuro, com pequenas ações de melhorias isoladas que podem aumentar a complexidade do *software* além do necessário.

A abordagem de realizar melhorias isoladas pode resultar em um incremento marginal na qualidade, porém é necessário ir a fundo para gerar uma mudança considerável o suficiente para suprimir esse débito. No entanto, os métodos tradicionais de melhoria de *software*, embora sejam eficazes, muitas vezes consomem um tempo considerável, o que pode não ser eficaz à vista da gerência do projeto, que pode estar sob pressão de prazos e contar com recursos limitados.

Diante desse cenário, o trabalho apresentado neste documento justifica-se por compreender esses déficits e utilizá-los para compor as bases necessárias para a formulação de regras que se utilizará da análise estática de código-fonte, capaz de identificar padrões passíveis de mudança de forma automatizada, gerando sugestões para refatoração de código legado, sendo assim capaz de reduzir o tempo necessário para melhoria deles e melhorando a qualidade do *software*.

Ao automatizar a identificação e sugestão de refatoração, a ferramenta e as regras utilizadas devem levar a ter-se uma maneira mais ágil e eficaz de lidar com o débito técnico em uma base de código legado, permitindo que a equipe de desenvolvimento tenha resultados mais eficientes e com menos interrupções nos fluxos de trabalho existentes.

1.2 OBJETIVOS

As subseções a seguir contém os objetivos gerais e específicos deste trabalho.

1.2.1 Objetivo Geral

Desenvolver uma ferramenta para análise estática de código escrito na linguagem de programação C# que utiliza métricas de qualidade de *software* para identificar e oferecer sugestões de melhorias no código-fonte.

1.2.2 Objetivos Específicos

- Pesquisar e implementar uma lista de métricas que possam ser utilizadas para averiguar a qualidade de um projeto de *software*.
- Estabelecer um processo de inspeção e categorização no código-fonte.
- Gerar um processo de inspeção e classificação no código.
- Realizar a automação do processo de refatoração.
- Validar as sugestões geradas, por meio de teste em ambiente legado.

1.3 Organização do Documento

Este documento está dividido em cinco capítulos, organizados da seguinte forma: O Capítulo 1 apresenta a introdução, justificativa e objetivos do trabalho, abordando a importância da qualidade de software e o uso de ferramentas como línters para melhorar a manutenibilidade do código. O Capítulo 2 é tratada a fundamentação teórica, explorando conceitos centrais como métricas de qualidade de software, débito técnico e análise estática de código, além de revisar trabalhos relacionados sobre o tema. No Capítulo 3, é descrita a metodologia, incluindo os materiais utilizados e o método proposto para o desenvolvimento da ferramenta de análise estática. O Capítulo 4 apresenta os resultados obtidos com a ferramenta desenvolvida, incluindo as sugestões de refatoração automatizadas e os testes realizados. Por fim, o Capítulo 5 traz as considerações finais, resumindo as principais conclusões do trabalho, as contribuições da ferramenta desenvolvida, suas limitações e possíveis extensões futuras.

2. FUNDAMENTAÇÃO TEÓRICA

Padrões de qualidade para *software* sempre existiram, como exemplo a norma ISO 900X que é um modelo para garantia da qualidade que teve suas primeiras versões lançadas a partir 1987, com o intuito de estabelecer padrões. Junto destes inevitavelmente precisava-se de métricas para que, de forma quantitativa, fossem demonstrados os indicadores da situação da qualidade de um determinado projeto.

Quando se trata de métricas, entende-se que não são apenas os aspectos enumerados e imutáveis. Especificamente, as métricas para um produto podem ser usadas de pelo menos três maneiras: fazer previsões em nível de sistema, identificar antecipadamente componentes de *software* de alto risco e construir diretrizes preventivas de design e programação. Esses usos permitem que uma organização, por exemplo, obtenha uma estimativa antecipada da qualidade e tome medidas antecipadas para reduzir o número de componentes de *software* defeituosos (El-emam, 2001).

Porém, para que realmente consigam fazer essa previsão, as métricas devem ser coerentes com o projeto em questão. Um exemplo a ser citado é com a métrica de tempo de desenvolvimento, que inegavelmente contabiliza o período para se fazer algo, mas pode ser aplicada em diferentes aspectos de acordo com cada projeto, contabilizando tanto o tempo para criação de arquitetura, a codificação, finalização de uma parte do projeto e a criação de cenários de teste.

Existem ainda outras métricas que podem ser levadas em consideração, tais quais o número de erros, a deterioração do *software*, o tamanho do código. “As métricas são os sensores que deixe-nos saber quando nossos projetos e processos estão se desviando, para que possamos trazer eles de volta sob controle.” (Pressman, 2000).

Como observado no texto anterior, as medidas para qualidade sempre estiveram na engenharia de *software*, então deve-se esperar que a prática da refatoração não seja recente, já que livros como “Refactoring: Improving the Design of Existing Code” (Fowler et al., 2018), estabelece discussões sobre a refatoração. A ideia de estabelecer métricas sólidas para a criação de *software* de qualidade não é algo que tenha surgido recentemente, embora tenha sido impulsionada pela grande produção de *software* na atualidade.

Essa premissa tem sido valorizada ao longo do tempo, qual seja a identificação de

parâmetros de qualidade comum para aplicar tanto nas fases de criação tanto de refatoração dos *softwares*. A literatura deixa claro que o esforço para realizar a refatoração em aplicações já existentes é muito maior, e revela a necessidade de um compromisso constante com a melhoria dessas técnicas citadas.

A tentativa da automatização do processo de refatoração é uma jornada que vários trilham com frequência, pode ser citada ferramentas como o *Metrics*[1], que é um plugin para IDE Eclipse que contém um conjunto de regras que são usadas para verificar se o código está de acordo como que se espera. Apesar da quantidade de autores dedicados nessa área de estudo, ainda não se percebe um consenso definitivo sobre os pontos que devem guiar o desenvolvimento de um processo para refatoração automática de *software* com qualidade.

Isso acontece pelo motivo que cada *software* se encontra inserido em seu próprio contexto e desafios específicos. Significa então que, devido à falta de uma fórmula universal, os desenvolvedores têm o espaço para interpretar e adaptar os diversos aspectos dos processos de refatoração às peculiaridades de cada projeto.

Observar a complexidade crescente do cenário de desenvolvimento de *software* é uma necessidade, já que na medida que as aplicações disponíveis se tornam mais poderosas e versáteis, também crescem as demandas por processos de refatoração que possam lidar com essa complexidade de maneira eficaz. Nesse contexto, é preciso fazer algumas concessões, e assumir determinadas nomenclaturas, mesmo que não existam diretrizes definitivas. Algumas das nomenclaturas vistas em trabalhos da área foram:

- SRAs - *Softwares* de refatoração automática.
- Ponto de mudança - quaisquer trechos de código não alinhado com as métricas.

Diante deste cenário, as equipes de desenvolvimento acabaram adotando e difundindo o uso das ferramentas de análise estática de código, também conhecidas como *línter*. Isso, por si só, trouxe uma vantagem significativa para as equipes no contexto da refatoração, já que no decorrer do processo de desenvolvimento, é possível identificar pontos propícios para mudanças, mesmos estes sendo limitados pelas tecnologias usadas no desenvolvimento, trazendo estes pontos a uma simplicidade menor que uma ferramenta mais robusta.

[1] Disponível em: https://metrics2.sourceforge.net/

Estes pontos simples que são abordados pelos *línters* geralmente estão ligados estrutura base do código, como a consistência de sua indentação, a utilização de caracteres especiais, bem como a organização geral das instruções e chamadas de métodos.

Essas regras são importantes para manter um código legível e coerente, mas não abordam necessariamente questões mais profundas da qualidade de *software*. Destacando que quando é referido que um ponto de mudança é simples não se infere o grau de importância, já que erros de sintaxe também causam grandes danos, por isso que a utilização de *línters* tendem a reduzir a necessidade de intervenções de manutenção em etapas posteriores, contribuindo para uma diminuição dos custos associados a tais atividades. Embora os atuais *línters* desempenhem um papel crucial, é importante mais uma vez destacar que boa parte deles está intrinsicamente relacionada às normas sintáticas da linguagem de programação utilizada no *software*.

Por isso que esta abordagem tem algumas limitações quando se pensa em projetos maiores. À medida que diferentes tecnologias e práticas entram em jogo, as abordagens recomendadas para o desenvolvimento também sofrem transformações. Isso significa que os *línters* acabam por não conseguir entender os processos mais abrangentes da qualidade e estruturação de código, que por sua vez podem ir além das regras puramente sintáticas.

No entanto, a busca por uma refatoração eficiente e automatizada demanda uma abordagem mais abrangente que vá além das verificações superficiais de sintaxe. A capacidade de interpretar a lógica inserida no código, avaliar sua coesão com as técnicas utilizadas no agrupamento das instruções, junto aos outros aspectos essenciais estudados na engenharia de *software*.

Torna-se por sua vez indispensável, em um caso ideal, que um *software* de refatoração automática (SRA) deveria ser capaz de se adaptar dinamicamente às mudanças nas práticas de desenvolvimento e nas tecnologias adotadas, para poder oferecer opções de soluções que se encaixe nos quesitos precisão e relevância para aprimorar a qualidade do *software*. Portanto, enquanto as ferramentas de análise estática, como os *línters*, constituem um passo valioso na jornada da refatoração, a busca por soluções mais abrangentes e adaptáveis continua a ser um foco essencial na engenharia de *software* moderna. Diante desse contexto, na seção seguinte são apresentados trabalhos acadêmicos relacionados à esses conceitos, com especial foco para *línters* adequados para análise de código escrito na linguagem C#.

2.1 Trabalhos Relacionados

O primeiro artigo vem ao encontro com o que foi mencionado anteriormente, que as métricas criam um recurso para mensurar alguns aspectos do *software*; porém, isso por si só é um dado que nem sempre pode ser utilizado para interpretar o que é qualidade naquela aplicação. Seguindo essa linha, é necessário que exista uma metodologia para poder atribuir uma relação entre o que foi encontrado e um ponto de melhoria que será necessário para poder aumentar a qualidade do *software* (ZUILHOF, 2021).

O autor propõe uma solução por meio de uma etapa onde a métrica evidenciada é relacionada com um atributo do sistema que seria sua causa. Então, se esse atributo estiver desconforme com os padrões estipulados para o projeto, uma mudança seria feita com o objetivo de ajustar a métrica correspondente, evidenciando o padrão aproximado do que seria ideal.

