Bad Smell Finder C#: Uma ferramenta para encontrar *Bad Smells* em códigos C#

Higor Fischer de Paula Lopes[1]

[1]Instituto de Ciencias Exatas e Informâtica - PUC Minas
Rua Claudio Manoel, 1.162 - 30140-100 - Belo Horizonte, MG – Brasil

higorlopes@sga.pucminas.br

Abstract. *The technological evolution makes it essential to create methods and concepts for identifying poor-quality code. The concept of 'bad smell' is becoming increasingly relevant as an indicator, in certain code segments, of poor programming practices, which can cause issues and hinder project maintenance. One way to assist in detecting bad smells is through the use of supporting tools, allowing developers to direct efforts towards their resolution upon detection. Therefore, the goal of this work is the development of a tool for detecting bad smells in codes developed in the C# language, open source, where users can extend and create their own dashboards and configurations in a scalable and personalized manner for each application. In addition to tool development, a comparative analysis was conducted between the detections made by the developed tool and those made by Sonar. As a result, there was observed accuracy in identifying bad smells in C# similar to the Sonar tool, which is already known in the market. Both identified the same classes with the same bad smells for the same source code, with the proposed tool indicating more bad smells.*

Resumo. *A evolução tecnológica torna essencial a criação de métodos e conceitos para a identificação de códigos de má qualidade. O conceito de bad smell vem se tornando cada vez mais relevante por ser um indicador, em determinados trechos de código, de más práticas de programação, o que pode causar problemas e dificultar a manutenção do projeto. Uma forma de auxiliar a detecção de bad smells é através do uso de ferramentas de apoio, que ao realizar a detecção dos bad smells, os desenvolvedores podem direcionar esforços para a solução dos mesmos. Sendo assim, o objetivo do presente trabalho é o desenvolvimento de uma ferramenta de detecção de bad smells em códigos desenvolvidos em liguaguem C#, open source, em que usuários possam estender e criar seus próprios dashboards e configurações de forma escalável e personalizada para cada aplicação. Além do desenvolvimento da ferramenta, foi realizado uma análise comparativa entre as detecções realizadas pela ferramenta desenvolvida com as detecções realizas pelo Sonar. Como resultado, notou-se uma assertividade em encontrar bad smells em C# similar a ferramenta Sonar, que já é conhecida no mercado. Ambas identificaram as mesmas classes com os mesmos bad smells, para o mesmo código-fonte, com a ferramenta proposta apontando mais bad smells.*

Bacharelado em Ciência de Computação - PUC Minas
Trabalho de Conclusão de Curso (TCC)

Orientador: Cleiton Silva Tavares - cleitontavares@pucminas.br
Belo Horizonte, 15 de Dezembro de 2023.

1. Introdução

Cada vez mais, os programas de computadores vêm se tornando essencial para a sociedade. Com isso, torna-se relevante encontrar formas de criar e assegurar a saúde dos códigos-fonte da melhor maneira, considerando aspectos como a escalabilidade do sistema de *software*. Por causa da necessidade de evolução rápida que vem sendo exigida, a propensão à erros aumenta, o que pode causar algum comportamento ou estrutura indesejada. O conceito de *bad smells* vem se tornando cada vez mais relevante, tendo em vista que pode ser considerado um indicador de que um determinado trecho de código possa causar algum problema ou dificultar a manutenção do projeto [Fowler 1999]. Uma forma de resolver problemas de *bad smells* é através de refatoração. A refatoração pode ser definida como o processo de modificar um sistema de *software* para melhorar a estrutura interna do código sem alterar seu comportamento externo [Fowler 1999].

Sob este contexto, formas de refatorar código de maneira eficaz, rápida e sem muitos efeitos colaterais são os pontos mais desejados em uma evolução de *software*. Existem diferentes trabalhos que abordam o tema *bad smells*. Alguns destes trabalhos utilizaram análises de ferramentas, como o *software FaultBuster*, que tem o principal objetivo de verificar códigos de forma estática [Szőke et al. 2015]. Outros fizeram análises de códigos-fonte manualmente para encontrar a consequência da remoção de *bad smells* em um código [Almogahed et al. 2022]. Além desses, existem outras ferramentas que conseguem encontrar e mostrar ajustes para serem feitos em *bad smells*, como o *Sonar* [1]. Este *software* é capaz de realizar a detecção de códigos em diversas linguagens de programação. Um problema do *Sonar* é a dificuldade de adicionar novos *bad smells* de forma ágil. Outras ferramentas como *Designite*[2] [Sharma et al. 2016], que funciona tanto para Java como para C#, e o *JDeodorant*[3] [Tsantalis et al. 2008], que funciona somente para Java. Porém, nenhum trabalho apresentou uma ferramenta para a linguagem C# *Open Source* em que seja possível visualizar, de forma gráfica, a qualidade de um código em relação ao seus *bad smells*.

Sendo assim, o objetivo geral do presente trabalho é o desenvolvimento de uma ferramenta de detecção de *bad smells* em códigos desenvolvidos em liguaguem C#, *open source*, em que usuários possam estender e criar seus próprios *dashboards* e configurações de forma escalável e personalizada para cada aplicação. Tem-se como objetivos específicos: detectar os *bad smells Long Method*, *Long Parameter List*, e *Large Class*; vincular a análise gráfica aos dados encontrados para auxiliar o processo de entendimento da qualidade do código, de forma que o usuário consiga visualizar de maneira ágil qual o *bad smell* mais crítico em sua base de código; e realizar uma análise comparativa da ferramenta proposta e desenvolvida neste artigo com as detecções realizadas pelo *Sonar*.

