Evidências Empíricas sobre a Efetividade da SmellDSL para Especificar Anomalias de Código: Um Estudo Controlado e Perspectivas Futuras

Arthur Inda Rocha
arthurin915@edu.unisinos.br
Universidade do Vale do Rio dos Sinos
São Leopoldo, Rio Grande do Sul, Brazil

Kleinner Farias
kleinnerfarias@unisinos.br
Universidade do Vale do Rio dos Sinos
São Leopoldo, Rio Grande do Sul, Brazil

ABSTRACT

A presença de anomalias de código em sistemas de software frequentemente indica problemas arquiteturais, potencialmente resultando em degradação. Desenvolvedores usam heurísticas (por exemplo, encontradas no SonarQube) para identificá-las, ou as especificam manualmente, usando alguma linguagem, tais como a SmellDSL, uma linguagem específica de domínio que permite aos desenvolvedores especificar anomalias de código. Atualmente, as técnicas empregadas na identificação dessas anomalias se baseiam em heurísticas (por exemplo, Jdeodorant, Jspirit, SonarQube e PMD) ou especificações (por exemplo, SmellDSL). Enquanto as técnicas baseadas em heurísticas tenham tido uma maior atenção nos últimos anos, as baseadas em especificação têm sido negligenciadas. Consequentemente, desenvolvedores acabam usando tais abordagens sem quaisquer evidências empíricas. Este estudo, portanto, reporta um experimento controlado com o intuito de analisar a eficácia da SmellDSL. No total, 35 participantes executaram 8 tarefas experimentais, representando 280 cenários de avaliação considerando a corretude da especificação, taxa de erro e esforço de especificação. Os resultados, suportados por testes estatísticos, sugerem que a SmellDSL exigiu baixo esforço para especificar anomalias (menos que 15 minutos) e ajudou na especificação correta das anomalias. Além disso, a maioria dos participantes (77%) indicou facilidade na utilização da SmellDSL para especificar anomalias. Por fim, este estudo contribuiu com evidências empíricas que exploram melhor as capacidades da SmellDSL em cenários realísticos de especificação de anomalias. Tais evidências representam um primeiro passo em uma agenda mais ambiciosa de pesquisa.

KEYWORDS

SmellDSL; Refactoring; Estudo Empírico

1 INTRODUÇÃO

As anomalias de código, ou *code smells* são estruturas internas de código-fonte que desafiam os princípios de desenvolvimento, afetando negativamente a qualidade interna do software [4, 22]. Anomalias manifestam-se em códigos que são frequentemente modificados e estendidos, fazendo com que suas alterações gerem não conformidades. No cenário ideal, essas não conformidades devem ser especificadas, definindo os padrões de comportamento que o código deve seguir, e corrigidas, caso identificadas. Há duas alternativas comumente utilizadas para identificar anomalias de código, a primeira é basear-se em heurísticas, através de ferramentas como o SonarQube, e a segunda é realizar a especificação dessas anomalias utilizando uma linguagem, através de ferramentas como a SmellDSL. O vigente trabalho pretende focar em analisar e avaliar técnicas de identificação de anomalias seguindo o formato de especificação, utilizando a SmellDSL como objeto de estudo.

Conforme [14], é possível concluir que a utilização de uma linguagem específica de domínio para especificação de anomalias de código facilita a construção de códigos de alta qualidade. Porém, o meio científico carece de estudos empíricos e experimentos controlados para avaliar a efetividade de uma DSL específica, dificultando a utilização de diferentes DSLs pelos desenvolvedores, como é o caso da SmellDSL. Sem o conhecimento empírico com relação a sua efetividade, o uso de uma ferramenta se torna questionável.

Foram encontrados alguns trabalhos acadêmicos que visam estudar e investigar anomalias de códigos e como mitigá-las no contexto de desenvolvimento de software. [2] observaram a relação entre refatoração de código e *code smells*, gerando métricas de qualidade de código e comprovando que os desenvolvedores não observam a criação de diversas anomalias de código ao trocarem de contexto de desenvolvimento. Em [11], os autores realizaram um experimento controlado com desenvolvedores para verificar a priorização de correção de anomalias de código utilizando uma ferramenta capaz de sinalizar as anomalias mais críticas no desenvolvimento. Em [1], os autores verificaram através de um estudo empírico a usabilidade de linguagens de domínio na manutenção de softwares e comentam que há muito a ser evoluído sobre DSLs no campo acadêmico, e que estudos que promovem métricas quantitativas em experimentos controlados serão de grande ajuda para evoluir o conhecimento a respeito da efetividade de linguagens específicas de domínio para diferentes contextos de aplicação.