Por sua vez, este padrão é selecionado através de mais um passo que, usando como base as métricas individuais do projeto com o qual se está trabalhando, utiliza os dados métricos adquiridos no passo anterior e os utiliza como base para criar uma função que quantifica, para que possa classificar os casos semelhantes com o encontrado comumente entre baixo, médio ou acima do esperado. Se caso estiver abaixo do esperado, as soluções de refatoração são implementadas.

Um exemplo citado relacionado à manutenção seria se, dentro do projeto, funções com um número reduzido de linhas e maiores abstrações, sendo compostos por expressões mais concisas, deveriam ser consideradas um ponto positivo. Isso ocorre pois muitas vezes os métodos menores aumentam a facilidade da leitura dentro do seu escopo, mas dificultam o entendimento pleno do que deveria ser descrito ali quando esse mesmo escopo é ampliado para uma base de código maior em que vários desenvolvedores devem dar manutenção.

No segundo artigo abordado, a ideia central é que códigos muito robustos acabam tendo *code clones*, que são fragmentos de códigos similares (YOSHIDA et al., 2019). Isso resulta em um aumento de tamanho de linhas (uma das métricas utilizadas para qualidade). Tendo em vista que uma lógica errada duplicada compreende também erros duplicados. Esses fragmentos não se limitam apenas a partes isoladas, mas também métodos que se completam. Eles podem ser identificados através de fatoração e mesclados em funções/métodos únicos, mas essa refatoração de maneira usual demoraria muito tempo e seria suscetível a erros. Portanto, conseguir encontrar uma maneira de automatizar através do reconhecimento desses *code clones*

seria a forma ideal de tratá-los (YOSHIDA et al., 2019).

Os autores discutem um caso muito parecido com o citado anteriormente, porém ele descreve uma ferramenta ainda mais robusta onde existiria um algoritmo genético capaz de buscar, classificar pontos no código onde as métricas não estariam alinhadas com a qualidade. A identificação desses pontos se daria utilizando algumas técnicas quais podem ser visualizados no Quadro 1**.**

Quadro 1 - Tipos de buscas e suas variações utilizadas pelo algoritmo

TIPO DE BUSCA	VARIAÇÃO
RANDOM SEARCH	*N/A*
HILL CLIMBING SEARCH	*FIRST ASCENT*
	STEEPEST ASCENT
SIMULATED ANNEALING	*N/A*
GENETIC ALGORITHM	*SIMPLE MONO-OBJECTIVE GA*
	NSGA-II
	NSGA-III

Fonte: Arquivo pessoal (09/2022).

Após a decisão do ponto de melhoria é utilizado um algoritmo genético de mutação cruzada, que é chamado assim por ter mapeado diversos genomas, cada um com técnicas de refatoração diferentes. Estes genomas geram descendentes modificados em pequenas ou grandes proporções os casos para otimização de código, e no final, através de outro algoritmo, escolhe-se um dos descentes para integrar a solução (MOHAN; GREER; MCMULLAN, 2019).

Para pôr em prática a ideia, os autores criaram primeiro um algoritmo simples que aleatoriamente escolhia trechos do código e tentava de maneira isolada compreender o trecho realçado e encaixar em alguma das métricas definida anteriormente. Os autores então fizeram a análise completa da saída do algoritmo genético: um problema de otimização da manutenção é encontrado e o *software* mapeava uma solução e depois ramificava em outras soluções, criando assim uma árvore de soluções para fatorações com base em mutações da primeira geração.

Após os passos descritos, ressaltam-se no Quadro 2 algumas das soluções gerados pelo programa, lembrando que não existe ação direta, sendo sugestões do que deveria ser feito, exemplos no.

Quadro 2 - Soluções de refatoração sugeridas pelo algoritmo genético.

CAMPO	MÉTODO	CLASSE
INCREASE FIELD SECURITY	INCREASE METHOD SECURITY	MAKE CLASS FINAL
DECREASE FIELD SECURITY	DECREASE METHOD SECURITY	MAKE CLASS NON FINAL
MAKE FIELD FINAL	MAKE METHOD FINAL	MAKE CLASS ABSTRACT
MAKE FIELD NON FINAL	MAKE METHOD NON FINAL	MAKE CLASS CONCRETE
MAKE FIELD STATIC	MAKE METHOD STATIC	EXTRACT SUBCLASS
MAKE FIELD NON STATIC	MAKE METHOD NON STATIC	COLLAPSE HIERARCHY
MOVE FIELD DOWN	MOVE METHOD DOWN	REMOVE CLASS
MOVE FIELD UP	MOVE METHOD UP	REMOVE INTERFACE
REMOVE FIELD	REMOVE METHOD	

Fonte: Arquivo pessoal (09/2022).

Este artigo abordou a importância da adoção das melhores práticas de programação no ambiente de desenvolvimento, destacando que a falta dessas não afetam apenas a manutenção do código, mas também compromete o desempenho, resultando em gargalos ao longo do código.

O principal desafio enfrentado pelos autores está na ausência de um método nativo para monitorar o cumprimento dessas práticas no ambiente/IDE do Visual Studio. Isso, por sua vez, leva à necessidade de instalar *software*s e plugins de terceiros, que podem se revelar pouco confiáveis e sujeitos a mudanças abruptas em suas políticas. O artigo também enfatiza a consideração da carga na máquina, que aumenta à medida que processos adicionais são executados em segundo plano. Como solução, a escolha de uma ferramenta integrada para analisar diversos aspectos, incluindo estilo de código, possíveis refatorações e navegação, torna-se crucial.

Nesse contexto, a adoção do Microsoft Roslyn[2] se apresentou como uma opção viável, atendendo a todos os requisitos ao integrar-se perfeitamente aos processos da IDE Visual Studio. Além disso, a utilização do Roslyn facilita a compreensão abrangente do código, transformando a dimensionalidade do código de simples instruções em um leque de opções, que incluem gráficos de desempenho e sugestões de inclusões e exclusões de código.

No terceiro artigo (Dubko; Staroletov, 2019) os autores vão mais a fundo no que diz a respeito de perfis de qualidade, assumindo que eles não podem ser completamente avaliados por meio dos métodos mais comuns, sendo necessário que uma checagem adicional seja adicionada ao compilador da IDE. Checagem essa que deve ser feita usando de base conceitos de qualidade da programação moderna, para isso emprega o Roslyn como uma ferramenta para prover uma análise, levando em consideração aspectos que vão além da semântica. Provendo

[2] Disponível em: https://github.com/dotnet/roslyn

um forte ambiente no qual se é capaz de desenvolver funcionalidades totalmente descentralizadas as que já existente ou usá-las de base para criar variações destas.

Por esses motivos, o Roslyn se destacou em relação as outras opções, pois ao utilizá-lo para análise, torna-se possível manipular todas as informações às quais a IDE tem acesso, tais como arquivos, entidades, fontes e modelos de sintaxe, bem como informações semânticas. Esses pontos são essenciais, uma vez que, para uma análise complexa, é fundamental obter uma árvore de sintaxe e um modelo semântico. A árvore de sintaxe é construída a partir do código-fonte de um programa e é usada para conectar diferentes construções linguísticas, enquanto o modelo semântico fornece informações sobre as classes (DUBKO, 2019).

O último artigo abordado começa com a seguinte afirmação: o contexto de cada projeto acaba por interferir em como o desenvolvedor acaba escrevendo código. Os criadores da maioria das linguagens de programação modernas reconhecem esses problemas, e muitas linguagens já vêm com um conjunto de práticas e diretrizes recomendadas para a formatação do código. O principal problema com essas convenções é que existe uma grande diferença entre saber que as diretrizes existem e efetivamente segui-las. Essa dificuldade surge do fato de que os programadores são ensinados a serem preguiçosos (em certos aspectos), e a menos que algo os force a reconhecer que seu código não está seguindo os padrões, eles tendem a adiar qualquer correção até “quando eu tiver tempo” (LINKA, 2019).

Nessa mesma linha de pensamento a ideia que o autor tenta passar é que a forma mais direta de fazer desenvolvedores seguirem convenções, ou refatorarem código, é que ativamente e de maneira automática, sejam lembrados disso. O exemplo usado é que ao escrever um método, e ser lembrado de que os nomes das variáveis são confusos, acaba tornando o tempo para resolver uma quantidade de tempo muito ínfima comparando a fazê-lo quando o programador finalizar a tarefa.

Então como resolver essa problemática? O principal problema tratado no artigo é que o Visual Studio não possui a capacidade de monitorar violações de estilo de código, então os desenvolvedores precisam instalar ferramentas de terceiros (LINKA, 2019), chegando à conclusão que extensões de terceiros podendo ser instaladas, sendo a escolha feita com base na familiaridade do desenvolvedor com as ferramentas.

Os autores ressaltam a necessidade de uma solução centralizada, usando um método que seja capaz de analisar todos os aspectos pertinentes a projetos de desenvolvimento que no texto são elencados nos pontos:

- Refatoração.
- Extensibilidade.
- Extensibilidade.
- Desempenho e Integração com a IDE.
- Validação de estilo.

Para conseguir isso, o autor reforça a escolha do Microsoft Roslyn, que é um projeto de código aberto desenvolvido pela Microsoft para conseguir aproveitar o código de compiladores do C# e VB.NET (originalmente escritos em C++). Além da análise sintática e semântica baseada em árvores, o roslyn utiliza árvores de sintaxe concreta e permite a integração direta com o pipeline de compilação do Visual Studio.

Diante de todo o exposto, a seguir apresenta-se a metodologia adotada, que delineia os passos necessários para a realização do trabalho com o suporte do Microsoft Roslyn.

3. METODOLOGIA

Inicialmente, delineou-se um plano de ação para guiar e estruturar a metodologia de trabalho. A seguir estão listados, em ordem, os passos definidos para a realização do trabalho proposto neste documento:

1. Aprofundar a pesquisa bibliográfica sobre qualidade de *software*, com foco na manutenibilidade e refatoração de programas desenvolvidos na linguagem C#;
2. Definir técnicas para automatizar a refatoração;
3. Escolher tecnologia e linguagem de programação;
4. Escolher ferramenta de suporte para criação do *línter*;
5. Identificar as métricas de qualidade de software;
6. Implementar e testar as métricas com suporte do Roslyn.

A seguir, os passos são detalhados.

