Análise e Identificação de Code Smells em Projetos Django

Eduardo Luiz Silva[1], **Orientador: Leopoldo Motta Teixeira**[1]

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Recife – PE – Brasil

[els6, lmt]@cin.ufpe.br

Abstract. *Python is a programming language known for its simplicity and readability, widely used across various domains, from web development to machine learning. In the context of web development, Django, a powerful Python-based framework, stands out for offering comprehensive tools and libraries. Despite its advantages, Django projects can exhibit poor coding practices, known as code smells, which indicate potential issues. This work proposes a study of the most common code smells in Django projects, aiming to develop a tool capable of identifying them and helping to avoid them, contributing to the improvement of code quality.*

Resumo. *Python é uma linguagem de programação conhecida por sua simplicidade e legibilidade, sendo amplamente utilizada em diversos domínios, desde desenvolvimento web até aprendizagem de máquina. No contexto do desenvolvimento web, o Django, uma poderosa estrutura baseada em Python, destaca-se por oferecer ferramentas e bibliotecas abrangentes. Apesar de suas vantagens, projetos Django podem apresentar más práticas de código, conhecidas como* code smells*, que indicam possíveis problemas. Este trabalho propõe um estudo dos* code smells *mais comuns em projetos Django, visando desenvolver uma ferramenta capaz de identificá-los e ajudar a evitá-los, contribuindo para a melhoria da qualidade do código.*

1. Introdução

Python é uma linguagem de programação de alto nível conhecida por sua simplicidade e legibilidade, tornando-se uma escolha popular entre desenvolvedores de software [Python Software Foundation b]. Sua sintaxe simples e versatilidade permitem que seja utilizada em uma variedade de domínios, desde desenvolvimento web até aprendizagem de máquina.

No contexto de desenvolvimento web, o Django é uma poderosa estrutura baseada em Python [Django Software Foundation 2024b]. Desenvolvido para facilitar a criação rápida e eficiente de aplicações web, o Django oferece um conjunto abrangente de ferramentas e bibliotecas. Sua arquitetura segue o padrão Model-View-Controller (MVC), permitindo a separação clara entre a lógica de negócios, a apresentação e o acesso aos dados. Além disso, o Django promove a reutilização de código por meio de aplicativos modulares e integrações simplificadas com banco de dados, tornando-o uma escolha popular para desenvolvedores em seus projetos web.

Como em todos os cenários de desenvolvimento, em Django também podem ocorrer más práticas de código, também conhecidas como *code smells*. Os *code smells* são indicadores de possíveis problemas no código que podem levar a problemas de manutenção e performance, dificultando a compreensão do código. No livro "Refactoring: Improving the Design of Existing Code", Martin Fowler conceituou e apresentou alguns *code smells* [Fowler 1999] e desde então vários outros estudos se aprofundaram no tema apresentando novos cenários e soluções.

Os *code smells* costumam a acontecer com mais recorrência entre desenvolvedores iniciantes e itermediários, em vista que não possuem domínio completo sobre as melhores práticas para a tecnologia que está sendo usada. Por ser um framework robusto, de arquitetura complexa e com uma grande riqueza de funcionalidades oferecidas, o Django possui uma curva de aprendizado acentuada em comparação a outros frameworks [Mariana Clark 2024]. Dessa forma, desenvolvedores que estão iniciando ou ainda não alcaçaram conhecimento mais abrangente sobre o framework podem facilmente adicionar *code smells* aos seus projetos por não terem, ainda, um entendimento completo. Com isso, a revisão e análise de código em Django se faz importante para mitigar a ocorrência de *code smells*.

Portanto, este trabalho tem o objetivo de fazer um estudo a respeito dos *code smells* mais comuns em projetos Django. A partir desta análise será possível identificar quais são os *code smells* mais relevantes e também verificar a possibilidade de desenvolver uma ferramenta de análise estática de código capaz de identificá-los além de verificar como eles podem ser evitados. Desta forma espera-se contribuir com uma nova abordagem na melhoria de código para trabalhos em Django que possa trazer mais facilidade no tratamento de *code smells*.

2. Fundamentação Teórica

Nesta seção serão apresentados os principais conceitos utilizados para o desenvolvimento deste projeto: Code Smells e o framework Django.