Este trabalho, portanto, apresenta um experimento controlado, cujo objetivo é investigar a efetividade da SmellDSL, através de atributos de corretude, erro e esforço de especificação. Para isso, um experimento controlado foi projetado e executado com 35 desenvolvedores de software para investigar três questões de pesquisa propostas pelo trabalho. Cada participante executou 8 tarefas experimentais, gerando 280 cenários de avaliação. Foram utilizadas métricas para a coleta de dados, análises descritivas e estatística e análises dos resultados de cada participante através de um processo dividido em cinco etapas organizados em três fases. Os resultados obtidos apontam que a SmellDSL é uma excelente ferramenta para especificação de anomalia, obtendo 98% de corretude dos participantes analisados, e sinalizando que esta linguagem pode vir a se tornar uma ferramenta excelente no futuro quando finalizada, comprovando que linguagens específicas de domínio são uma solução no campo de especificações de anomalias de código.

Os resultados obtidos através desse experimento beneficiam profissionais com interesse na área de: Desenvolvimento e Qualidade de Software, pois permite integrar uma ferramenta que definiria e identificaria padrões de anomalias de códigos em seus sistemas e pesquisadores, pois este estudo levanta informações a respeito da utilização de linguagens específicas de domínio para especificar anomalias de código, em conjunto com um teste empírico conduzido para coleta de dados e métricas, enriquecendo a comunidade devido ao fato de não haver muito conteúdo disponível sobre este assunto na atualidade.

O estudo está dividido conforme a seguinte estrutura: A Seção 2 descreve o referencial teórico, com os principais conceitos para entendimento do estudo proposto. A Seção 3 aborda os trabalhos relacionados, explorando o processo de seleção utilizado e também realizando um comparativo destes com o presente. A Seção 4 apresenta a descrição da metodologia para desenvolvimento do estudo. A Seção 5 descreve a análise dos resultado obtidos. E por fim, a Seção 6 traça algumas considerações finais e descreve trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Esta Seção apresenta os principais conceitos para o entendimento do trabalho desenvolvido. Para isso, a Seção 2.1 descreve os conceitos essenciais para o entendimento de anomalias de código. A Seção 2.2 apresenta embasamento a respeito do objeto de pesquisa do trabalho, a SmellDSL, descrevendo suas premissas, estrutura, gramática e fornecendo um exemplo de utilização. Por fim, a Seção 2.3 discorre brevemente sobre a identificação de anomalias de código utilizando ferramentas heurísticas e alguns de seus efeitos e limitações.

2.1 Anomalias de código

Segundo [8], anomalias de código são partes da estrutura de um código que prejudicam a qualidade do software por violar determinado padrão ou princípio de desenvolvimento, dificultando sua manutenção e evolução. Conforme mencionado em [15], a presença de anomalias de código em um software é um forte indicativo de degradação arquitetural, impactando negativamente em sua qualidade e, em muitos casos, performance.

Desenvolvedores normalmente tem dificuldades para identificar anomalias de código [17]. As anomalias de código podem ser classificadas de diferentes formas, porém, algumas definições padrões já foram criadas para especificar anomalias comumente encontradas em *softwares*, conforme mostra a Figura 1.

Um exemplo de categoria de anomalia de código que é encontrada na maioria dos códigos fontes são os *Bloaters*, que se referem a códigos inchados, cujo o tamanho e complexidade se estendem para além do esperado, como por exemplo, a anomalia *Long Method* se refere a métodos com um excessivo número de linhas.

[21] descreve uma característica interessante desses tipos de anomalias, comentando que os *Bloaters* são difíceis de aparecer em códigos em fases iniciais, pois são construídas e fortificadas ao longo do tempo, principalmente quando não identificadas com antecedência.