3.1 Pesquisa sobre métricas e qualidade de software

Fez-se necessário uma expansão na pesquisa bibliografia sobre qualidade de *software* para que se possa ter uma base sólida dos principais conceitos da área, visando a construção de uma base de conhecimento processual das técnicas mais relevantes na área, especialmente de como identificar e tratar pontos de mudança em código quando o projeto utiliza a linguagem C#.

Por exemplo, Chidamber e Kemerer (1994) descobriram, ao estudar dois sistemas, que a maioria das classes tendia a ter um pequeno número de métodos, sugerindo que a maioria das classes é relativamente simples em sua construção, oferecendo abstrações e funcionalidades específicas (El-emam, 2001). As métricas mais comuns na literatura são baseadas em atributos mensuráveis, sejam eles atributos visíveis de imediato, como o número de linhas em um código, ou o tempo para realizar determinadas ações.

Em relação ao número de linhas, esse atributo é pertinente, pois, em testes, quanto maior for o número de linhas em uma mesma função, é consensual afirmar que essa função engloba várias lógicas dentro da aplicação, e essas estão em um único lugar, responsável por diversos comportamentos, o que torna a manutenção e criação mais difíceis devido à falta de clareza sobre o limite do escopo, transformando-a em uma função “faz-tudo”.

O tempo para realizar ações é uma métrica mais autoexplicativa, sendo o período necessário para completar uma ação dentro da aplicação, como uma consulta a um banco de dados ou o processamento de uma requisição HTTP. Dentro desse mesmo escopo, também podem ser citados os *code clones*, que são trechos de código repetidos de forma idêntica ou que, mesmo com escritas diferentes, essencialmente fazem a mesma coisa.

Existem ainda métricas que não podem ser mensuradas pelos meios comuns, que são as métricas relacionadas às convenções estabelecidas para as práticas de desenvolvimento dentro de um determinado projeto. Essas métricas envolvem aspectos como o estilo de nomeação de métodos e variáveis, indentação e formatação do código, organização e estrutura dos arquivos e diretórios do projeto, aderência aos padrões de documentação, estilo de escrita de comentários, consistência na arquitetura por todo o projeto, tamanho dos módulos e divisão das funcionalidades e reutilização de cada módulo.

Um exemplo de padrões de regras pode ser citado o padrão Capability Maturity Model (CMM) que é um modelo que pode ser utilizado para avaliar e melhorar os processos de desenvolvimento de software em empresas, criado pelo Software Engineering Institute (SEI), que inclui algumas diretrizes que não se concentram especificamente no tratamento do código-fonte. Aproveitando essas diretrizes, a proposta apresentada neste documento visa um meio de transformar as diretrizes ligadas ao tratamento direto do código em diagnósticos para facilitar a implementação desses padrões em bases de código legado. Ferramentas como línters são capazes de automatizar a verificação de conformidade com essas diretrizes, ajudando a identificar e corrigir problemas no código que possam afetar a qualidade do sistema.

3.2 Definição das técnicas para automatizar a refatoração

Durante a definição foram selecionadas mais de uma métrica, as quais estão organizadas em diversas etapas. A primeira etapa tem como foco a escolha de um trecho de código que apresente baixa qualidade. A partir deste trecho, cria-se uma regra baseada em uma métrica de qualidade que seja relevante para o projeto, relacionando-a a um atributo mensurável, como o número de linhas, com o objetivo de estabelecer uma regra para a análise estática. Inicialmente, como já mencionado, são identificados trechos de código que estão desalinhados com as métricas escolhidas pela equipe. Com base nesse código, deve-se criar uma regra que justifique por que o código não atende às expectativas.

Prosseguindo o próximo passo é a criação da regra em linguagem humana de como aquele código não se adequa e definindo os atributos de medição do trecho escolhido de código,

definindo a regra o atributo é o que de forma mensurável o que define aquele código como pobre em qualidade de maneira que se pode traduzir para a análise estática. Estes que servem como guia para a etapa de criação dos diagnósticos.

Então, utilizando critérios criados desses mesmos conjuntos para determinar se o trecho escolhido atende ou não às boas práticas, os pontos identificados são tratados por meio da refatoração, visando a conformidade com as melhores práticas.

3.3 Escolha da linguagem e ferramental de suporte

Concluídos os passos anteriores, inicia-se a busca por uma tecnologia para apoiar o desenvolvimento da ferramenta. A escolha desta tecnologia levou em consideração certos critérios, especialmente devido ao desafio inicial que o desenvolvimento de *línters* representa.

Isso ocorre porque, tradicionalmente, nos *línters* "o problema central com o lint é o empacotamento das informações que ele coleta. Existem muitas opções que servem apenas para desativar ou modificar ligeiramente certos recursos. Há pressões para adicionar ainda mais dessas opções." (Johnson, 1976, p. 9) em quesitos como otimização do uso de memória, instruções mais claras, quando comparadas com as ferramentas de desenvolvimento mais populares. Como resultado, os desenvolvedores muitas vezes se encontram em ambientes de alto nível, o que pode aumentar a dificuldade de integração dos *línters*.

No decorrer da pesquisa para seleção da tecnologia a ser utilizada no projeto, diversos critérios foram definidos, alinhados com os objetivos estabelecidos. Inicialmente, foi destacada a necessidade de compatibilidade com a linguagem de programação C#, com o intuito de possibilitar a integração com ela para análise dos padrões de codificação. Além disso, a importância da flexibilidade foi considerada, uma vez que cada projeto é realizado em um contexto distinto, exigindo capacidade de adaptação a diferentes *software*s. Não menos relevante, a existência de uma comunidade ativa e um suporte robusto foram avaliados como aspectos essenciais na escolha da tecnologia a ser empregada no desenvolvimento do projeto.

Por fim, caso a tecnologia atendesse aos critérios mencionados acima, sua performance seria analisada para identificar se poderia haver impacto negativo. Caso contrário, a tecnologia identificada seria a escolha para este trabalho.

Dentre as possíveis escolhas, aquela que mais satisfaz os pontos mencionados anteriormente é o Roslyn[3], um conjunto de diversas ferramentas para análise de código criado

[3] Disponível em: https://github.com/dotnet/roslyn

pela Microsoft em 2018. O projeto, que é de código aberto, possibilita a análise da estrutura do projeto e do código-fonte, utilizando os padrões de análise léxica, sintática e semântica. Todo esse processo é realizado utilizando a linguagem C# para o desenvolvimento, o que contribui para resolver muitos dos desafios relacionados à criação de um *línter*, promovendo eficiência e qualidade no processo.

3.4 Identificação das métricas

Para a identificação das métricas, cada projeto terá um conceito único que pode variar, sendo necessário um estudo do que se espera alcançar com a refatoração proposta. Contudo, para exemplificar esse trabalho, primeiramente tentaremos começar com a estruturação.

Então para isso é preciso que consigamos entender os limites para facilitar a adição de novos recursos. Para que isso seja possível, precisamos identificar os pontos no código onde não seja possível definir bem o objetivo para aplicar a métrica. Como identificamos que, para estruturar o código, é necessário desacoplar esses trechos, vamos relacionar essa métrica com o atributo de número de linhas, já que é como podemos medir de maneira quantitativa para poder automatizar a tarefa de identificação.

Seguindo como guia a estruturação, temos já a contagem do tamanho das funções, e, dentro disso, pode ser feita a verificação dentro dessas funções imensas de instruções igualmente grandes. Com essa etapa realizada, podemos ir para outras métricas, como identificar lógicas demoradas, ou continuar a estruturação através de mudanças relacionadas ao estilo, como as normas para criação e nomeação de instruções e variáveis.

3.5 Materiais

O ponto inicial para o desenvolvimento de uma ferramenta eficiente de refatoração é a seleção e refinamento das escolhas dos materiais que serão utilizadas na aplicação. Em resumo de como proceder com esta escolha, deve se entender que esta deve permitir em algum ponto utilizar alguma das abordagens da análise de código estática. Algumas técnicas de análise estática incluem:

Verificação por estilo, segundo Fowler (1999) deve ser feito o uso de métodos pequenos para dividir processos complexos. Se feito de forma inadequada, isso pode causar uma grande confusão ao tentar descobrir o que todos esses pequenos métodos fazem. A chave para evitar essa confusão está no nome dos métodos. Eles devem ser nomeados de forma que comuniquem claramente sua intenção. (FOWLER, 1999, p. 210). A verificação por estilo busca este tipo de

falhas na formatação do texto do código como algo mais próxima as regras de gramática como espaços erradas, indentação fora do lugar, e se a descrição condiz com que o trecho faz. Boas práticas “que consiste em realizar uma varredura onde se aplicam uma gama de regras com o objetivo de verificar se as práticas corretas estão sendo usadas para evitar a duplicação de código, tamanho de classes, métodos e parâmetros, a criação desnecessária de variáveis locais” (Comis et al., 2022, p. 3) fortemente influenciadas por autores como Kent Beck, autor do livro "TDD: Desenvolvimento Guiado por Testes" (Beck, 2013, p.104), que destacou em sua obra os efeitos positivos dessas práticas.

Verificação por Boas Práticas, como desenvolvedor provavelmente já foi seriamente atrasado pelo código confuso de outra pessoa. A extensão desse impacto pode ser significativa. Ao longo de um ou dois anos, equipes que eram muito rápidas no início de um projeto podem acabar se movendo a passos de tartaruga. Toda mudança feita no código quebra duas ou três outras partes. Nenhuma alteração é trivial. Cada adição ou modificação ao sistema exige que os emaranhados, torções e nós sejam "compreendidos" para que mais deles possam ser adicionados. Com o tempo, a bagunça se torna tão grande, tão profunda e tão complexa que a equipe não consegue mais limpá-la. Não há jeito algum. (Martin, 2008, p. 35). Diante esse cenário onde em primeiro momento o foco é a velocidade as boas práticas são deixadas de lados, então em algum ponto vai ser necessário revisitar as convenções adotadas pelos desenvolvedores ao logo dos anos. Algumas são mais empíricas, e sabe-se diretamente sem pensar muito como evitar código duplicado, já outras precisam estar diretamente ligados ao desenvolvimento para entender o real ganho, “adicionar estrutura ao programa primeiro e, em seguida, fazer a alteração necessária” (FOWLER, 2018, p.4). Ou seja, a organização da arquitetura é o ponto crucial da boa prática.