2.1. Code Smells

No livro "Refactoring: Improving the Design of Existing Code" [Fowler 1999], em parceria com Martin Fowler, Kent Beck instituiu o termo *Code Smells* para se referir a padrões no código-fonte que indicavam possíveis problemas ou defeitos. Tais "maus cheiros" não impedem o funcionamento do programa mas sinalizam pontos que podem deteriorar a qualidade do código e deixá-lo difícil de manter e escalar. No mesmo livro, Martin Fowler destacou a importância de detectar esses *code smells* porque geralmente eles indicam áreas onde o código precisa de uma refatoração, sendo um elemento importante no processo de melhoria do código para garantir sua sustentabilidade.

Ao longo dos anos os estudos a respeito dos *code smells* aumentaram, em [Kokol et al. 2020] foi feita uma análise sobre *code smells* e a produção literaria relacionada ao tema, sendo possível confirmar o crescimento das pesquisas com destaque para Estados Unidos, Brasil e Itália. Nesse estudo também se pôde levantar em quais pontos as pesquisas precisam ser mais abrangentes, além de reforçar o quanto identificar e corrigir esses "maus cheiros" é essencial para manter a qualidade do código em alto nível, para assim facilitar a manutenção e eficência do processo de desenvolvimento de software.

Porém, reconhecer esses sinais de problemas pode ser desafiador, ainda mais para desenvolvedores com pouca experiência. Ferramentas de análise de código podem ser de grande utilidade nesse processo para destacar áreas do código que possam precisar de atenção. O aumento da educação e concientização sobre as boas práticas de programação e padrões de desenvolvimento também possuem um papel importante na prevenção de *code smells*, para reforçar a importância de um código-fonte de qualidade desde o início do processo de desenvolvimento.

2.2. Django Framework

No desenvolvimento web é comum a utilização de frameworks para facilitar o desenvolvimento, uma vez que oferecem uma grande quantidade de ferramentas que aceleram e dão qualidade aos projetos. Django é o principal framework web para o desenvolvimento em Python e, em linhas gerais, segue o padrão de arquitetura MVC (Model-Controller-View). Segundo a documentação do framework, seu padrão de arquitetura é chamado de MTV (Model-Template-View) [Django Software Foundation], sendo uma variação do padrão MVC.

Na arquitetura MVC, temos a camada Model que é responsável pela lógica de negócios e pelo acesso aos dados da aplicação. A camada View é responsável pela apresentação dos dados do modelo ao usuário. Por fim a camada Controller faz o papel de intermediário entre as camadas Model e View, controlando as entradas do usuário e escolhendo qual a view será exibida.

No caso do Django, a camada Template tem o papel equivalente ao da camada View da arquitetura MVC, ela é responsável por como as informações serão exibidas para o usuário. Enquanto a camada View tem um papel semelhante ao da camada Controller da arquitetura MVC. A camada View recebe as requisições do usuário, se comunica com a camada Model e retorna respostas ao usuário normalmente utilizando um template. A Figura 1 dá uma visão de como é a organização da arquitetura MTV (Model-Template-View).

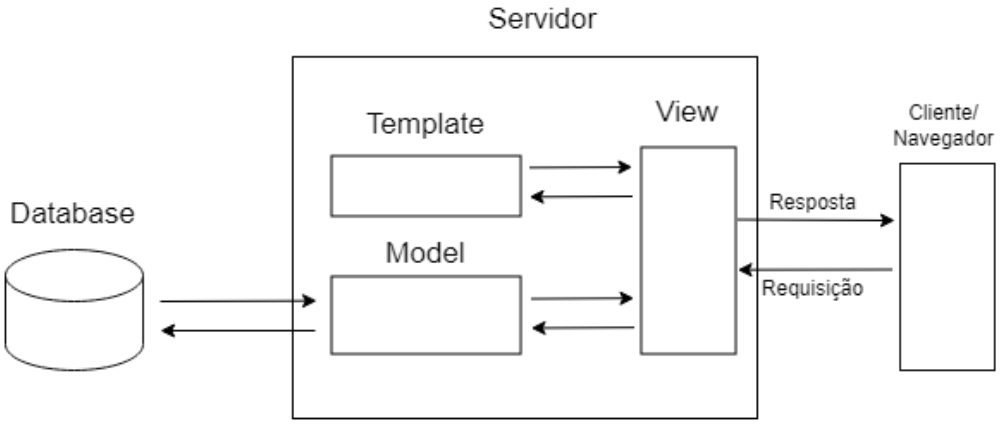

Figura 1. Arquitetura MTV (Model-Template-View)