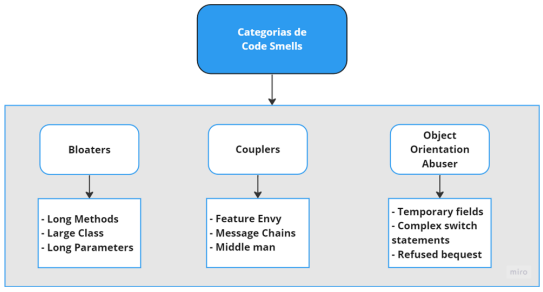

Figure 1: Categorias de code smells

Na Figura 2 é possível verificar a anomalia de código *Object Orientation Abuser*. Nesse código uma classe *Bird* (pássaro) é representada, mostrando o seu método *getSpeed* que implementa um tipo de retorno diferente para cada tipo de pássaro. A primeira vista não há nenhuma anomalia de código, o que faz com que desenvolvedores inexperientes não identifiquem o problema que está acontecendo. Na verdade este código é um exemplo da anomalia *Object Orientation Abuser*, pois viola os principios da orientação a objetos ao retornar diferentes tipos de implementação ao invés de resolver o código utilizando principios básicos da orientação a objetos, como a herança.

Para visualizar um exemplo de como seria este código com suas anomalias corrigidas, veja a Figura 3. A solução foi criar uma classe para cada tipo de pássaro e sobrescrever o método da classe base para cada um, já que cada um retorna um valor diferente. Dessa forma, além de seguir os princípios da orientação a objetos, o desenvolvedor garante mais qualidade ao código desenvolvido, melhorando sua manutenibilidade ao isolar partes de código que não deveriam se misturar. Isso mostra como a identificação e resolução de anomalias pode melhorar drasticamente a qualidade de um código.

Hoje em dia, uma variedade de ferramentas de software foram desenvolvidas para especificar e automatizar a detecção de anomalias de código, cada uma com sua peculiaridade e especialidade [12]. Uma delas, é a SmellDSL, que foca em permitir que o desenvolvedor especifique suas próprias regras para detecção de anomalias de código, visto que, segundo [12], determinar quando um código configura uma anomalia de código ainda é subjetivo, devido a carência de padrões para as suas determinações.

2.2 SmellDSL

A SmellDSL é uma linguagem específica de domínio projetada exclusivamente para especificação de anomalias de código. Ela foi criada pelo doutorando Robson Keemps em conjunto com o professor doutor Kleinner Farias, em 2018, com o foco em auxiliar desenvolvedores a especificar anomalias de código de forma mais livre e abrangente, visando permitir aos desenvolvedores definirem regras e anomalias próprias para o contexto em que se encontram através de notações e regras customizáveis.

A linguagem SmellDSL foi projetada baseando-se em algumas premissas, sendo elas descritas abaixo:

Especificação sensível ao desenvolvedor: permitir que desenvolvedores especifiquem anomalias que se encaixam no seu

```
class Bird {
  // ...
  getSpeed(): number {
    switch (type) {
      case EUROPEAN:
        return getBaseSpeed();
      case AFRICAN:
        return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
      case NORWEGIAN_BLUE:
        return (isNailed) ? 0 : getBaseSpeed(voltage);
    }
    throw new Error("Should be unreachable");
  }
}
```

Figure 2: Exemplo de code smell - Object Orientation Abuser [9]

```
abstract class Bird {
  // ...
  abstract getSpeed(): number;
}

class European extends Bird {
  getSpeed(): number {
    return getBaseSpeed();
  }
}
class African extends Bird {
  getSpeed(): number {
    return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
  }
}
class NorwegianBlue extends Bird {
  getSpeed(): number {
    return (isNailed) ? 0 : getBaseSpeed(voltage);
  }
}

// Em algum lugar no código do cliente
let speed = bird.getSpeed();
```

Figure 3: Code smell - Object Orientation Abuser corrigido [9]

contexto de trabalho ao mesmo tempo que permite especificar anomalias já catalogadas.

Notações de orientação a objetos: utilizar conceitos e notações de orientação a objeto para as especificações das anomalias, visando facilitar a adoção de desenvolvedores familiarizados com esse paradigma.

Separação de definição: separar a definição de anomalias das regras para identificá-las, com objetivo de fornecer mais liberdade ao desenvolvedor para identificar certos tipos de anomalia, pois dependendo do contexto do desenvolvedor ou da funcionalidade desenvolvida, as mesmas anomalias poderão ser identificadas de formas diferentes. Por exemplo, a anomalia *God Class* define uma classe que contém muitas variáveis, métodos e linhas de código, porém a definição concreta de como quantificar esses atributos varia de desenvolvedor para desenvolvedor.