Verificação por bugs é a mais direta entre as citadas onde se procurar por bugs verificando a escrita do código por erros comuns, também utiliza o auxílio de *softwares*, para fazerem predições lógicas dos dados. A verificação por bugs que busca defeitos com base na sintaxe, ou lógica do código implementado, sendo uma camada adicional aos interpretadores, e compiladores. Outra abordagem, que adotei em minha pesquisa, é definir e prototipar uma ferramenta de refatoração para verificar se uma refatoração pode ser aplicada de forma segura a um programa e, caso seja possível, realizar a refatoração. Isso evita muitos dos bugs que podem ser introduzidos por erros humanos. (FOWLER, 1999, p. 321). Nesses pontos aonde

iremos refatorar código essa técnica é ainda mais essencial, como visto precisa ter o cuidado para não quebrar o que já existe.

As técnicas mencionadas são resultantes do processo evolutivo da engenharia de *software*, desenvolvidas a partir de diversas pesquisas de autores distintos ao longo dos anos, tendo com primeiro grande marco o lançamento da ISO9000 na década de 80 pelos John T. Rabbitt, partindo dessa era normativa que se instauram com as ISO o livro "Usability Meanings and Interpretations in ISO Standards" (SURYN et al., 2003), que mostra abordagens de como aplicar as regras desses padrões em pontos do desenvolvimento.

Durante a mesma época outras discussões sobre áreas abordadas pela ISO, também tiveram um crescimento, como o QMF, que se trata de um framework para lidar com processos de qualidade, segundo (O'Mahony et al,. 2012) os sistemas de gestão da qualidade são cada vez mais comuns em muitas organizações (Psychogios e Priporas, 2007; Brookes e Becket, 2008). Para aumentar a competitividade e a eficácia, as organizações buscam um nível mais elevado de eficiência em suas funções e processos. Os sistemas de gestão da qualidade são considerados uma intervenção adequada para alcançar esses objetivos (Baidoun, 2003). Diversos frameworks de gestão da qualidade foram propostos; no entanto, o exemplo mais amplamente utilizado é a série ISO 9000.

No QMF a capacidade de medir a qualidade de processos e produtos e ser capaz de atribuir valores quantitativos reais à qualidade do item. Para isso, as definições e conceitos de medida e medição são os seguintes: Métrica é determinar as características ou recursos (extensão, dimensão, quantidade, capacidade e habilidade) de algo, especialmente comparando com um padrão; medição é uma dimensão, capacidade, quantidade ou montante de algo (por exemplo, 300 linhas de código-fonte ou sete páginas de documento de design) (Gordon, 2007, p.6)

O destaque nessa parte é que pode-se ver que nela se pode ter uma noção real do que se tem como métrica, algo que muitos textos após usam de base quando falam no assunto, os conceitos que a ISO e o QMF utilizaram já não eram recentes em sua época, e então vendo que são reproduzidos até hoje, é o que pode ser considerado o mais próximo de assertivo na prática de qualidade.

A escolha dessas técnicas é baseada no propósito de que, quando utilizadas em conjunto, podem auxiliar na automatização da detecção de inconsistências no código em diferentes

escopos. Cada técnica possui sua própria utilidade, como identificação de bugs e inconsistências nos padrões de codificação.

O Roslyn atende a todos esses requisitos estipulados para a escolha da tecnologia, principalmente a compatibilidade, uma vez que seu funcionamento se baseia na divisão do código em pequenos blocos, representativos da estrutura do código C#. Por exemplo, um laço de repetição é dividido em blocos e todos os elementos da linguagem, como palavras-chave, identificadores e operadores, são interpretados como *tokens* de informação. Esses *tokens* são então interligados em uma estrutura de árvore hierárquica, preservando os relacionamentos entre os trechos de código especificados. Após a construção dessa árvore, busca-se compreender o significado do código por meio da atribuição de tipos, referências e análise das regras semânticas.

Após a formação da árvore, o Roslyn disponibiliza para o usuário um objeto manipulável, que pode ser analisado e modificado dependendo do que se espera conseguir da análise no código. Com base no que foi especificado nas regras de compilação, podem ser identificados erros de compilação, otimização de desempenho e pontos de mudança.

3.6 Ambiente de desenvolvimento

A seguir tem-se a descrição do ambiente de desenvolvimento em dois âmbitos diferentes, primeiro o físico, que corresponderá as especificações de hardware da máquina utilizada, e as ferramentas de *software*, compreendendo a IDE (ambiente de desenvolvimento) junto de todos as dependências externas (incluindo bibliotecas). Segue a descrição da máquina utilizada como o ponto de apoio para comparação de alocação de recursos e tempo utilizado para cada tarefa da aplicação:

Quadro 2 - Especificações técnicas da máquina.

CPU	Ryzen 5 2600
GPU	AMD 580X
ARAMAZENAMENTO	King M.2 500GB
MEMÓRIAM RAM	2X8GB 2666Mhz
MEMÓRIA DE VÍDEO	8GB

Fonte: Arquivo pessoal (09/2022).

No ambiente de desenvolvimento será utilizado a IDE Visual Studio Community 2022 para facilitar a instalação, e gerenciamento de módulos necessários.

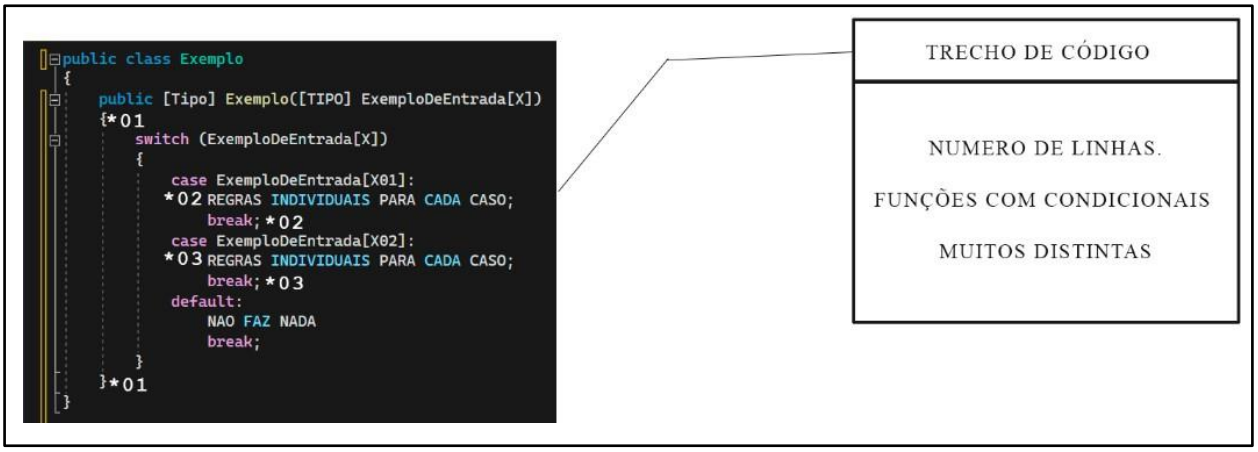
3.7 Modelagem das regras

O próximo passo consiste na escolha de trechos de código do projeto que se deseja refatorar para utilizar as métricas como um guia. Após a escolha, passa-se a escrever um documento com regras que deverão ser transformadas em diagnósticos personalizados. Esse conjunto é composto por trechos de código C# que apresentam codificação que não seguem boas práticas A fim de classificar esses trechos de código de forma adequada, foram desenvolvidas técnicas para metrificar esses pontos.

A escolha desses trechos com pouca qualidade é fundamental para o processo de criação de regras mais eficazes com o Roslyn. Ao utilizar os trechos de código que representam exemplos de código que se deseja classificar, poderão ser diagnósticos com parâmetros mais assertivos para codificação, e quais são os desalinhamentos que devem ser corrigidos.

Como pode ser observado na Figura 1, a ferramenta *línter* tem três pontos de apoio, dos quais se utilizará de base. O primeiro ponto de apoio, destacado como 01, refere-se ao número de linhas dentro da função na classe especificada, acompanhado de um switch com regras individuais (compreende-se como regras individuais qualquer trecho de código) correspondentes aos destaques 02 e 03. Ressalta-se que a modelagem citada é hipotética, sendo uma representação teórica do processo.

Figura 1. Exemplo de funcionamento da leitura do código



Fonte: Arquivo pessoal (09/2022).

No desenvolvimento do sistema de pontuação, um analista atribui uma pontuação entre 0 e 1 para cada parte do código, indicando se está dentro ou fora das expectativas de qualidade. Essa pontuação é fundamental para gerar soluções de refatoração, já que permite o entendimento: 0 representa fora da qualidade desejada, 1 representa dentro da qualidade desejada. Essas informações pontuadas devem gerar um documento de análise, que transfere

para linguagem natural em um documento que descreve como criar essas regras customizadas, que é exemplificada na Figura 2

Figura 2. Documento de diagnóstico

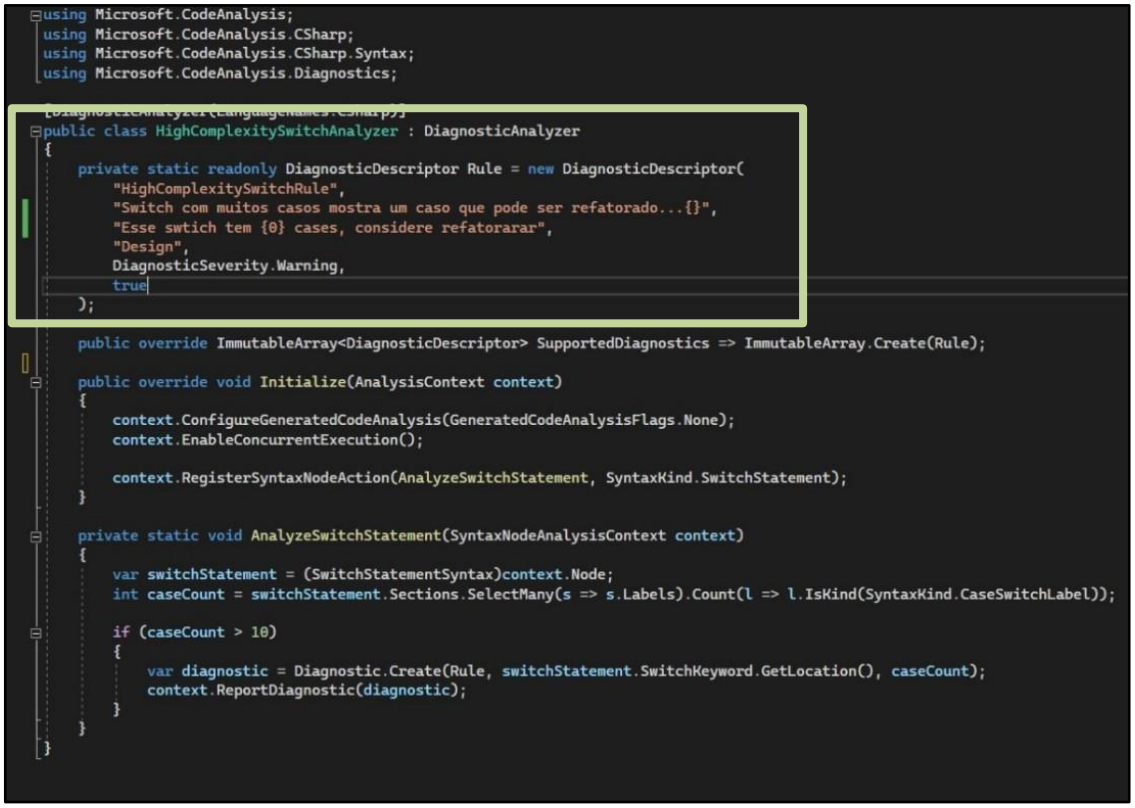
Caso 1
Título: Switch com muitas cases.
Descrição: Switch com cases que se estendem em um espaço com mais de 200 linhas de código.
Existe alternativa: Sim, outros patterns poderiam ser usados para refatorar.
Diagnóstico: Verificar a existência de estruturas condicionais switch com mais de 10 cases implementados.
Mensagem: O switch statement contém mais de 8 casos. Considere refatorar utilizando outra estrutura.
Severidade: Error.