[1] `https://www.sonarsource.com/products/sonarqube/`

[2] `https://marketplace.visualstudio.com/items?itemName=designite.Designite2019`

[3] `https://github.com/tsantalis/JDeodorant`

Como resultado deste trabalho, foi desenvolvida uma ferramenta que traz a possibilidade de identificar e tratar os *bad smells* configurados no formato de um painel gráfico, em que seja possível mostrar ao usuário o total de arquivos analisados, quantos arquivos estão com *bad smells* e quantos *bad smells* existem no código-fonte. Além disso, a ferramenta desenvolvida apresenta um indicador percentual da quantidade de arquivos que estão com pelo menos um *bad smell*. Além dessa visualização, é possível alterar os valores de cada um dos parâmetros de configuração para rastreabilidade de um *bad smell* e, dessa forma, conseguir adaptar a análise para o formato que melhor se enquadra para cada repositório de código. Das comparações feitas entre o *Sonar* e a ferramenta proposta, constatou-se que a quantidade de *bad smells* encontrada foi maior na ferramenta desenvolvida neste artigo.

Este artigo se estrutura da seguinte forma: na Seção 1 é apresentada a contextualização e introdução do trabalho. Na Seção 2, são apresentados os conceitos relevantes e a revisão bibliográfica utilizada para desenvolvimento. Na Seção 3, são contemplados os trabalhos relacionados ao objeto proposto neste artigo. Na Seção 4, são apresentadas as ferramentas e as metodologias utilizadas para a execução e desenvolvimento da ferramenta. A Seção 5 apresenta os resultados e comparação com o *software Sonar*. A Seção 6 apresenta as ameaças à validade. Por fim, a Seção 7 apresenta a conclusão do trabalho.

2. Referencial teórico

Esta seção tem como objetivo apresentar os conceitos relevantes para o desenvolvimento e entendimento do trabalho aqui descrito, bem como a relevância do auxílio desta ferramenta.

2.1. Bad Smells

Bad smells (em português, ”maus cheiros”) são indicadores de problemas ou deficiências no código de um programa [Fowler 1999]. São sintomas de más práticas de programação que podem levar à dificuldades na manutenção, compreensão e até mesmo em *bugs*. Os *bad smells* são um conceito da área de Engenharia de *Software*, popularizado pela metodologia de desenvolvimento ágil conhecida como *Extreme Programming (XP)*. Podem ser identificados através de uma análise do código, buscando por características indesejáveis que possam prejudicar sua qualidade e eficiência. Existem diferentes tipos de *bad smells*, cada um indicando um problema específico. Um exemplo é o *Long Method*, que indica quando um método possui muitas linhas e pode acabar indicando uma responsabilidade que talvez o método não precise ter. Alguns exemplos comuns incluem: Duplicação de código; Métodos ou classes muito extensos; Nomes de variáveis ou métodos confusos; entre outros [Kokol et al. 2021]. Na exemplo de código 1, é possível visualizar um exemplo do *bad smell Long Method*, já que o código utilizado como exemplo, que altera o capítulo selecionado de um livro para o mais atual, mostra uma alta quantidade de linhas, superando 50.

```
1 void ShowChapter (Chapter chapter, Tutorial tutorial)
2 {
3     if(chapter.orderIndex == 1)
4     {
5         this.redirectToRoot ()
6     }
7
8     var higherChapters = chapter.items.filter(item => item.higher);
9     var lowerTutorials = chapter.items.filter(item => !item.higher);
```

```
10
11      var title = chapter.title;
12      var description = chapter.description;
13
14      if (chapter.lowerItem)
15      {
16          nextPath = getChapterPath(tutorial, chapter.lowerItem);
17          nextLink = nextPath;
18      }
19      else if(!!tutorial.pathId && !!lowerTutorials)
20      {
21          nextPath = getTutorialPath(lowerTutorials[0])
22          nextLink = nextPath
23      }
24      else if (chapter.last && !tutorial.footLinks.length > 0)
25      {
26          nextPath = tutorial.footLinks.url
27      }
28
29      if (higherChapters.length == 0)
30      {
31          getChapterPath (tutorial, higherChapters);
32      }
33      else if (!!tutorial.pathId && tutorial.higherItem)
34      {
35          var higherTutorial = tutorial.higherItem;
36      }
37
38      lastChapter = higherTutorial.chapters [higherTutorial.chapters.length -1];
39
40      if (lastChapter.orderIndex == 1)
41      {
42          prevLink = getTutorialPath(higherTutorial)
43      }
44      else
45      {
46          prevLink = getChapterPath(higherTutorial, higherChapters)
47      }
48
49      this.storeProps = new {
50          checkableType: "Chapter",
51          checkableId: chapter.id,
52          checkboxes: signedIn ? currentUser.getCheckboxesFor(chapter) : [];
53      }
54
55      setStore();
56
57      modal = getModalChapter(chapter) || getModalTutorial(tutorial);
58  }
```

Exemplo de código 1. Método longo

3. Trabalhos relacionados

Os trabalhos relacionados discutidos nesta seção envolvem a refatoração de código e seus métodos para poder encontrar e facilitar a edição do código. São apresentados trabalhos encontrados na literatura atual que objetivam a melhoria da manutenabilidade e escalabilidade de um código.