Alguns conceitos chaves da SmellDSL é utilizar atributos para caracterizar as anomalias e descrever as consequências geradas pela sua ocorrência, também conhecido como "sintoma", e relatar uma possível recomendação para mitigar sintomas indesejados, ilustrado dentro da linguagem como "tratamentos". Aliado a estes conceitos,

a SmellDSL também foi projetada para a utilização de notações do paradigma de orientação a objetos, permitindo que as anomalias especificadas compartilhem seus atributos, sintomas e tratamentos através de herança. Isso permite ao desenvolver especificar anomalias de códigos genéricas ou específicas, além de permitir a criação de regras de identificação para cada um.

A seguir serão descritas as estruturas implementadas pela SmellDSL para que a realização das especificações de anomalias de código sejam possíveis. Sendo elas: *smelltype*, *smell*, *feature*, *symptom*, *treatment* e *rule*.

- **Smelltype:** define um tipo de *smell* ao qual os *smells* podem ser definidos. Seguindo a Figura 1, os *Smelltypes* seriam os *Bloaters*, *Couplers* e *Object-oriented-abusers*. Basicamente os *smelltypes* servem para criar classificações para representar um ou mais *smells*.
- **Smell:** é a palavra reservada utilizada para definir a anomalia em si, para a SmellDSL ela representa um modelo lógico para definir uma estrutura não desejada dentro de um código fonte. Um *smell* pode conter *features*, *symptom* e *treatment*.
- **Symptom:** utilizado dentro de *smells* para definir o problema que o *smell* causa ao código em que há sua ocorrência.
- **Treatment:** também utilizado dentro de *smells*, porém essa palavra reservada descreve possíveis soluções para resolução do sintoma.
- **Feature:** definido dentro de *smells* representar os atributos que um *smell* pode conter ou possíveis valores que o desenvolvedor pode utilizar para identificar o *smell* posteriormente utilizando rules, como por exemplo o número de linhas de um bloco de código para verificar se é um *Long Method*.
- **Rule:** são regras que os desenvolvedores criam para determinar a identificação/ocorrência de um ou mais *smells* dentro do código fonte. Para a criação de rules é preciso criar condições de verificação das *features* de um ou mais *smells* com os seus possíveis valores. É possível descrever o que fazer para cada regra quando suas condições forem atendidas, significando que o *smell* foi identificado no código fonte.

Na figura 4 é apresentado um exemplo de código de especificação de anomalias utilizando a SmellDSL. Nele, esta sendo definido um *smelltype Bloaters*, três *smells* que são da categoria *Bloaters* e duas regras, especificadas utilizando a estrutura *rules*. A primeira regra diz respeito a condições de gatilho construídas para o *smell Long Method*, já a última combina diferentes *features* dos *smells* para criar uma condição de gatilho mais complexa.

O primeiro *smell* é o *LongMethod*, conforme visto em Figuras anteriores, sua especificação é realizada pensando em analisar o número de linhas, complexidade e número de parametros de um método. Esta especificação foi feita utilizando a estrutura *feature*. Além disso ele também possui um sintoma (*symptom*) chamado de "Objeto muito grande" e um tratamento (*treatment*) chamado de "Quebrar objeto em partes". Atualmente as definições de *symptom* e *treatment* estão em fase inicial, suportando somente textos corridos, mas a ideia no futuro é suportar tipos de estruturas mais complexos para dar mais flexibilidade a especificação. Os próximos *smells* seguem a mesma regra de especificação da anomalia citada acima, porém um é para identificar o *smell Data Clumps* e o outro para identificar o *smell Primitive Obsessions*.

Por fim, podemos observar a sintaxe de definição das regras (*rules*). Ao analisar a "regra1", é possível verificar que seu gatilho de execução é o número de linhas de um método ser menor que 50 linhas e sua complexidade for alta. Dessa forma, a regra retorna uma mensagem ao usuário informando que ele deve reavaliar a codificação pois o método está muito complexo. Já na "regra2", é criado uma condição complexa, combinando *features* de diferentes *smells* para retornar uma mensagem para o usuário, apresentando o poder e flexibilidade de especificação da SmellDSL.