Fonte: Arquivo pessoal (09/2022).

Para modelar as regras é necessário extrair do código trechos que sejam considerados código ruim e partindo desse escolhido devem ser pontuados os aspecos dele quais estão fora da espectativa de qualidade desejada; como por exemplo o pseudo código da Figura 1 onde temos um exemplo de switch como 2 cases individuais, imaginando que no código extraido fosse uma estrutura switch como no exemplo porém com muito mais casos individuais com regras muitos diferentes entre-si, nesse cénario devem ser mapeados esses pontos, uma vez feito com base no mapeamento podemos utilizar para criar as regras, seguindo o exemplo discutindo pode ser feitos algumas analise, mas seguiremos com um fictício no qual foi descrito que as regras serem individuais e não conversarem é visto como um ponto negativo, e com base nessa descrição deve ser nesses casos é devido escolher outra estrutura que possa dividor o escopo das regras para isolar usas responsabilidades.

Após a criação do conjunto de regras para descrição do diagnóstico, é necessário transcrever as regras definidas em linguagem natural para código, utilizando o Roslyn. Traduzem-se os padrões definidos anteriormente para sintaxe em C#, por exemplo, se for especificado que a presença de muitos *cases* em um bloco *switch* é indesejável, pode-se definir essa regra para o Roslyn. Se houver mais casos do que um número especificado, como 10, a regra é acionada e uma mensagem é gerada no compilador. Na Figura 3. Código Roslyn de um analisador personalizado. observa-se a transcrição dessas regras em linguagem natural para codificação utilizando o Roslyn (em amarelo).

Figura 3. Código Roslyn de um analisador personalizado.



```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using Microsoft.CodeAnalysis.Diagnostics;

public class HighComplexitySwitchAnalyzer : DiagnosticAnalyzer
{
    private static readonly DiagnosticDescriptor Rule = new DiagnosticDescriptor(
        "HighComplexitySwitchRule",
        "Switch com muitos casos mostra um caso que pode ser refatorado...{}",
        "Esse swtich tem {0} cases, considere refatorarar",
        "Design",
        DiagnosticSeverity.Warning,
        true
    );

    public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics => ImmutableArray.Create(Rule);

    public override void Initialize(AnalysisContext context)
    {
        context.ConfigureGeneratedCodeAnalysis(GeneratedCodeAnalysisFlags.None);
        context.EnableConcurrentExecution();

        context.RegisterSyntaxNodeAction(AnalyzeSwitchStatement, SyntaxKind.SwitchStatement);
    }

    private static void AnalyzeSwitchStatement(SyntaxNodeAnalysisContext context)
    {
        var switchStatement = (SwitchStatementSyntax)context.Node;
        int caseCount = switchStatement.Sections.SelectMany(s => s.Labels).Count(l => l.IsKind(SyntaxKind.CaseSwitchLabel));

        if (caseCount > 10)
        {
            var diagnostic = Diagnostic.Create(Rule, switchStatement.SwitchKeyword.GetLocation(), caseCount);
            context.ReportDiagnostic(diagnostic);
        }
    }
}
```

Fonte: Arquivo pessoal (08/2023).

Ao concluir a etapa de criação das regras, obtém-se um escopo inicial para o desenvolvimento, pois, neste estágio, o desenvolvedor dispõe de uma amostra do que deve ser convertido em regras de busca para o Roslyn, as quais serão então incorporadas à compilação do projeto. Na próxima seção deste trabalho, é detalhado o processo de criação das referidas regras para a compilação, fornecendo assim uma compreensão mais aprofundada sobre a implementação prática dos conceitos discutidos anteriormente.

3.8 Utilização do Roslyn

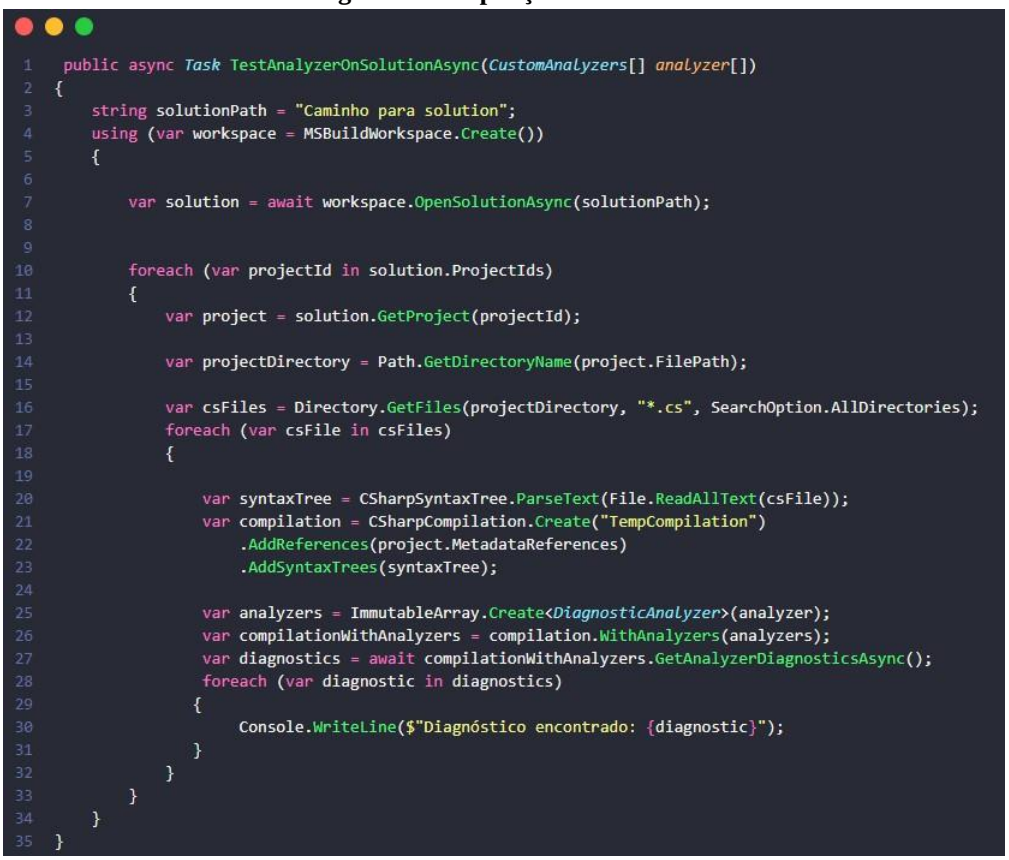
A descrição de regras no Roslyn consiste na criação de uma classe com um "DiagnosticDescriptor". Os parâmetros do DiagnosticDescriptor são:

- **Nome da Regra**: É atribuído um nome único à regra, que a identifica de maneira exclusiva.
- **Descrição Geral**: Uma descrição geral da regra é fornecida, explicando brevemente o que a regra faz ou verifica.
- **Texto de Apresentação**: Um texto de apresentação é especificado, que será exibido quando a regra for violada, fornecendo informações adicionais ao desenvolvedor.

- **Nível de Severidade**: O nível de severidade da regra é definido. Isso pode variar de um aviso, que permite a compilação do código, a um nível impeditivo, que impede a compilação até que a violação seja corrigida.

Um exemplo da definição das regras no momento do build de projeto pode ser visto na Figura 4 onde podem ser vistos os analisadores são injetados. Ao aplicar essas regras, desenvolvedores podem estabelecer orientações personalizadas para a análise do código, o que resulta em uma maior consistência na aplicação de padrões e boas práticas ao longo do desenvolvimento. Além disso, a automação da detecção de padrões contribui significativamente para a redução de erros e aprimoramento da manutenibilidade do código-fonte.

Figura 4. Compilação modificada.



```
public async Task TestAnalyzerOnSolutionAsync(CustomAnalyzers[] analyzer[])
{
    string solutionPath = "Caminho para solution";
    using (var workspace = MSBuildWorkspace.Create())
    {

        var solution = await workspace.OpenSolutionAsync(solutionPath);


        foreach (var projectId in solution.ProjectIds)
        {
            var project = solution.GetProject(projectId);

            var projectDirectory = Path.GetDirectoryName(project.FilePath);

            var csFiles = Directory.GetFiles(projectDirectory, "*.cs", SearchOption.AllDirectories);
            foreach (var csFile in csFiles)
            {

                var syntaxTree = CSharpSyntaxTree.ParseText(File.ReadAllText(csFile));
                var compilation = CSharpCompilation.Create("TempCompilation")
                    .AddReferences(project.MetadataReferences)
                    .AddSyntaxTrees(syntaxTree);

                var analyzers = ImmutableArray.Create<DiagnosticAnalyzer>(analyzer);
                var compilationWithAnalyzers = compilation.WithAnalyzers(analyzers);
                var diagnostics = await compilationWithAnalyzers.GetAnalyzerDiagnosticsAsync();
                foreach (var diagnostic in diagnostics)
               {
                    Console.WriteLine($"Diagnóstico encontrado: {diagnostic}");
               }
            }
        }
    }
}
```

Fonte: Arquivo pessoal (09/2023).