O trabalho desenvolvido por Rizvi and Khanam (2011) mostra alguns pontos importantes como: diminuir o acoplamento dos códigos existentes; remover componentes reutilizáveis; definir prioridades para execução das alterações no código. Além disso, é discutido e analisado quando um determinado *bad smell* detectado é relevante de ser revisto, tendo em vista que, mesmo que seja código legado, ou seja, possui uma estruturação antiga, existe a possibilidade de que ainda esteja sendo executado corretamente o que foi programado. Com isso, é chegada a conclusão de que, ocasionalmente, não é possível fazer uma refatoração, uma vez que os recursos que serão utilizados podem acabar não compensando o trabalho e as horas que serão gastas ajustando os problemas encontrados. Por isso, a ferramenta desenvolvida neste trabalho complementa o trabalho descrito acima, facilitando a identificação de *bad smells*.

Zhang et al. (2011), seguem alguns caminhos para encontrar possíveis pontos de falhas no código. São apresentados cinco *bad smells* mostrados por [Fowler 1999], entre eles a duplicação de código. São utilizadas duas bases de código para realizar

comparações e pesquisas. No fim do artigo, é possível encontrar o melhor caminho para iniciar o processo de refatoração de código para que se resolvam os problemas, desde os mais simples, até os mais complexos. Com isso, a ferramenta proposta por esse artigo, permite que esses *bad smells* sejam encontrados de forma ágil para que assim sejam refatorados, correlacionando com o trabalho citado neste parágrafo.

Almogahed et al.(2022) mostram como a refatoração de código é um processo importante para garantir a segurança do código. É dito que não se têm muitos estudos com o tema de refatorações voltadas para a segurança após as mudanças. É mostrado todos os 68 tipos de refatoração de código citados por [Fowler 1999], e são escolhidos somente 5 tipos para a validação dos dados. Os parâmetros que indicam a segurança do código são testados em cima de 5 refatorações. Após serem feitas, a conclusão é de que os desenvolvedores não podem iniciar o processo de refatoração sem anteriormente analisarem os prós e contras do processo, tendo sob a ótica que isso trará um melhor entendimento do contexto do problema e poderá diminuir o custo de manutenção e outros problemas que possam ser causados. Dessa maneira, utilizar uma ferramenta, como a proposta por este artigo, diminui o tempo de identificação de *bad smells* e permite que o foco seja na refatoração visando a segurança.

Vedurada and Nandivada (2017) mostram dois tipos de principais refatorações de *bad smells*. O *Replace Type Code With Subclass (SC)* e o *Replace Type code with state (ST)*. Esses métodos se utilizam de polimorfismo, ao invés de condicionais, para manter o código mais organizado e legível. É dito que é bem desafiador encontrar manualmente lugares de refatoração em projetos grandes, e que várias abordagens existentes para poder encontrar refatorações *ST* falham por causa de restrições da abordagem. Com isso, é apresentado o desafio: identificação dos locais de refatoração, pois existem muitas variações e formatos de *bad smells* que podem ser encontrados. Sabendo disso, a ferramenta desenvolvida no presente artigo apresenta a escalabilidade e estruturação necessária para que, ao encontrar alguma variação de formato de *bad smell*, o próprio usuário tenha a possibilidade de inserir uma nova forma de detecção.

Sz oke et al. (2015) apresentam que, conforme o código vai sendo desenvolvido e aumentando sua escala, torna-se difícil manter a qualidade e uma refatoração contínua. Isso faz com que a refatoração não seja sempre tão simples, pois é necessário uma análise para verificacção dos possíveis efeitos colaterais. Por isso, é apresentado nesse ponto o *FaultBuster*. Esta ferramenta verifica o código de forma estática e assim permite encontrar possíveis problemas no código. Como conclusão, é mostrado que esse produto tornou possível refatorar mais de 5 milhões de linha de códigos em diferentes empresas e para mais de 40 problemas diferentes. Foram resolvidos 11.000 problemas de códigos. Outras ferramentas similares que são citadas para outras para encontrar *bad smells* são o *ReSharper* e o *CodeRusher* para o .NET. Dessa maneira, é possível entender o impacto que uma ferramenta que auxilia na construção de código sem *bad smells* otimiza o tempo de desenvolvimento de um *software*.

Fernandes (2022), no contexto de ferramentas de refatoração, apresenta e comenta sobre o conceito de *Live Environment*. Nesse estudo, é pesquisado qual a melhor forma de mostrar ao desenvolvedor *bad smells* em seu código. São pensadas formas de: mostrar ao desenvolvedor, de uma forma não intrusiva, de possíveis *bad smells*; quais seriam os impactos de uma detecção em tempo real na noção de qualidade de código; qual o impacto

no código-fonte; e o impacto do tempo necessário para poder chegar em uma boa solução de programação. Para realização dos testes, foi desenvolvida uma ferramenta para a *IDE* IntelliJ da linguagem Java. Dessa forma, foi possível mostrar ao usuário de forma visual onde seria necessário intervenção em seu código. Sendo assim, a ferramenta desenvolvida neste artigo, tem um objetivo similar ao do trabalho apresentado neste parágrafo, focando em códigos da linguagem C#.