Django possui uma variedade de recursos prontos para uso, como sistema automatizado de administração, serialização de dados, suporte para validação, entre outros,

que reduzem consideravelmente o tempo necessário para desenvolver e facilitam a manutenção [Python Geeks 2024]. Além disso, a capacidade de dimensionamento para lidar com tráfego intenso sem comprometer a segurança ou desempenho torna o Django uma escolha sólida para o desenvolvimento de aplicativos web, adequado tanto para projetos menores quanto para soluções em grande escala.

3. Metodologia

A metodologia deste trabalho teve o foco em duas etapas principais: investigar e analisar os problemas de *code smells* específicos ou mais recorrentes no desenvolvimento de aplicações Django. A primeira etapa foi a realização de uma revisão cinzenta sobre code smells associados ao Django, uma vez que nas pesquisas iniciais não foram encontrados materiais acadêmicos abordando especificamente o tema de *code smells* específicos do Django, apenas trabalhos como [Aniche et al. 2018] que faz um estudo mais geral sobre *code smells* em arquiteturas MVC.

A revisão cinzenta é uma técnica de revisão literária que visa a identificação e análise de informações disponíveis em fontes não convencionais, como fóruns, blogs, documentação técnica não oficial e códigos em repositórios públicos. Como discutido em [Kamei 2022], a literatura cinza desempenha um papel importante na pesquisa em engenharia de software pois, além de oferecer uma gama maior de perspectivas, oferece uma riqueza de conhecimento prático e experiências diretamente do campo de conhecimento e isso é muito relevante pois nesta área as ferramentas e tecnologias evoluem muito rápido, por vezes mais rápido do que o ciclo de publicação acadêmica consegue acompanhar.

A pesquisa teve como base, além da própria documentação do Django, diversas publicações de fóruns e blogs focados no desenvolvimento web com o framework e suas boas práticas. Como também é citado em [Kamei 2022], na revisão cinzenta é importante garantir a validação das informações e sua confiabilidade. Dessa forma, neste trabalho além de realizar verificação prática para cada fonte, também foi realizada a validação das informações com base na documentação oficial do framework. Nesse sentido os *code smells* encontrados e que não puderam ser validados, não foram considerados para este trabalho.

Após identificar os problemas de código mais comuns ao Django, a segunda etapa do trabalho se propõe ao desenvolvimento de uma ferramenta de análise estática de código. A análise estática de código se trata de um processo de verificação automatizada do código-fonte em busca de identificar erros, padrões de codificação e auxiliar na melhoria da qualidade do código [JetBrains 2024] . Esta ferramenta tem o objetivo de identificar padrões que indicam a ocorrência de algum dos *code smells* encontrados na pesquisa. Para validação tanto da ferramenta quanto das pesquisas foram selecionados e analisados diversos repositórios públicos de projetos Django.

4. Code Smells

Nesta seção são descritos e exemplificados os *code smells* encontrados durante as pesquisas.

4.1. Views Complexas

Idealmente views devem ser simples e curtas [Sallie Sorenson 2022], uma view complexa pode gerar problemas de performance além de torná-la dificil de entender e manter. Na

view apresentada na Listing 1, ela acumula várias responsabilidades diferentes: gerenciar autenticação, carregar informações de perfil do usuário, processar dados do formulário e renderizar diferentes respostas. O ideal seria separar esta view em outras views ou funções diferentes para que cada uma trate de uma responsabilidade específica. O uso de *Class-based Views* [Django Software Foundation 2024a] também pode ser mais adequado, uma vez que views no formato de função costumam ter uma maior tendência a sair de controle e se tornarem mais complexas do que deveriam.

```
def user_profile_view(request):
    if request.method == 'POST':
        user_form = UserForm(request.POST)
        profile_form = ProfileForm(request.POST, request.FILES)
        if user_form.is_valid() and profile_form.is_valid():
            user = user_form.save()
            profile = profile_form.save(commit=False)
            profile.user = user
            profile.save()
            user = authenticate(username=user.username, password=
   request.POST['password'])
            login(request, user)
            return redirect('profile_success')
    else:
        user_form = UserForm()
        profile_form = ProfileForm()

    return render(request, 'user/profile.html', {
        'user_form': user_form,
        'profile_form': profile_form
    })
```