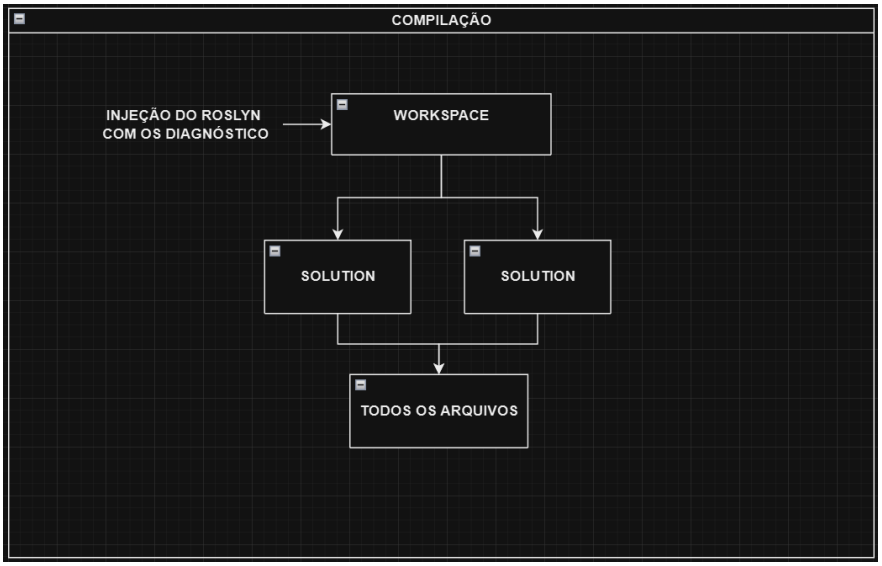
Para alcançar esse objetivo, é necessário realizar algumas modificações em aplicações já desenvolvida em C# .NET. A alteração consiste em indicar no arquivo de inicialização o desejo de adicionar o diagnóstico criado durante a compilação do projeto.

O processo de compilação ocorre da seguinte maneira exemplificada na Figura 5 : o "*workspace*", que representa a maior hierarquia, inicia a execução e build de todas as dependências, fazendo um com o *workspace* com um mediador onde é injetado as regras de compilação padrões do compilador, e as personalizadas que foram criadas chamando-a no

arquivo de inicialização do projeto podendo variar de acordo com a versão do C# na versão utilizadas o arquivo é o "Program.cs” adicionando a classe que contém a chamada dos nossos analisadores personalizados como observado na Figura 3, para orientar a compilação destas, uma vez que essas dependências estejam prontas, o "*workspace*" passa para o projeto repetindo o passo anterior, seguindo um padrão de cima para baixo (*top-down*).

Uma vez que os diagnósticos foram injetados no workspace, durante o processo de *build* estes são chamados, os analisadores personalizados partem de uma abordagem TOP-DOWN partido da maior hierarquia para menor: o workspace inicializa fazendo a chamada dos analisadores criando todas as dependências necessárias para o diagnostico, a qual irão verificar em cada solução atrelada ao workspace os arquivos listados que sejam da extensão “.cs” durante o processo de execução e build das soluções. Esse processo é exemplificado na Figura 5, de forma visual, o funcionamento da compilação que partindo da workspace, *solution* e então todos os arquivos. .

Figura 5. Funcionamento da compilação.



Fonte: Arquivo pessoal (09/2023).

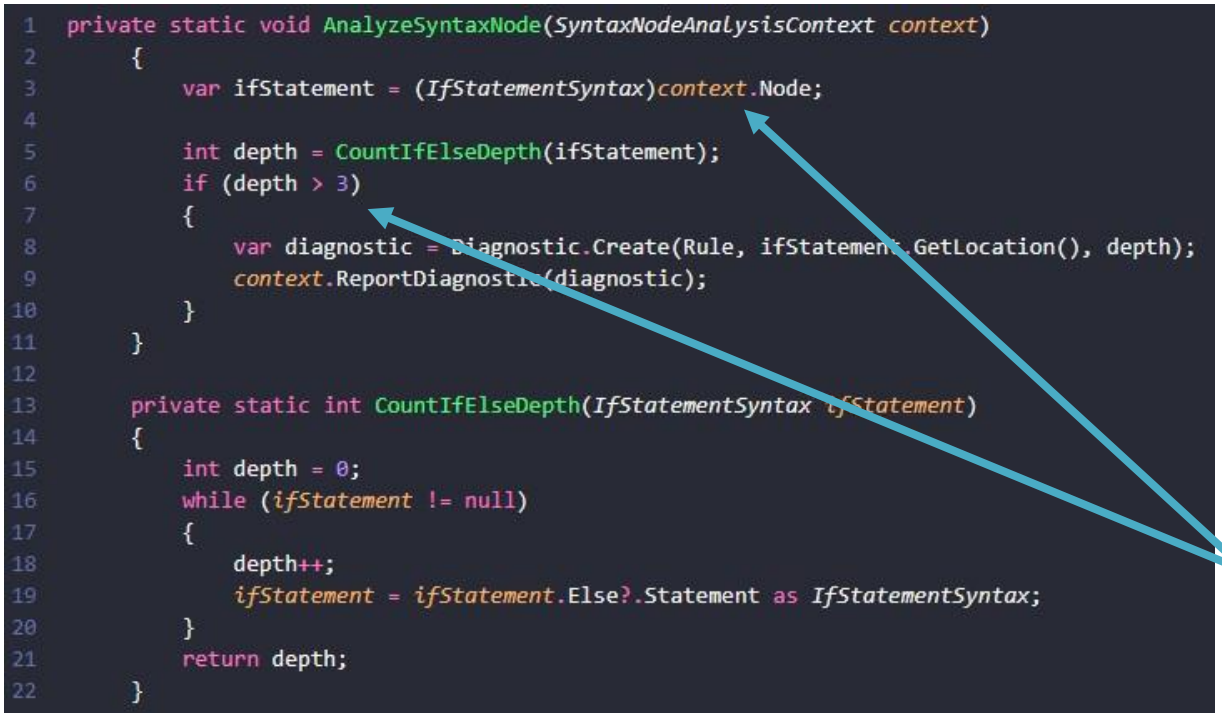
Uma vez identificado o arquivo de código dentro de qualquer hierarquia qual se encaixa no conjunto de regras previstos como passíveis de mudança, o documento é transformado em um objeto ou classe que pode ser manipulado pelos métodos dentro do Roslyn para análise. Esse objeto é decomposto em uma árvore sintática que contém o código em si com as palavras reservadas, funções e tudo que compõe a estrutura do C#, possibilitando a visualização de questões como "IF", "ELSE", "FOREACH" e seus aninhamentos.

A partir de uma modelagem baseada nas características de qualidade específicas do projeto em análise, é possível criar diagnósticos que ofereçam diversas possibilidades para o desenvolvimento de analisadores. No contexto deste estudo, utilizou-se como exemplo a

identificação de aninhamentos excessivos de estruturas condicionais *IF-ELSE* para ilustrar o funcionamento do descritor a ser utilizado no Roslyn.

A análise hipotética realizada, quando há múltiplos casos *IF-ELSE* em um código, é sugerida a possibilidade de substituição por outras estruturas alternativas. Para realizar essa análise, utiliza-se a visualização da sintaxe em árvores disponibilizada pelo Roslyn. O procedimento envolve a solicitação ao analisador para examinar o contexto representado por um nó que, por sua vez, representa o arquivo de código completo. Dessa forma, verifica-se se dentro desse contexto existe um nó correspondente à estrutura de código buscada ("*IF-ELSE*"), possibilitando a contagem de suas declarações, exemplo visual na Figura 6.

Figura 6. Exemplo de node sendo verificado.



Fonte: Arquivo pessoal (09/2023).

Conforme pode ser observado na Figura 6, o context.Node é uma representação rudimentar de alguns aspectos de sintaxe. Ele oferece a capacidade de referenciar objetos contidos dentro dele, permitindo o tratamento de todos os elementos presentes no código-fonte, independentemente de sua complexidade ou estrutura.

Além disso, caso o projeto em análise envolva estruturas personalizadas ou específicas, o context.Node também permite fazer referência a elas. Isso garante uma integração eficaz e precisa dentro do ambiente de desenvolvimento.

Com esse recurso em mente, podemos visualizar que, ao passar o context.Node como um ifstatement, é possível capturar as declarações do bloco de código. Isso inclui verificar, por

exemplo, se as declarações do if continuam sem a declaração final do else, o que possibilita contar a profundidade desse bloco, com isso podemos relacionar esses elementos com as regras construídas para melhoria do código um exemplo do uso pode ser definir uma regra que um aninhamento de x profundidade dispararia um alerta, como visto utilizaríamos esse recurso.

4. RESULTADOS

Para controle dos resultados, foi utilizado um projeto TODO-LIST, e dentro foi produzido código desalinhado com as práticas esperada, então verificado se os logs estão sendo criados como esperado.

Após a implementação da integração com o Roslyn, observou-se que os logs de erros estavam sendo gerados sem a ocorrência de falhas durante a compilação do projeto TODO-LIST[4]. No que diz respeito ao desempenho, não foram identificados aumentos significativos à medida que novos diagnósticos foram adicionados.

Os exemplos apresentados abaixo ilustram cenários de uso comum, sendo um deles com apenas os diagnósticos aplicados e o outro com stress, incluindo todos os diagnósticos em uso paralelo. Ambos os casos demonstraram um aumento no tempo de compilação de aproximadamente 30%. (Teste utilizando o mesmo hardware do Quadro 3).

Quadro 3 - Tempo de compilação.

USO NORMAL	TEMPO DE COMPILAÇÃO
CÓDIGO SEM ROSLYN	4,00s
CÓDIGO COM ROSLYN	5,23s
USO COM STRESS	**TEMPO DE COMPILAÇÃO**
CÓDIGO SEM ROSLYN	6,86s
CÓDIGO COM ROSLYN	8,79s

Fonte: Arquivo pessoal (10/2023).

Os diagnósticos criados são executados durante a compilação do *workspace*, gerando logs com as más práticas detectadas durante a compilação, com um texto descritivos podem ser observados, adicionado durante a criação do *descriptor ruler*.

Figura 7. Exemplo de *warning*.

```
Diagnóstico encontrado: Evite blocos 'if-else' excessivos com outroas estruturas (Acima do máximo permitido: 3).
em E:\REPOAPI\MANAGERTODOAPI\MANAGERTODOAPI\Controllers\UserController.cs, linha 54, coluna 12
```

Fonte: Arquivo pessoal (10/2023).