Ivers et al. (2022) abordam o *refactoring* em indústrias e em ampla escala, que envolve mudanças significativas no sistema. É apresentado que, atualmente, os desenvolvedores enfrentam desafios na realização do *refactoring* em escala ampla e utilizam diversas ferramentas para apoiar as atividades envolvidas. A pesquisa revelou que o *refactoring* é comum e motivado por preocupações empresariais mais amplas. A falta de ferramentas adequadas e o custo são razões comuns para não realizá-lo. No entanto, a indisponibilidade destas ferramentas resulta na incapacidade de entregar novos recursos. Melhores instrumentos podem ajudar a mudar decisões comerciais e reduzir a quantidade de esforços envolvidos em escala ampla. Com isso, pode-se notar a relevância de ferramentas como a desenvolvida por este trabalho.

Fernandes et al. (2022), apresentam, neste outro artigo, a abordagem do *Live Refactoring*, que visa melhorar a experiência de refatoração de *software*. É um ambiente de desenvolvimento que oferece *feedback* em tempo real aos desenvolvedores, identificando possíveis problemas no código, como *bad smells*, e sugerindo refatorações adequadas. Através da aplicação de métricas de qualidade de código, o ambiente detecta os *bad smells* e recomenda as refatorações correspondentes. Além disso, são utilizadas técnicas de visualização para auxiliar na identificação dos candidatos à refatoração. O objetivo é criar um código mais limpo, compreensível e adaptável desde estágios iniciais de desenvolvimento, reduzindo a necessidade de grandes refatorações tardias. Conclui-se que o quanto antes a detecção dos problemas for realizada, melhor será a evolução e qualidade do código. Um ponto de destaque do trabalho desenvolvido e apresentado neste artigo é a identificação ágil de *bad smells*.

Khanzadeh et al. (2023) apresentam o *Opti Code Pro*, uma ferramenta que utiliza algoritmos de busca *best-first* para automatizar e simplificar o processo de refatoração de código em sistemas orientados a objetos. O objetivo da ferramenta é minimizar o acoplamento e maximizar a coesão do programa, alterando as dependências entre classes e módulos. O artigo descreve o problema de refatoração de código como um problema de busca, define os estados e ações possíveis e estabelece os critérios de estado objetivo. Embora algumas simplificações tenham sido feitas para melhorar o desempenho da busca, os resultados mostram que o *Opti Code Pro* é capaz de encontrar soluções eficazes que melhoram a coesão e o acoplamento do código, premissas propostas como objetivo do estudo. Pode-se concluir que a utlização de uma ferramenta, como a desenvolvida neste artigo, torna-se indispensável para uma boa evolução e escrita de código.

4. Metodologia

Essa seção tem como objetivo descrever a metodologia utilizada para o desenvolvimento da ferramenta de detecção de *bad smells* em código C# apresentado neste artigo.

A refatoração de código se faz necessária para que sistemas consigam evoluir e se manter escaláveis e manuteníveis. Com isso, este trabalho tem o foco no desenvolvimento

de uma ferramenta *open source* [Aberdour 2007] de detecção de *bad smells* em códigos desenvolvidos com a linguagem C#. A ferramenta proposta permite a novos interessados estender funções já existentes, criar novas classes de validação e, assim, conseguir evoluir continuamente o *software*, garantindo a qualidade de seu produto final. Assim, torna-se possível montar estratégias específicas para determinadas formas de codificação e assim, permitindo o usuário determinar qual o paradigma e métodos utilizar para sua aplicação particular.

4.1. Ambiente

O ambiente que será utilizado para o desenvolvimento é o sistema operacional *Windows 10*. Sua escolha deve-se ao fato de que a linguagem abordada neste artigo, por ser desenvolvida e mantida pela *Microsoft*, possui uma base de suporte técnico apropriado e uma ampla variedade de ferramentas que tornam o desenvolvimento do código acessível e compreensível.

4.2. Métodos utilizados

O desenvolvimento terá como início o estudo e definição da melhor maneira de encontrar os *bad smells* dentro do código. Este estudo introdutório avaliou *tresholds* de configuração para avaliar quais limites devem ser determinados para serem analisados e tratados em cima do conceito de *bad smells*. Os primeiros *bad smells* que serão utilizados são os que fazem enumeração de váriaveis, como por exemplo o que conta a quantidade de parâmetros de um método e o que conta quantidade de linhas de um método. Esta decisão foi tomada para que o foco seja na estruturação e na montagem da base de código da ferramenta, sendo a maior preocupação a escalabilidade e extensibilidade.

A utilização da árvore sintática da linguagem C# é a maoir razão que torna o desenvolvimento mais rápido e escalável. A própria *Microsoft* disponibiliza uma biblioteca denominada *Roslyn* [4]. Esta biblioteca gera a árvore sintática e, assim, entrega a identificação de métodos, variáveis e propriedades. As informações que são disponibilizadas pelo retorno do *Roslyn* no código analisado facilitam a escalabilidade do código e a adaptação desta ferramenta proposta para um uso específico, objetivando a identificação de *bad smells*. No exemplo de código 2, é possível visualizar a implementação de uma classe de exemplo, denominada *Sample*, que tem como função a obtenção da árvore sintática e, desta forma, adquirir informações para identificações dos *bad smells*.

```
1 public class Sample
2 {
3     public string FooProperty { get; set; }
4     public void FooMethod(string foo, string bar ){ }
5 }
```

Exemplo de código 2. Classe para *input* da análise sintática.