Nota-se que as condições de regras ficam a critério do usuário que está especificando, justamente por propor essa liberdade de criar regras para identificação de *smells* que torna linguagens de especificação, como a SmellDSL, objetos de estudo muito interessantes no que diz respeito a sua efetividade.

Para facilitar o entendimento de como funciona a escrita dos elementos da SmellDSL, foi incluído a Figura 5, descrevendo um diagrama sintático de como a escrita da SmellDSL funciona para cada uma de suas estruturas/palavras reservadas. Em conjunto com a Figura 4, é possível entender como escrever as estruturas propostas pela linguagem.

A gramática da SmellDSL foi definida utilizando notação BNF e incorporada no *framework* Xtext [1]. Como o XText é um plug-in do ambiente de desenvolvimento Eclipse, a SmellDSL está disponível para uso dentro do próprio Eclipse e conta com uma versão inicial das estruturas descritas acima, permitindo ao desenvolvedor criar suas especificações de forma conceitual.

Especula-se que hoje haja uma degradação na qualidade dos softwares, sendo criados com problemas arquiteturais crônicos e lotados de anomalias de código que afetam negativamente a qualidade geral do sistema desenvolvido. Além disso, a falta de especificação e correção de anomalias de código apenas promove e reafirma a presença de um software parcialmente ou inteiramente degradado. [16]. Portanto, acredita-se que a utilização de uma linguagem que permita ao desenvolvedor especificar as anomalias do seu código, de forma livre e adaptável a diferentes contextos de desenvolvimento seja de suma importância no cenário atual.

2.3 Ferramentas heurísticas

Embora as ferramentas atuais apresentem uma ampla gama de métricas e heurísticas para detectar anomalias de código, estudos recentes indicam que métricas e regras comumente acionadas e quantificadas geram muitos alertas, fazendo com que os desenvolvedores ignorem tais indicações [23]. Quando os conceitos não são bem definidos e compreendidos, tais práticas podem ter o efeito oposto, onerando a tarefa de desenvolvimento, reforçando que certas práticas de revisão de código podem aumentar o risco de degradação do projeto, incluindo discussões demoradas e uma alta taxa de tempo de desenvolvimento. Alguns exemplos de ferramentas heurísticas seriam: SonarQube, PMD e JSPIRIT.

Apesar de sua boa precisão, trabalhos anteriores apontaram três limitações importantes que podem impedir o uso de detectores de anomalias de código na prática:

[1] XText: https://www.eclipse.org/Xtext/

```
smelltype Bloaters

smell LongMethod extends Bloaters{
    feature numero_linhas is Nominal with threshold menor50, entre50e100,maior100
    feature complexidade is Nominal with threshold Alto, Medio, Baixo
    feature numero_parametros is Interval with threshold dez,vinte, trinta

    symptom ObjetoMuitoGrande
    treatment QuebrarObjetoPartes
}


smell PrimitiveObsession extends Bloaters{
    feature Numero_variaveis_Strings is Interval with threshold TEN, TWENTY, THIRTY, FORTY, FIFTY

    symptom Apenas_um_campo_para_armazenar_alguns_dados
    treatment Substituir_Valor_de_Dados_por_Objeto
}

smell DataClumps extends Bloaters{
    feature Num_Parm_Identico is Interval with threshold TEN, TWENTY, THIRTY, FORTY, FIFTY

    symptom Estrutura_de_programa_deficiente
    treatment Substituir_ou_eliminar_informacoes_identicas
}


rule regra1 when LongMethod.numero_linhas <= numero_linhas.menor50 AND LongMethod.complexidade == complexidade.Alto
then Reavaliar a codificacao a sua complexidade

rule regra2 when LongMethod.numero_linhas > numero_linhas.maior100 OR
                 LongMethod.complexidade == complexidade.Baixo OR
                 DataClumps.Num_Parm_Identico >= Num_Parm_Identico.TEN AND
                 DataClumps.Num_Parm_Identico < Num_Parm_Identico.TWENTY
then Reavaliar a codificacao devido conter informacoes identicas
```

Figure 4: Exemplo de código - SmellDSL

- Subjetividade dos desenvolvedores em relação aos *code smells* detectados por tais ferramentas;
- Má concordância entre diferentes detectores;
- Dificuldades em encontrar bons limiares para utilização na detecção;

Para superar essas limitações, o uso de técnicas de aprendizado de máquina representa uma área de pesquisa crescente, pensando nisso futuramente a SmellDSL pretende utilizar-se de *Machine Learning* [20].