Listing 1. Exemplo de God View em Django

4.2. Criação de objetos dentro de laço

Criar objetos dentro de um laço é considerada uma das más práticas mais comuns no desenvolvimento com Django. Apesar de parecer uma saída "mais lógica" para a criação de objetos em lotes, ela realiza mais consultas ao banco de dados do que o necessário [Manuel Matosevic 2022]. O Django possui a função bulk_create que é utilizada exatamente para este propósito e possui maior eficiência computacional. O mesmo se aplica para a atualização em lotes e para isso o Django possui a função bulk_update. A seguir temos um exemplo do *code smell* e da forma correta para executar a ação de criação em lotes.

```
for index in range(number):
    User.objects.create(username=f"user-{index}")
```

Listing 2. Exemplo de criação de objetos dentro de um loop for

```
users = [User(username=f"user-{index}") for index in range(number)]
User.objects.bulk_create(users)
```

Listing 3. Exemplo de criação de objetos usando bulk_create

No primeiro código (Listing 2) [Michel Sabchuk 2021], o Django irá fazer um número de consultas ao banco de dados equivalente ao tamanho da lista que está percorrendo. Dessa forma, a depender do tamanho da lista, pode-se ter um gargalo de desempenho muito grande. No código seguinte (Listing 3) [Michel Sabchuk 2021], temos o uso do bulk_create que permite a não utilização do laço para a criação dos objetos e reduz consideravelmente o tempo de execução da criação dos objetos no banco de dados.

4.3. Não uso de prefetch_related e select_related

O Django possui duas funcionalidades poderosas para trabalhar com objetos que possuem relacionamentos e que auxiliam no acesso aos objetos relacionados. O prefetch_related é utilizado quando se está trabalhando com relacionamentos *many-to-many* e o select_related com relacionamentos *one-to-one*. Utilizando prefetch_related e select_related, reduzimos significativamente o número de consultas ao banco de dados, melhorando o desempenho da aplicação [Goutom Roy 2019]. Dessa forma, o não uso dessas funcionalidades quando se está buscando objetos relacionados torna a aplicação mais lenta, podendo se tornar um problema maior no futuro.

```
1 def store_list():
2     queryset = Store.objects.all()
3     stores = []
4     for store in queryset:
5         books = [book.name for book in store.books.all()]
6         stores.append({'id': store.id, 'name': store.name, 'books':
      books})
7     return stores
```

Listing 4. Exemplo: Não uso do prefetch_related

```
1 def store_list_prefetch_related():
2     queryset = Store.objects.prefetch_related('books')
3     stores = []
4     for store in queryset:
5         books = [book.name for book in store.books.all()]
6         stores.append({'id': store.id, 'name': store.name, 'books':
      books})
7     return stores
```

Listing 5. Exemplo: uso adequado do prefetch_related

No primeiro exemplo (Listing 4) [Goutom Roy 2019], ao buscar os objetos de lojas, os objetos relacionados de livros não são carregados nesta consulta ao banco de dados, apenas o identificador deles. Para que os objetos de livros relacionados a cada loja sejam carregados, o Django precisa fazer uma consulta adicional ao banco de dados na chamada da linha número 5. Já no código seguinte (Listing 5) [Goutom Roy 2019], ao utilizar o prefetch_related o django carrega previamente os objetos de livros relacionados aos objetos da tabela de lojas, não sendo necessária a execução de uma consulta adicional.

4.4. Fat Models

Os *Fat Models* são modelos que possuem todas as regras e lógicas de negócio, aumentando o nível de complexidade esperado na camada *Model*. Esses modelos são difíceis de testar e violam o princípio da responsabilidade única [Vercosa 2020].

```python
class Order(models.Model):
    customer = models.ForeignKey(Customer, on_delete=models.CASCADE)
    order_date = models.DateTimeField(auto_now_add=True)
    shipped_date = models.DateTimeField(null=True, blank=True)
    status = models.CharField(max_length=10)

    def calculate_total_price(self):
        # Implementacao complexa que calcula o preco total
        pass

    def is_eligible_for_discount(self):
        # Logica complexa para determinar a elegibilidade para desconto
        pass

    def update_stock(self):
        # Atualiza o estoque baseado na ordem
        pass

    def send_confirmation_email(self):
        # Envia um email de confirmacao para o cliente
        pass
```

Listing 6. Exemplo de Fat Model em Django

O código acima exemplifica um modelo que possui diversas funções com objetivos diferentes, centralizando lógicas de negócio e aumentando a complexidade do modelo. A medida que o projeto cresce, modelos como este se tornam cada vez mais difíceis de manter. O ideal seria separar essas responsabilidades em funções utilitárias para que possam ser utilizada em todo o projeto sem aumentar a complexidade do modelo.