4.3. Código-fonte

A base de código, desenvolvida em C#, utilizada neste trabalho é de propriedade privada de uma empresa e não pode ser disponibilizada por se tratar de propriedade intelectual. O autor do artigo é membro da empresa e tem acesso para realização de testes

[4] https://github.com/dotnet/roslyn

diretos. O sistema utilizado como teste é da área de *supply chain*, possui mais de 1800 arquivos C#, conta com mais de 61 mil linhas de código e apresenta mais de 1000 classes dentre elas, algumas possuindo *bad smells* como *Long Methods*, *Large Class* e *Long Parameter List*.

4.4. Detecção de Smells

Para iniciar o processo de detecção de *bad smells*, utilizou-se o código descrito no item 4.3. A Figura 1 apresenta como é possível obter informações sobre a propriedade encontrada dentro do código de exemplo utilizado para o desenvolvimento deste trabalho. São visualizados: o nome da propriedade, sua visibilidade, em qual linha se encontra, dentre outras informações. Ja na Figura 2 é possível observar, as mesmas informações, só que de um método analisado.

Name	Value
property	PropertyDeclarationSyntax PropertyDeclaration public string FooProperty {get; set;}
AccessorList	AccessorListSyntax AccessorList {get; set;}
AttributeLists	{} View
ContainsAnnotations	false
ContainsDiagnostics	false
ContainsDirectives	false
ContainsSkippedText	false
ExplicitInterfaceSpecifier	null
ExpressionBody	null

Figura 1. Informações da análise sintática de uma propriedade.

Name	Value
method	MethodDeclarationSyntax MethodDeclaration public void FooMethod(string test, ...
Arity	0
AttributeLists	{} View
Body	BlockSyntax Block { }
ConstraintClauses	{} View
ContainsAnnotations	false
ContainsDiagnostics	false
ContainsDirectives	false
ContainsSkippedText	false
ExplicitInterfaceSpecifier	null
ExpressionBody	null
FullSpan	{[69..125)}
HasLeadingTrivia	true
HasStructuredTrivia	false
HasTrailingTrivia	true
Identifier	SyntaxToken IdentifierToken FooMethod
IsMissing	false
IsStructuredTrivia	false
KindText	"MethodDeclaration" View
Language	"C#" View
Modifiers	{public} View
ParameterList	ParameterListSyntax ParameterList (string test, string test2)
Parent (Microsoft.CodeAnaly...	ClassDeclarationSyntax ClassDeclaration public class Sample{ public string FooPro...
ParentTrivia	SyntaxTrivia None
RawKind	8875
ReturnType	PredefinedTypeSyntax PredefinedType void
SemicolonToken	SyntaxToken None
Span	{[71..123)}
SpanStart	71
SyntaxTree (Microsoft.CodeA...	{public class Sample{ public string FooProperty {get; set;}public void FooMethod(...
SyntaxTreeCore	{public class Sample{ public string FooProperty {get; set;}public void FooMethod(...
TypeParameterList	null
Non-Public members	

Figura 2. Informações da análise sintática de um método.

Para a validação da solução proposta, será feita a comparação da quantidade de *bad smells* que são encontrados dentro do código apresentado no item 4.3. Após essa etapa, o código-fonte será compilado e processado. Posteriormente, será feita a análise e

verificação dos erros. Também é apresentado, no item 4.5 uma comparação qualitativa e quantitativa da ferramenta de detecção de *bad smells* desenvolvida com outra ferramenta existente. O resultado da análise proposta é apresentado juntamente aos resultados descritos no item 5.

4.5. Análise comparativa

Para a realização da análise comparativa, a fim de testar a acurácia da ferramenta proposta e desenvolvida, será utilizado o *software Sonar* que irá analisar o código, apresentado no item 4.3, e retornará os *bad smells*. Em seguida, os indicadores retornados pelo *Sonar* serão comparados com os indicadores retornados pela ferramenta proposta. A análise é apresentada no item 5.3. O *Sonar* possui detecção de mais de 400 *bad smells* em C# e consegue identificar em qual arquivo e em qual linha se encontram. Existem outras ferramentas como o *Designite*, *JDeodorant*, porém, para o desenvolvimento analisado, não é possível utilizá-las, uma vez que são focadas em identificar *bad smells* na linguagem Java.

5. Resultados

Nesta seção, são apresentados os resultados obtidos com a ferramenta desenvolvida para detecção de *bad smells* em códigos desenvolvidos utilizando a linguagem C#.

5.1. Classes e Configurações

Para garantir a refatoração e escalabidade do código proposto, desenvolveu-se uma arquitetura para este projeto baseada em DDD *(Domain Driven Design)*. Com isso, considerando o contexto da aplicação, tem-se as principais classes do projeto. Uma das classes desenvolvidas é a *CodeAnalysis*, apresentada no exemplo de código 3. É possível visualizar as características do arquivo que se deseja analisar, tais quais: qual a classe dentro do arquivo, quais métodos essa classe possui, quais propriedades e outras informações referentes a classe. Dentro dessa classe, encontram-se subclasses mais específicas para poder isolar, ou seja, separar por contextos, as informações de cada entidade e domínio.

```
1  public class CodeAnalysis
2  {
3          public string Name { get; init; } = "";
4          public int Lines { get; init; }
5
6          public IList<CodePropertyInfo> Properties { get; } = new List<CodePropertyInfo>();
7          public IList<CodeMethodInfo> Methods { get; } = new List<CodeMethodInfo>();
8          public IList<CodeClassInfo> Classes { get; } = new List<CodeClassInfo>();
9          public IList<CodeConstructorInfo> Constructors { get; } = new List<CodeConstructorInfo>();
10
11         public IList<BadSmell> BadSmells { get; } = new List<BadSmell>();
12 }
```

Exemplo de código 3. Classe CodeAnalysis.