Como pode ser observado na Figura 7, os logs são gerados com o caminho absoluto até o arquivo, e a linha específica de onde o problema se encontra, indo até o local especificado nos arquivos sendo essa a estrutura dos *warnings* personalizados.

[4] Projeto pessoal disponível em: https://github.com/KalielsonSouza/MANAGERTODOAPI

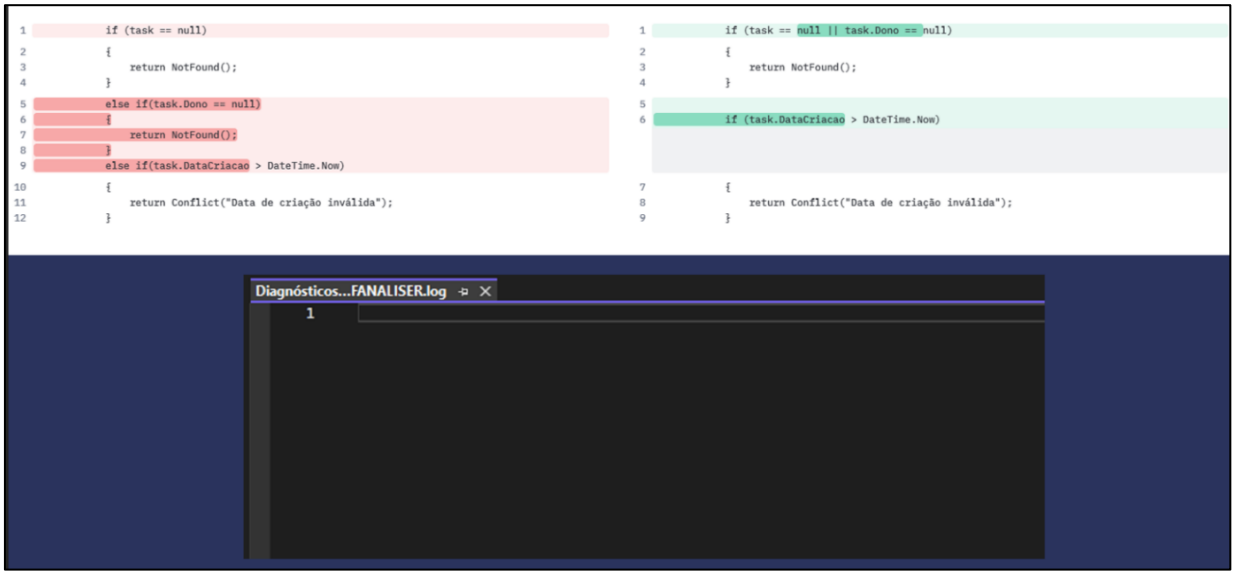
Figura 8. Warning descrito em código.

```
54              if (task == null)
55              {
56                  return NotFound();
57              }
58              else if(task.Dono == null)
59              {
60                  return NotFound();
61              }
62              else if(task.DataCriacao > DateTime.Now)
63              {
64                  return Conflict("Data de criação inválida");
65              }
```

Fonte: Arquivo pessoal (10/2023).

Quando o mesmo arquivo é refatorado com as mudanças requisitadas a descrição log é gerado vazio, já que não existe diagnóstico gerado para aquela regra em particular. Os mesmos resultados foram encontrados para os outros, conforme Figura 9.

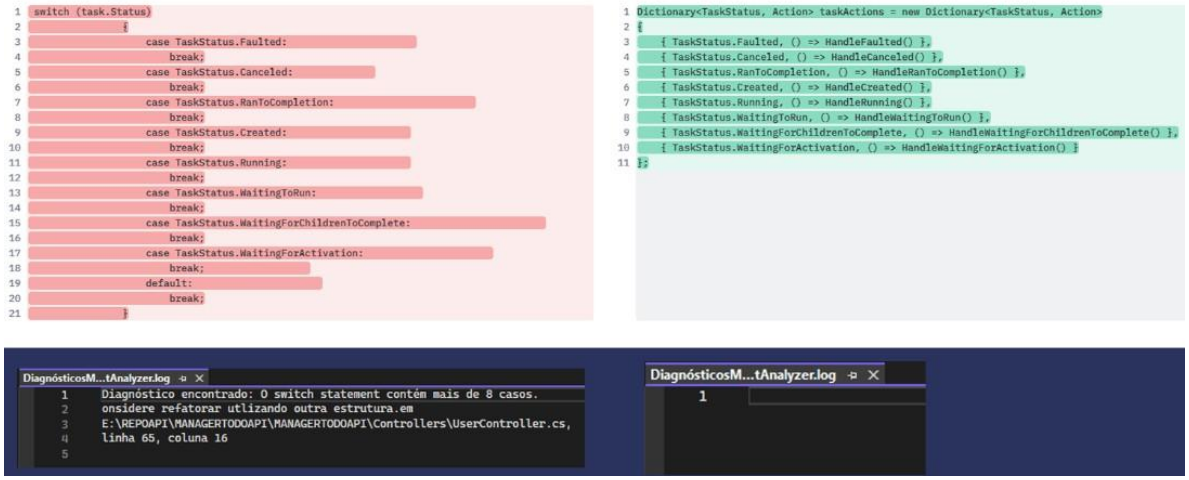
Figura 9. Refatorado.



Fonte: Arquivo pessoal (10/2023).

O mesmo comportamento pôde ser observado utilizando a regra de quantidade de *cases* em um *switch*, onde após o diagnóstico a sua refatoração ocorreu, parando assim de exibir o *warning* e onde não foi acatado ele permaneceu, Figura 10.

Figura 10. Exemplo de switch warning.

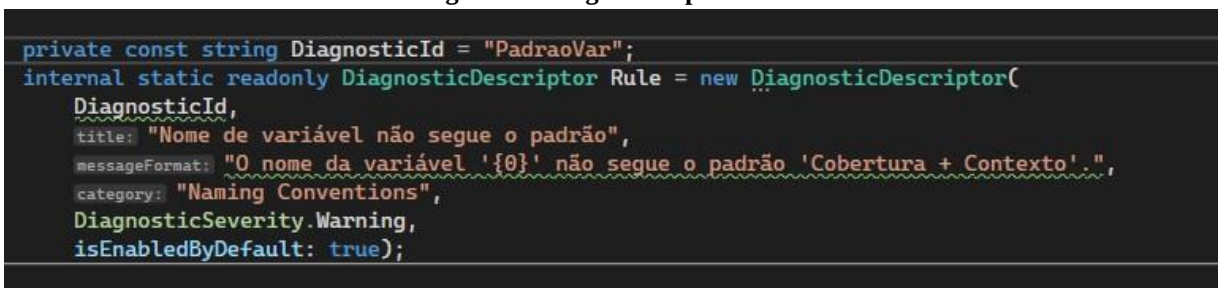


Fonte: Arquivo pessoal (10/2023).

Como observado na Figura 10 o log novamente permaneceu vazio após a alteração da base de código. A descrição detalhada sobre as terminologias está no GLÓSSARIO.

No Roslyn, regras de verificação por estilo também podem ser integradas aos diagnósticos, como exemplificado na Figura 11, onde a regra busca por um padrão no estilo qual não se adequa com a convenção adotada, sendo mais uma camada de verificação adicionada para aumentar a qualidade.

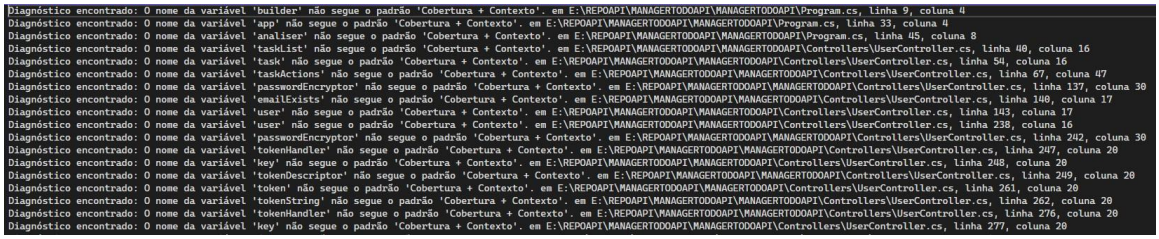
Figura 11. Diagnostico por estilo.



Fonte: Arquivo pessoal (02/2024).

O exemplo mencionado nos testes resultou em registros similares. Entretanto, como previsto, estes são mais abrangentes, pois todas as declarações de variáveis do projeto entram na diagnostico, conforme demonstrado na Figura 12.

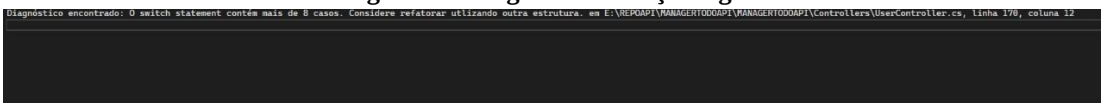
Figura 12. Logs da verificação por estilo.



Fonte: Arquivo pessoal (02/2024).

Por fim, o teste final na base simulada é um diagnóstico mais específico que busca funções legadas que necessitam de mudança, por diversos motivos, desde falhas de segurança até a necessidade de implementação de novos recursos. Entretanto, em diversas situações, é inviável realizar a substituição de todas essas funções de uma só vez, sendo necessário estudar os casos individualmente. Nessa situação, podemos diminuir a severidade do diagnóstico para "info" e apenas obter informações sobre onde essas funções estão sendo utilizadas, o que permite mediar a troca de forma mais eficiente exemplificado na Figura 13.

Figura 13. Diagnósitocs função legado



Fonte: Arquivo pessoal (02/2024).

Encerrados os testes no ambiente simulado, prosseguiram-se os testes em uma base legado real, para verificar se os resultados seriam replicados da mesma forma em um cenário autêntico.No teste da base não simulada, serão compreendidos todos os testes anteriores de maneira individual e, em seguida, todos os diagnósticos serão realizados simultaneamente. Além da verificação da base, o teste também avaliará como o Roslyn se comporta em caso de possíveis erros. Para lidar com estes, cada diagnóstico está inserido em uma estrutura de tratamento de exceções try-catch, de modo que, caso ocorra um possível erro, um registro de log será criado contendo as informações pertinentes ao erro seguindo o esquema dos avisos, incluindo a página, a linha e a descrição do erro, facilitando assim sua identificação. Essa abordagem se mostra necessária, uma vez que será verificada em diversas páginas de código com mais de mil linhas em média. Este teste será utilizado para verificar como será o comportamento e se serão identificados possíveis erros pelo volume de informação processada.