4.5. Templates com lógica complexa

Templates que possuem muita lógica podem se tornar difíceis de entender e de manter [Codium AI 2023]. Este problema acontece ao adicionar várias ações que deveriam ser feitas em funções à parte ou na view que utiliza o template. O código a seguir exemplifica este *code smell*, onde o template está realizando várias ações que fogem do seu objetivo principal que é a apresentação dos dados.

```
{% for product in product_list %}
    {% if product.is_available %}
        <div class="product">
            <h2>{{ product.name }}</h2>
            <p>{{ product.description }}</p>
            {% if user.is_authenticated %}
                <p>Preco: {{ product.price }}</p>
                {% if product.discount %}
                    <p>Desconto: {{ product.discount }}%</p>
                    <p>Preco com Desconto: {{ product.price|discount:
    product.discount }}</p>
                {% endif %}
            {% else %}
                <p>Entre para ver os valores</p>
            {% endif %}
        </div>
    {% endif %}
```

```
17 {% endfor %}
```

Listing 7. Exemplo de lógica complexa em um template Django

No código acima temos verificações de autenticação e de regra de negócio que tornam o template mais complexo do que deveria, desta forma, o ideal é que todas as reponsabilidades desse tipo sejam implementadas nas views ou models e as informações cheguem prontas ao template para que só tenha a responsabilidade de exibição.

5. Desenvolvimento da Ferramenta

Após a pesquisa para levantar os principais *code smells* encontrados no desenvolvimento em Django, o outro objetivo deste projeto foi o desenvolvimento de uma ferramenta capaz de identificar possíveis problemas de código Django atráves de análise estática.

Para isto, a ferramenta foi desenvolvida em Python utilizando a biblioteca AST [Python Software Foundation a], nativa da linguagem, que é capaz de converter código-fonte Python em Árvore de Sintaxe Abstrata (AST, do inglês Abstract Syntax Tree). A partir da AST de um código é possível identificar padrões para reconhecer o que cada trecho de código faz. Neste sentido, a ferramenta desenvolvida lê arquivos de código de projetos django, os converte em AST e então busca padrões de code smells para indicar um possível problema de código.

O desenvolvimento da ferramenta teve como foco inicial a análise de padrão para dois dos *code smells* encontrados na pesquisa. A escolha dos problemas a serem abordados na ferramenta se deu com base no quanto o problema é recorrente entre os *smells* encontrados e também se buscou a variedade baseada nas camadas do Django. Como a camada *Template* é composta majoritariamente por arquivos de código HTML, os dois problemas escolhidos são referentes às camadas *Model* e *View*, uma vez que a ferramenta foi desenvolvida em Python e a biblioteca AST possui suporte apenas para código da linguagem. Dessa forma, a ferramenta é capaz de identificar os *code smells* de *Views Complexas* e *Criação de objetos dentro de um laço*.

A Figura 2 ilustra o fluxo de execução da ferramenta. Primeiro ela lê os arquivos de código-fonte e em seguida os converte para a forma de árvore de sintaxe abstrada, então as árvores passam pelas funções de checagem que irão verificar a existência de padrões que remetem a *code smells* para que, em caso positivo, o resultado seja gerado. O código-fonte da ferramenta pode ser acessado no GitHub[1].