Devido ao fato de os *bad smells* serem características de códigos, esses valores podem ser subjetivos para cada projeto e para cada linguagem. Portanto, um dos pontos principais é possibilitar a configuração e a utilização de forma rápida de valores para a análise. Dessa forma, foi criada a classe *CodeConfig*, apresentada no exemplo de código 4. Assim, cada usuário consegue definir seus próprios limites para um código limpo, legível e de forma fácil para ser escalável e sofrer manutenções.

```
1  public class CodeConfig
2  {
3          public string Name { get; set; }
4          public long LongMethod { get; set; }
5          public long LongParametersList { get; set; }
```

```
        public long TooManyMethods { get; set; }
        public long TooManyProperties { get; set; }
        public long LargeClass { get; set; }
}
```

Exemplo de código 4. Classe CodeConfig.

Além das classes citadas, foram criadas classes que armazenam dados específicos sobre as classes, como os métodos. Na classe *CodeMethodInfo*, apresentada no exemplo de código 5, são mantidas informações principais de um método, como: número de linhas; linha em que o método está localizado; quantidade de parâmetros dentro deste método e outras possíveis informações que forem adicionadas. Dessa maneira, é possível desacoplar completamente a forma de encontrar as informações do código da ferramenta *Roslyn*. O principal benefício desse desacoplamento é a escalabilidade, pois caso seja necessário alterar a forma de análise, é possível inserir as novas informações de outra maneira. As informações de propriedades, construtores e afins seguem a mesma estrutura, o que facilita a utilização e escalabiliade.

```
public class CodeMethodInfo
{
        public string Name { get; set; }

        public int Line { get; set; }
        public int Lines { get; set; }
        public int Parameters { get; set; }
}
```

Exemplo de código 5. Classe CodeMethodInfo.

Além de se utilizar *thresholds* dos estudos na literatura, foram criados métodos matemáticos para analisar e entregar valores baseados na base de código. Foram feitas duas análises: a de média e a de interquartil. Ambas entregam os valores de cada váriavel a ser analisada dentro da ferramenta, como apontado na Figura 3. Dessa forma, cabe ao usuário entender e analisar se os valores analisados encaixam-se em sua estrátegia.

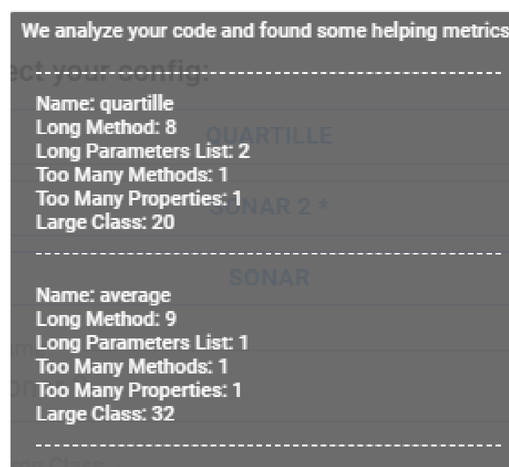

Figura 3. Dicas de valores baseados nas métricas do código.

Foi desenvolvido uma classe de totalizador para ser possível encontrar informações gerais sobre a base de código utilizada, apresentada no exemplo de código 6. A classe possui informações como: o total de arquivos verificados, total de *bad smells* e outras possíveis métricas que possam vir a fazer sentido de adicionar para evoluir a análise. Como apresentado anteriormente, as definições de quais *threshold* serão utilizados são subjetivas, o que faz com que o usuário tenha a possibilidade analisar múltiplas

configurações para entender o comportamento da base. Dessa forma, a ferramenta apresenta uma seção, visto na Figura 4, em é possível mostrar múltiplas configurações e somente executar o que deseja ver no instante da análise.

```
1 public class ProjectAnalysis
2 {
3         public int TotalVerifiedFiles { get; private set; }
4         public int FilesWithBadSmells { get; private set; }
5         public int TotalBadSmells { get; private set; }
6         public int TotalLines { get; private set; }
7         public int TotalClasses { get; private set; }
8         public int TotalMethods { get; private set; }
9 }
```

Exemplo de código 6. Classe ProjectAnalysis.

Select your config:
QUARTILLE
SONAR 2 *
SONAR
Name *
Sonar 2
Large Class
200
Long Method
80
Too Many Properties
5
Too Many Methods
9
Long Parameter List
7
ADD NEW

Figura 4. Seção de configuração da ferramenta.

Como resultado da interface gráfica desenvolvida, a ferramenta apresenta, Figura 5, um *Front end* que está dividido em duas seções. A primeira seção é a parte analítica, onde obtem-se dados gerais, como a quantidade de arquivos analisados, total de *bad smells*, quantidade de arquivos com erros. Os gráficos de barras apresentam o totalizador dos *bad smells* encontrados. No canto inferior direito, no gráfico de pizza, é possível visualizar, em dados percentuais, quais foram os *bad smells* encontrados, quantos e qual a proporção total de cada um em razão do total de *bad smells* encontrados.