3 TRABALHOS RELACIONADOS

Esta Seção apresenta uma análise dos trabalhos relacionados disponíveis na literatura, para isso foi feito uma seleção de trabalhos por meio do Google Acadêmico que, através de experimentos controlados, avaliaram a efetividade de linguagens de domínio ou técnicas baseadas em heurísticas para especificar ou identificar anomalias de código. Os trabalhos selecionados são apresentados na Seção 3.1. A análise comparativa e oportunidades de pesquisa são discutidos na Seção 3.2.

3.1 Análise dos Trabalhos Relacionados

[2]: propuseram, através de um estudo experimental, observar a relação entre refatorações de código e *code smells* para apresentar uma métrica de qualidade de código a respeito dos indicativos de *code smells* detectados nos locais refatorados. Como objeto de pesquisa, foram coletados dados de 63 releases de três projetos open source desenvolvidos em Java, analisando manualmente 15.008 operações de refatoração e 5.478 *code smells* e investigando se onde as refatorações de código foram aplicadas havia a presença ou indicativos de anomalias de código. Por fim, foi visualizado que 42% das refatorações foram realizadas em entidades afetadas por anomalias de código, porém somente 7% dos *code smells* foram resolvidos, provando que os desenvolvedores da amostra não tinham como foco a resolução das anomalias de código ao refatorar, utilizando-as majoritariamente como indicativo para reconhecimento de em que local do código uma reescrita deveria acontecer.

[13]: apresentaram um estudo de revisão sistemático terciário, analisando 40 estudos secundários para explorar profundamente o campo de definições e descrições de anomalias de código e quais os meios e ferramentas utilizadas para detectá-las. Conduzindo o estudo, os autores categorizaram como os métodos eram baseados: percepção humana, métricas, estratégia/regras, engenharia reversa, machine learning e visualização. Com essas categorizações, foi possível realizar análises mais profundas para identificar a causa e existência de algumas anomalias de código e concluíram que a criação de *code smells* está diretamente ligada ao nível de experiência do desenvolvedor, ao conhecimento dele da arquitetura do sistema e a questões de insegurança e falta de cuidado. No âmbito das ferramentas utilizadas, foram encontrados o CCFinder, PMD, inCode, DECOR/DETEX e o inFusion e sintetizando uma visão geral sobre