[1] `https://github.com/eduardoluiz7/django-smells-checker`

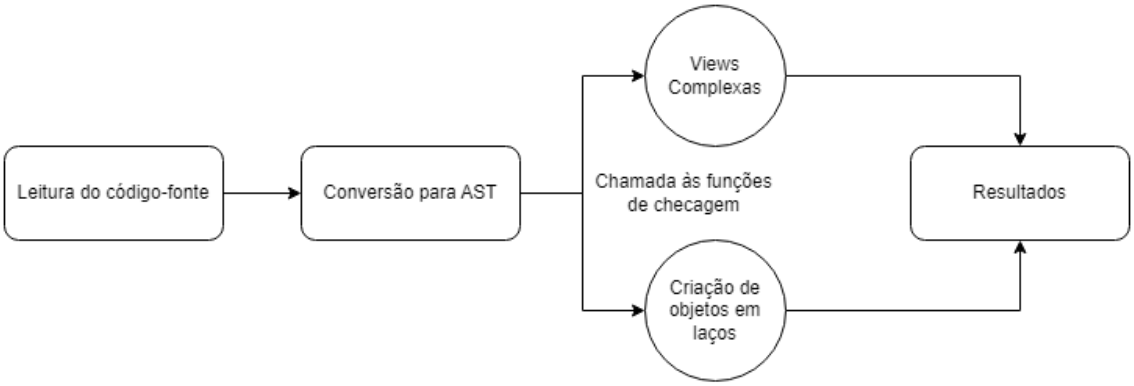

Figura 2. Fluxo de Execução da ferramenta

6. Testes e Resultados

Para os testar a eficiência e capacidade da ferramenta em identificar os *code smells* selecionados, foram analisados dez repositórios públicos de projetos Django, de diferentes tamanhos e complexidades. Na Tabela 3 estão listados os repositórios utilizados nos testes, os repositórios foram classificados por tamanho e complexidade com base na quantidade de módulos, modelos e views como mostra a Tabela 1.

Tabela 1. Classificação dos repositórios por tamanho

Tamanho	Módulos	Modelos	Views
Pequeno	1 a 3	1 a 10	1 a 10
Médio	4 a 10	11 a 30	11 a 30
Grande	11+	31+	31+

Para cada *code smell* encontrado a ferramenta salva em um *dataframe* informações como o tipo do problema, nome do arquivo em que ocorre o *smell*, linhas iniciais e finais onde o problema acontece no arquivo, etc. Com essas informações se torna possível localizar com facilidade o problema para assim resolvê-lo e, no caso dos testes, analisar a recorrência dos smells em cada projeto.

Como é possível ver na Tabela 2, nos 10 repositórios testados foram encontrados 13 casos de *Views Complexas* e 4 casos de *Criação de objetos dentro de laço*, com uma média por repositório de 1.3 e 0.4 casos respectivamente.

A partir dos testes foi possível perceber que todos os projetos que utilizam suas views em formato de função foram apontados com o problema de *Views Complexas*. Algo esperado, uma vez que views neste formato possuem uma organização mais complicada, enquanto projetos que utilizam views template (Class Views) dispõem de uma separação de responsabilidade natural e tem uma menor complexidade de organização.

No caso do *code smell* de *Criação de objetos dentro de laço* apenas três dos repositórios possuíam uma ação de criação em lotes, dentre esses três projetos apenas dois

foram apontados com este problema enquanto o repositório restante fazia uso correto da função bulk_create.

Tabela 2. Resultados dos testes com a ferramenta em repositórios públicos

Code Smell	Total	Média por Repositório
Views Complexas	13	1.3
Criação de objetos dentro de laço	4	0.4

Tabela 3. Repositórios utilizados para análise da ferramenta

Repositório	Descrição	Tamanho/Complexidade
Karrot-backend	Servidor para plataforma de voluntariado	Grande
django-rest-authemail	API para autenticação utilizando endereço de e-mail	Médio
real-time-chat-api	Aplicação de chat em tempo real	Pequeno
django-oscar	Aplicação de e-commerce	Grande
event-calendar	Calendário de eventos	Pequeno
email_microservice	Micro Serviço de e-mail	Médio
schedulingApp	Sistema para organização de cursos	Pequeno
Recetario	Sistema de receitas	Pequeno
metpetdb-py-1	Implementação de sistema educacional	Médio
django-library	Sistema de gerenciamento de biblioteca	Pequeno

7. Considerações e Trabalhos Futuros

Este trabalho detalhou a identificação e análise de alguns dos *code smells* mais recorrentes em projetos Django, resultando no desenvolvimento de uma ferramenta capaz de identificar automaticamente alguns desses *code smells*. Ao aplicar esta ferramenta em uma variedade de projetos Django se pode notar sua eficácia na detecção das más práticas de código para as quais ela foi projetada. Os resultados confirmam sua utilidade e

proporcionam um caminho de evolução para a abordagem das melhores práticas de desenvolvimento dentro da comunidade de desenvolvedores.