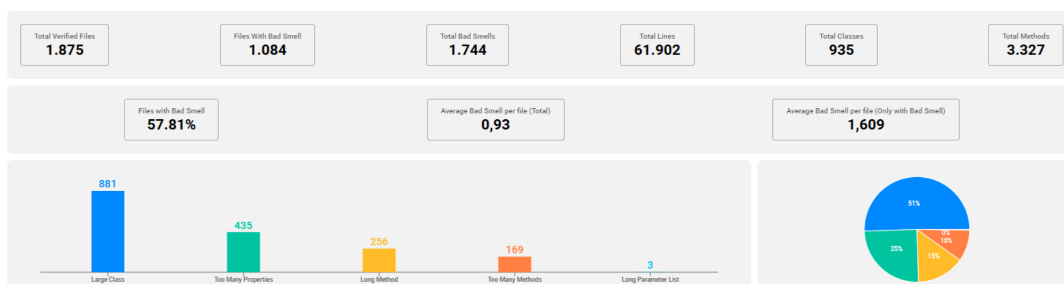

Figura 5. Dados dos *bad smells* de uma base de código

Na Figura 6 é apresentada uma lista dos *bad smells* e consequentemente qual arquivo se enquadra em cada problema apresentado. Dessa forma é possível atuar diretamente no arquivo e no problema.

Figura 6. Listagem dos *bad smells*

5.2. Análise comparativa de auxílios disponíveis entre as ferramentas

Para a análise comparativa, tomou-se como base configurações já definidas e utilizadas por *softwares* conhecidos. Sabendo disso, foram pesquisadas múltiplas ferramentas como *Designite*, *JDeodorant* e *Sonar*. Essas ferramentas foram escolhidas por possuirem alguns dos parâmetros que foram implementados na ferramentas. Porém, tanto a *Designite* como a *JDeodorant* são ferramentas que identificam somente *bad smells* em Java. Contudo, foi estudada a possibilidade de utilizar estes *thresholds* para validar códigos C#.

Na Tabela 1, é possível verificar quais funcionalidades a ferramenta desenvolvida apresenta em comparação as outras ferramentas que foram citadas no parágrafo anterior. Observa-se, no primeiro momento, que as outras ferramentas apresentam uma capacidade de encontrar mais tipos de *bad smells*. Entretanto, é necessário ressaltar que a arquitetura desenvolvida neste trabalho permite a escabalabilidade do código, ou seja, existe a possibilidade de inserir mais tipos de *bad smells* para serem identificados, ponto destacado na linha 5 da Tabela 1. A fácil evolução e a escalabilidade da ferramenta, permite com que o próprio usuário, conforme a necessidade, consiga adicionar novos *bad smells* de forma simples e direta.

	JDeodorant	Designite	SonarCube	Bad Smell Finder C#
Base de *bad smells*	5	11	435	5
Indicação de resoluções	15	18	Sim	Não
Linguagem de detecção	Java	C# e Java	C# e outras	C#
Adição de detecção de *bad smells*	Não	Não	Não	Sim
Alteração de parâmteros	Não encontrado	Sim	Sim	Sim
Open source	Sim	Sim	Não	Sim

Tabela 1. Comparação de auxílios disponíveis entre ferramentas.

5.3. Validações e análises comparativas entre a Bad Smell Finder C# e o *Sonar*

O principal ponto de destaque da ferramenta proposta sobre as concorrentes é o fato de ser focado em C#, uma vez que somente o *Sonar* possui essa capacidade. Outro ponto de destaque que faz a ferramenta sobressair é ser *open source*. Entretanto, alguns pontos desfavoráveis da ferramenta desenvolvida foram analisados. Um deles é a falta de solução para o problema apontado, ou seja, após a identificação de um *bad smell*, não são indicadas melhorias para resolução do problema. Além disso, até o momento de desenvolvimento deste artigo, o código base da ferramenta encontra-se em evolução, possuindo um caráter demonstrativo e demonstrando sua robustez.

Após o desenvolvimento da estrutura e da forma de visualização, pela interface gráfica desenvolvida, realizou-se a validação da ferramenta comparado com uma ferramenta já existente. A escolhida foi o *Sonar*, já que possui análise de códigos C#. A primeira etapa da validação foi a escolha da base de código, apresentada no item 4.3. O próximo passo foi executar o *Sonar* e o *Bad Smell Finder C#* em cima dessa mesma base de código, para, dessa forma, ter uma base de comparação e verificação da acurácia da ferramenta desenvolvida.

Após a execução de ambas as ferramentas, os resultados foram organizados de forma a se obter a Tabela 2. É possível verificar que, a ferramenta proposta identificou um maior número de *bad smells* do que o *Sonar*. Ao focar somente um *bad smells*, o *Long Method*, é analisada a diferença entre as ferramentas. Como visto na Figura 7, foram encontrados 6 arquivos através do *Sonar*. Ao analisar a ferramenta desenvolvida, Figura 8, verifica-se que os mesmos arquivos encontrados pelo *Sonar* foram encontrados pela ferramenta. Porém, a ferramenta desenvolvida encontrou outros métodos a mais, tendo sob entendimento que a forma de encontrar *bad smells* de ambas variam. Deve-se destacar que ambos utilizaram a mesma configuração de *treshold* para realização das análises comparativas.

	Configuração	Sonar C#	Bad Smell Finder C#
Large Class	200 linhas	16	36
Long Parameter List	7 parâmetros	15	18
Long Method	80 linhas	6	15
Too Many Methods	9 métodos	-	78
Too Many Parameters	5 parâmetros	-	169

Tabela 2. Comparação de *bad smells* encontrados.