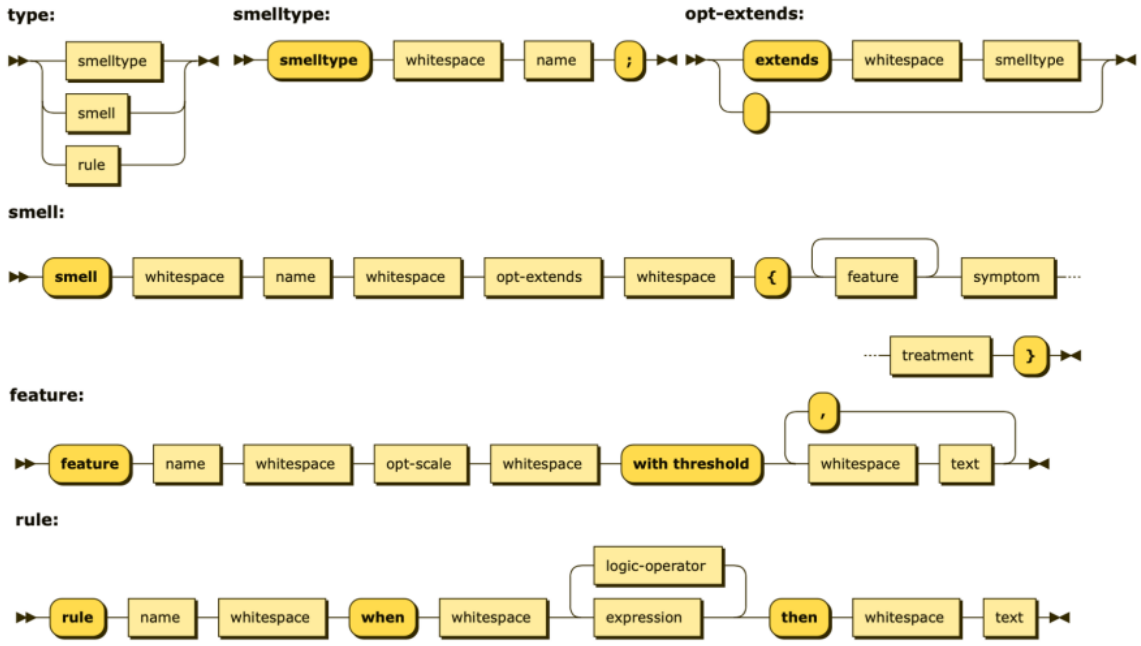

Figure 5: Diagrama sintático - SmellDSL [20]

elas, foi concluído pelos autores que as ferramentas majoritariamente focam em identificar anomalias de código de uma linguagem de programação específica e poucas sugerem possíveis refatorações para o código analisado. Por fim, os autores pontuam que por mais que as ferramentas identifiquem as anomalias, como poucas refatorações são eventualmente sugeridas, não há um grande impacto na qualidade de código.

[11]: investigam através de um experimento controlado como os desenvolvedores priorizam a correção das anomalias de códigos ao possuir apoio de uma ferramenta capaz de sinalizar anomalias de código relacionados diretamente com um problema arquitetural do sistema. A ferramenta utilizada para avaliar este caso, foi a JSpIRIT, um plugin do Eclipse altamente configurável e capaz de sinalizar as anomalias de código mais prioritárias para correção. Os autores queriam comprovar que a utilização de diagramas com alto nível de abstração para enunciar a arquitetura de um software seria benéfica para a identificação e correção de anomalias de código, e citaram que 80% dos *code smells* são relacionados a problemas arquiteturais do software. Ao executar o experimento com desenvolvedores de um sistema com mais de 54 KLOC foi verificado que 71.4% dos pesquisados julgam os diagramas como úteis para identificar e priorizar anomalias de código, porém muitos falsos positivos foram gerados devido a erros humanos, levando os autores a refazer o experimento utilizando o JSpIRIT como ferramenta de automação, melhorando a precisão do resultado esperado, porém concluindo que a ferramenta funciona melhor em sistemas de pequeno a médio porte, tornando-se de difícil utilização em softwares maiores.

[1]: propõem um estudo empírico para classificar a usabilidade de duas linguagens específicas de domínio voltadas a identificação de degradação arquitetural de softwares, descrevendo as dificuldades de adoção de novas linguagens de domínio no desenvolvimento por parte dos desenvolvedores e quantificando métricas enfatizando seus benefícios. Como o estudo dos autores está voltado a investigar linguagens específicas de domínio, independente de serem voltadas para especificação de anomalias de código, as métricas utilizadas pelos autores acabam se desvirtuando um pouco do trabalho proposto, referindo-se assuntos como abstrações, viscosidade, flexibilidade e não há atributos de esforço de especificação e corretude. Porém, é válido notar que os autores concluem que ainda há muito a ser evoluído no campo das DSL e que estudos que utilizam de métricas quantitativas serão de grande ajuda para evoluir o conhecimento acadêmico sobre as falhas e possíveis melhorias de DSLs.

[12]: realizaram um estudo empírico com propósito de avaliar e comparar duas ferramentas de detecção de anomalias de código, a JDeodorant e a inCODE, ambas implementadas como um plugin do Eclipse, assim como a SmellDSL. Após realizarem a análise de ambas ferramentas, concluíram que suas detecções de *code smells* geravam resultados diferentes, isso devido a cada uma ter suas próprias heurísticas e regras para considerar uma parte de código um *code smell* ou não. Por fim, os autores comentam que por mais que ambas ferramentas tenham seus pontos fracos e fortes, reforçam a ideia de que ambas não são maduras o suficiente para o trabalho e que o cenário atual necessita que novas ferramentas sejam criadas e revisadas. Infelizmente este estudo também carece de métricas relacionadas a esforço de especificação, corretude e erro por investigar ferramentas e não diretamente DSLs que auxiliam a especificação de anomalias de código.

3.2 Análise Comparativa e Oportunidades de Pesquisa

A análise comparativa foi realizada com base em critérios, porque outros trabalhos já publicados [3, 5, 18, 19, 24] e que utilizaram esta abordagem, mostraram que esta é uma forma efetiva para se gerar uma comparação entre trabalhos e identificar as oportunidades de pesquisa.