Como perspectivas futuras, se propõe o avanço do escopo e funcionalidade da ferramenta. Aumentar o a quantidade de *code smells* que a ferramenta é capaz de identificar irá expandir significamente sua aplicabilidade e eficiência no auxilio à desenvolvedores. Também se vê como um passo importante a implementação da integração da ferramenta com ambientes de desenvolvimento integrado (IDEs) populares que facilitarão o acesso dos desenvolvedores.

Além disso, realizar mais casos de estudo em projetos de código aberto oferecerá mais informações de como a ferramenta pode ser melhorada e também quais outros *code smells* podem ser abordados para que possamos aumentar a abrangência da discussão e análise de más práticas no desenvolvimento Django.

Em resumo, este trabalho oferece uma contribuição para o desenvolvimento Django e estabelece uma base para que a discussão e estudo sobre o assunto evolua. Que mais pesquisas futuras surjam e possam continuar fortalecendo e incentivando a qualidade, eficiência e sustentabilidade do desenvolvimento Django.

8. Referências

Aniche, M., Bavota, G., Treude, C., Gerosa, M. A., and Deursen, A. (2018). Code smells for model-view-controller architectures. *Empirical Softw. Engg.*, 23(4):2121–2157.

Codium AI (2023). Django templates: Best practices for web development. https://www.codium.ai/blog/django-templates-best-practices-for-web-development/. Acessado em: 10 de Fevereiro de 2024.

Django Software Foundation. Django documentation (5.0) - faq: General, seção "django appears to be a mvc framework, but you call the 'controller' the 'view' and the 'view' the 'template'. how come you don't use the standard names?". https://docs.djangoproject.com/pt-br/5.0/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-v
Acessado em: 20 de Dezembro de 2023.

Django Software Foundation (2024a). class-based-views - django documentation (5.0). https://docs.djangoproject.com/en/5.0/topics/class-based-views/. Acessado em: 20 de Janeiro de 2024.

Django Software Foundation (2024b). Django documentation (5.0). Acessado em: 20 de Janeiro de 2024.

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

Goutom Roy (2019). Django select_related and prefetch_related. https://betterprogramming.pub/django-select-related-and-prefetch-related-f23043fd635d. Acessado em: dia mês ano.

JetBrains (2024). Static code analysis. https://www.jetbrains.com/pt-br/

teamcity/ci-cd-guide/concepts/static-code-analysis/. Acessado em: 10 de Fevereiro de 2024.

Kamei, F. K. (2022). *Understanding and Supporting the Decision-making Whether to Use Grey Literature in Software Engineering Research*. Ph.d. thesis, Universidade Federal de Pernambuco, Centro de Informática, Recife. Programa de Pós-Graduação em Ciência da Computação.

Kokol, P., Kokol, M., and Zagoranski, S. (2020). Code smells: A synthetic narrative review. *Library Philosophy and Practice (e-journal)*.

Manuel Matosevic (2022). Using django bulk_create and bulk_update. https://zerotobyte.com/using-django-bulk-create-and-bulk-update/. Acessado em: 10 de Fevereiro de 2024.

Mariana Clark (2024). Alternativas ao django. https://blog.back4app.com/pt/alternativas-ao-django/. Acessado em: 20 de Janeiro de 2024.

Michel Sabchuk (2021). Load large sets of data into a postgres database using python/django. https://michelts.medium.com/load-large-sets-of-data-into-a-postgres-database-using-python-django-7
Acessado em: 15 de Fevereiro de 2024.

Python Geeks (2024). Features of python django framework. https://pythongeeks.org/features-of-python-django-framework/. Acessado em: 10 de Fevereiro de 2024.

Python Software Foundation. ast - python documentation. https://docs.python.org/3/library/ast.html. Acessado em 5 de Janeiro de 2024.

Python Software Foundation. Python documentation. https://www.python.org/doc/. Acessado em: 20 de Dezembro de 2023.

Sallie Sorenson (2022). 10 django views best practices. https://climbtheladder.com/10-django-views-best-practices/. Acessado em: 15 de Janeiro de 2024.

Vercosa, J. (2020). Django model guideline. https://jairvercosa.medium.com/django-model-guideline-d48a96c9b38c. Acessado em: 20 de Janeiro de 2024.