Para a realização deste trabalho, empregou-se uma base de código privada com acesso restrito por questões de confidencialidade, tal restrição se justifica devido à natureza confidencial da base de dados. Esta base é crucial para avaliar a proposta de abordagem de

análise estática de código apresentada neste trabalho, uma vez que consiste em código desenvolvido em ambiente real, em contraposição a códigos sintéticos ou educacionais.

Como mencionado os testes irão começar na mesma ordem do ambiente sintético de maneira individual, no primeiro momento já foi enfrentado um problema de crash com a IDE, qual estava começando a consumir muita memória e processamento até parar de funcionar.

Diante do grande volume de informação gerado ao tentar analisar classes com mais de 20k linhas em média, foram alterados todos os diagnósticos para que apenas um máximo de 200 fossem mostrados por vez, a fim de evitar problema como estouro de memória e falhas, exemplificado na Figura 14.

Figura 14. Limitador de casos.

```
1 referência
private static void AnalyzeSyntaxNode(SyntaxNodeAnalysisContext context)
{
    var SwitchStatementSyntax? switchStatement = (SwitchStatementSyntax)context.Node;
    var int caseCount = switchStatement.Sections.Sum(SwitchSectionSyntax section => section.Labels.Count);

    const int batchSize = 200;
    var int batchCount = (int)Math.Ceiling((double)caseCount / batchSize);

    var Location? location = switchStatement.GetLocation();

    for (int i = 0; i < batchCount; i++)
    {
        var int startIndex = i * batchSize;
        var int endIndex = Math.Min(startIndex + batchSize, caseCount);
        var int count = endIndex - startIndex;

        var Diagnostic? diagnostic = Diagnostic.Create(Rule, location, count.ToString());
        context.ReportDiagnostic(diagnostic);
    }
}
```

Fonte: Arquivo pessoal (02/2024).

Agora com os casos o teste para verificar se são assertivos ou não, será usado as mesmas métricas do ambiente sintético com um ponto a mais que são casos que o diagnóstico acabou causando falhas Da IDE quando removido o limite máximo de 200 diagnósticos, e se mesmo com assim persiste.

Quadro 4 - Resultados dos testes individuais em ambiente não sintético

CLASSE	DESCRIÇÃO	RESULTADO	COM LIMITADOR	SEM LIMITADOR
SwitchAnalyzer	Verifica os casos de switch com muitos cases	Log gerado	Não ocorreu falha	Não ocorreu falha
ElseIfAnalyzer	Verifica casos com aninhamentos excessivos ifs	Log gerado	Não ocorreu falha	Não ocorreu falha
PatternAnalyzer	Verifica casos em que estão declaradas variáveis diferentes do padrão do projeto	Log gerado	Não ocorreu falha	Não ocorreu falha
LegacyAnalyzer	Verifica utilização de funções legados	Log gerado	Não ocorreu falha	Não ocorreu falha

Fonte: Arquivo pessoal (02/2024).

Apesar de com limitador não se obteve mais crashes na aplicação, os logs gerados eram muito grandes deixando muita informação. Com os resultados dos testes individuais completos

o último passo é o teste com todos os diagnósticos ao mesmo tempo, os critérios do teste anterior permanecem.

Ao utilizar todos os diagnósticos juntos, foi notado que o problema do início voltava a ocorrer quando se utilizava todos ao mesmo tempo sem o uso de um limitador, o processamento demorava tempo suficiente para a aplicação parar de funcionar.

Quadro 5 - Resultado do teste om todos os analisadores.

DIAGNÓSTICO	DESCRIÇÃO	RESULTADO	COM LIMITADOR	SEM LIMITADOR
Todos	Todos os analisadores	Ok	Não ocorreu	Ocorreu

Fonte: Arquivo pessoal (02/2024).

Diante do exposto, a análise estática apresentada permitiu a identificação de padrões inadequados, promovendo a correção de erros que, sem essa ferramenta, poderiam passar despercebidos. O desempenho do sistema foi monitorado, e não foram observados aumentos significativos no tempo de compilação, o que indica que a eficiência do processo de desenvolvimento foi mantida. Com base nesses resultados, no próximo capítulo serão discutidas as considerações finais e propostas para trabalhos futuros.

5. CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Neste trabalho foi apresentada a utilização de línters como ferramenta para automatizar a refatoração de sistemas usando análise estática do código fonte, uma abordagem que foi demonstrada como eficaz para melhorar a qualidade do código. A análise estática, conforme discutido no Capítulo 2, permite a identificação de padrões inadequados e a proposição de ajustes de forma eficiente, contribuindo para a manutenção da legibilidade e confiabilidade do código. Foi constatado que a implementação do línter em um sistema de testes contribui para a detecção de problemas que, sem essa abordagem, exigiriam maior esforço manual.

Uma consideração importante é que a solução ainda carece da realização de testes em bases de código maiores, o que poderá validar a eficácia da abordagem desenvolvida, e tais testes deverão incluir averiguação da performance do linter e da capacidade de identificar e sugerir melhorias em cenários mais complexos.

Como trabalho futuro, sugere-se a modularização do linter apresentado, considerada um passo importante para aprimorar o desempenho e reduzir o consumo de recursos. A modularização permitirá que a ferramenta seja injetada em workspaces específicos na ferramenta Visual Studio, facilitando sua adaptação a diferentes contextos de desenvolvimento. Além disso, recomenda-se a exploração de novas métricas e regras de refatoração que possam ser integradas ao línter utilizado, o que permitirá uma análise mais abrangente do código.

A utilização de análises estáticas, combinadas com uma abordagem automatizada, pode melhorar o processo de refatoração, resultando em um sistema mais confiável e de mais fácil manutenção.

REFERÊNCIAS

COMIS, Diego et al. Avaliação das ferramentas de análise estática de códigos PHP: utilizando templates de websites eGov baseados em CMS. Universidade Federal do Pamp, [S. l.], p. 1-10, 22 jun. 2022.

EL-EMAM, Khaled. Object-Oriented Metrics: A Review of Theory and Practice. Canada: National Research Council, 2001. 32 p.

DUBKO, Anatoliy; STAROLETOV, Sergey. A Method to Verify Parallel and Distributed Software in C# by Doing Roslyn AST Transformation to a Promela Mode. Polzunov Altai State Technical University, [S. l.], p. 1-32, 26 jul. 2019.

FOWLER, Martin et al. Refactoring: Improving the Design of Existing Code, 2018.

GUELLEN, Luis Miguel; ZANATTA, Alexandre Lazaretti. Análise e Melhorias no processo de estimativas de tempo no desenvolvimento de software. Ciência da Computação – Universidade de Passo Fundo, [S. l.], p. 1-18, 1 jan. 2010.

JIANHONG, Ma; JING, Yang; TAO, Luo. Quality tree of QFD: A New Method to Ensure Software Quality. World Congress on Software Engineering, [S. l.], p. 53-58, 14 jul. 2009.

JOHNSON, S.C. Lint, a C Program Checker. Bell Laboratories, Murray Hill, p. 2-11, 26 jul. 1976.

MARTIN, Robert Cecil. Código Limpo. 1. ed. [S. l.]: Alta Books, 2008. 425 p. v. 1. ISBN 8576082675.

MARTINS, Renner Costa; PELLEGRINO, Sergio Roberto Matiello; SANTELLANO, Jony. Tratamento de "dead codes" em software de uso aeronáutico. Conferência Brasileira de Dinâmica, Controle e Aplicações, [s. l.], 13 jul. 2010.

MELO, Hugo Faria. Caracterizando os fluxos excepcionais em linhas de produto de software: um estudo exploratório. 2012. TCC (Bacharelado) - Universidade Federal do Rio Grande do Norte Centro de Ciências Exatas e da Terra Departamento de Informática e Matemática Aplicada, [S. l.], 2012.

MOHAN, Michael; GREER, Des; MCMULLAN, Paul. Maximizing Refactoring Coverage in an Automated Maintenance Approach Using Multi-Objective Optimization. IEEE/ACM 3rd International Workshop on Refactoring, [S. l.], p. 31-38, 12 dez. 2019.

PRESSMAN, Roger. Software Engineering: A Practitioner's Approach. 2000.

QIAN, Xiaolei. Semantic Interoperation Via Intelligent Mediation. Computer Science Laboratory SRI International, Menlo Park, CA, p. 228-231, 12 nov. 1993.

SCHNEIDEWIND, Norman F. Discussion of Metrics Validation. Report on the IEEE Standard for a Software Quality Metrics Methodology, Monterey, CA, p. 155-157, 10 abr. 1991.

STAROLETOV, Sergey. A Method to Verify Parallel and Distributed Software in C# by Doing Roslyn. Altai State Technical University, Altai Krai, Rússia, p. 13-28, 17 jul. 2019.

SURYN, Witold; ABRAN, Alain; KHELIFI, Adel. Usability Meanings and Interpretations in ISO Standards. Software Quality Journal, 2003.

ZUILHOF, Bart. Code Quality Metrics for the Functional Side of the Object-Oriented Language C#. Wiskunde en Informatica Master Software Engineering, [S. l.], p. 1-48, 18 abr. 2019.

.

GLÓSSARIO

DiagnosticDescriptor: Uma estrutura de dados que define as informações básicas sobre um diagnóstico, incluindo ID, título, mensagem, severidade e categorias.

Diagnóstico: Uma mensagem que indica um possível problema no código.

ID do Diagnóstico: Identificado do diagnóstico. Pode ser usado para referenciar o diagnóstico em configurações, relatórios e código.

Título do Diagnóstico: Uma breve descrição do problema indicado pelo diagnóstico.

Mensagem do Diagnóstico: Uma descrição customizada para o diagnostico criado.

Severidade do Diagnóstico: Um nível de gravidade associado ao diagnóstico, como Error, Warning, Info ou Hidden.

Categorias do Diagnóstico: Rótulos opcionais, e customizáveis para agrupar os diagnósticos.

Ação Sugerida: Recomendação de mudança para corrigir ou lidar com o problema apontado pelo diagnóstico.

Escopo do Diagnóstico: O alcance do código fonte onde um diagnóstico é válido.

Supressão de Diagnóstico: A capacidade de optar por ignorar temporariamente um diagnóstico específico.