Figura 7. Listagem de *bad smells* no Sonar.

Figura 8. Listagem de *bad smells* na ferramenta proposta.

6. Ameaças à validade

Nesta seção são discutidas ameaças a validade [Wohlin et al. 2012].

O artigo propõe uma ferramenta para a identificação de *bad smells* em C#. Ao utilizar técnicas de análise estátistica, a ferramenta demonstra eficácia na detecção de práticas de programação prejudiciais, contribuindo para a melhoria da qualidade do código. A implementação cuidadosa de algoritmos e a abordagem sistemática na definição de *bad smells* fundamentam a confiabilidade da ferramenta.

No entanto, existem desafios a serem considerados. A validade interna é suscetível a viéses de implementação e possíveis erros no código. A generalização para diferentes contextos e a evolução da linguagem C# representam desafios na validade externa. A definição dos *bad smells* e as métricas utilizadas para avaliação são considerações cruciais para a validade de construção.

7. Conclusão

O desenvolvimento da ferramenta de detecção de *bad smells* para a linguagem C# representa um avanço significativo no contexto da manutenção e evolução de *software*. Ao longo deste trabalho, foram abordados aspectos cruciais relacionados à qualidade do código e à identificação de práticas de programação inadequadas. A análise de *bad smells* é essencial para identificar potenciais problemas que podem prejudicar a compreensão, a manutenção e a evolução do *software*. No entanto, a falta de ferramentas *open source* específicas para a linguagem C# motivou a criação desta ferramenta, preenchendo uma lacuna importante no ecossistema de desenvolvimento.

Sendo assim, a ferramenta construída permite a detecção dos *bad smells Long Method*, *Long Class*, *Long Parameter List* e *Too Many Properties*. Além disso, também proporciona uma visão clara das métricas e *bad smells* encontrados nos arquivos analisados. A interface gráfica desenvolvida, permitiu a criação de gráficos que oferecem uma representação visual da qualidade do código, permitindo uma avaliação rápida e intuitiva. Os resultados obtidos na fase de testes, e apresentados no item 5, demonstraram a eficácia da ferramenta na identificação de *bad smells* comparando com outras ferramentas já difundidas como o *Sonar*.

Para trabalhos futuros, é possível citar alguns desenvolvimentos que visam eliminar os pontos desafavoráveis, citados ao longo do artigo, da ferramenta aqui desenvolvida e apresentada. Pode-se garantir a implementaçao de detecção de novos *bad smells*, aumentando assim o repositório de identificações do *software* e garantindo a escalabidade do sistema, ponto de extrema relevância e de motivação para o desenvolvimento da ferramenta. Além disso, planeja-se adicionar formas de entrega contínua, que é uma análise em tempo real do código que está sendo desenvolvido. Dessa forma o *feedback* é constante e ágil. Pretende-se também evoluir o *software* de forma a garantir que, ao identificar um *bad smell*, o usuário obtenha um retorno de um método de resolução e eliminação do problema. Atualmente, a ferramenta desenvolvida não possui a característica citada acima, o que a coloca em desvantagem quando comparada aos outros *softwares* disponíveis.

Pacote de Replicação

O pacote de replicação deste trabalho encontra-se disponível em: `https://github.com/Orientacoes-TCC/puc-tcc-pl-cc-2023-1-higorFischer`

Referências

Aberdour, M. (2007). Achieving quality in open-source software. *IEEE Software*, 24(1):58–64.

Almogahed, A., Omar, M., and Zakaria, N. H. (2022). Refactoring codes to improve software security requirements. *Procedia Computer Science*, 204:108–115.

Fernandes, S. (2022). Towards a live environment for code refactoring. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–5.

Fernandes, S., Aguiar, A., and Restivo, A. (2022). A live environment to improve the refactoring experience. In *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming*, pages 30–37.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

Ivers, J., Nord, R. L., Ozkaya, I., Seifried, C., Timperley, C. S., and Kessentini, M. (2022). Industry's cry for tools that support large-scale refactoring. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 163–164.

Khanzadeh, S., Chan, S. A. N., Valenzano, R., and Alalfi, M. (2023). Opti code pro: A heuristic search-based approach to code refactoring. *arXiv preprint arXiv:2305.07594*.

Kokol, P., Kokol, M., and Zagoranski, S. (2021). Code smells: A synthetic narrative review.

Rizvi, S. and Khanam, Z. (2011). A methodology for refactoring legacy code. In *2011 3rd International Conference on Electronics Computer Technology*, volume 6, pages 198–200. IEEE.

Sharma, T., Mishra, P., and Tiwari, R. (2016). Designite - a software design quality assessment tool. In *2016 IEEE/ACM 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities (BRIDGE)*, pages 1–4.

Szőke, G., Nagy, C., Fülöp, L. J., Ferenc, R., and Gyimóthy, T. (2015). Faultbuster: An automatic code smell refactoring toolset. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 253–258. IEEE.

Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. (2008). Jdeodorant: Identification and removal of type-checking bad smells. In *2008 12th European Conference on Software Maintenance and Reengineering*, pages 329–331.

Vedurada, J. and Nandivada, V. K. (2017). Refactoring opportunities for replacing type code with state and subclass. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 305–307. IEEE.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

Zhang, M., Baddoo, N., Wernick, P., and Hall, T. (2011). Prioritising refactoring using code bad smells. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 458–464. IEEE.