Com a realização da análise comparativa, foi possível identificar pontos que se assemelham e diferenciam entre o trabalho atual e os trabalhos relacionados selecionados. Essa comparação foi necessária para a identificação de oportunidades de pesquisa de forma objetiva, e para isso foram selecionados 7 Critérios de Comparação, que são os descritos a seguir:

- **Experimento Controlado (CC01):** trabalhos que realizaram um experimento controlado.
- **Atributo de Corretude (CC02):** estudos que verificam atributos de corretude.
- **Atributo de Erro (CC03):** estudos que consideram atributos de erro.
- **Atributo de Esforço de Especificação (CC04):** estudos que consideram atributos de esforço de especificação.
- **Uso de Linguagem Específica de Domínio (CC05):** estudos onde a utilização de uma linguagem de domínio fosse parte do tema central da pesquisa.
- **Uso de Ferramenta de Suporte (CC06):** estudos onde a uma ferramenta de suporte fosse utilizada.
- **Contexto da Aplicação (CC07):** estudos em que o contexto do experimento é similar ao contexto do vigente trabalho.

Table 1: Análise comparativa dos trabalhos relacionados.

Trabalho Relacionado	Critério de Comparação						
	CC1	CC2	CC3	CC4	CC5	CC6	CC7
Trabalho proposto	●	●	●	●	●	●	●
[2]	●	○	○	○	○	●	●
[13]	●	○	○	◐	○	◐	●
[11]	●	●	○	◐	○	●	●
[1]	●	○	○	○	●	●	◐
[12]	●	○	○	○	●	●	◐

● Similar ◐ Parcialmente similar ○ Não similar

Fonte: elaborado pelo autor.

O resultado da comparação dos trabalho relacionados e do trabalho proposto com base nos Critérios de Comparação é demonstrado na Tabela 1. Analisando o resultado da comparação, foram identificados os seguintes pontos:

- somente um trabalho avaliou atributos de corretude;
- nenhum trabalho explorou a utilização de atributos de erro para avaliação;
- nenhum estudo relaciona propriamente a utilização de linguagens específicas de domínio com foco na especificação de anomalias de código, ou verificam DSLs isoladas, ou verificam identificação e especificação de forma isolada;
- é sinalizado em diversos trabalhos a importância de estudos empíricos e métricas para avaliar anomalias de códigos;
- diversos trabalhos apontam o carecimento acadêmico de pesquisas que avaliam DSLs e/ou anomalias de código;
- poucos trabalhos avaliam esforço de especificação de anomalias de código e os que apresentaram, acabaram por realizá-los de forma informal, sem métricas concretas.

Oportunidade de Pesquisa. Com base nos pontos sinalizados, foi identificada a oportunidade de acrescer a comunidade acadêmica com um estudo empírico com objetivo de avaliar a efetividade de uma linguagem específica de domínio para especificação de anomalias de código, utilizando a SmellDSL como objeto de pesquisa. Esta oportunidade de pesquisa é explorada nas próximas seções.

4 METODOLOGIA

Esta Seção descreve as decisões e o andamento do estudo experimental e exploratório. A Seção 4.1 apresenta o objetivo e as questões de pesquisa. A Seção 4.2 introduz as hipóteses com base nas questões de pesquisa. A Seção 4.3 apresenta as variáveis quantitativas consideradas. A Seção 4.4 descreve o processo de experimentação. A Seção 4.5 informa como foi feita a seleção dos participantes para o experimento controlado. Por fim, a Seção 4.6 demonstra os procedimentos de análise.

4.1 Objetivo e Questões de Pesquisa

Este estudo essencialmente procura avaliar empiricamente a efetividade de utilização da linguagem de específica de domínio SmellDSL para especificação de anomalias de código, através de um estudo empírico em um espaço amostral de desenvolvedores experientes na área de tecnologia. Após receberem o treinamento necessário para utilização da SmellDSL, os desenvolvedores foram submetidos a testes de especificação de anomalias de código, com a finalidade de coletar dados e realizar métricas capazes de avaliar sua efetividade. O objetivo desse estudo é apresentado e baseado no modelo GQM, tal modelo mostrou-se eficaz na elaboração de objetivos, que pode ser visto nos seguintes trabalhos: [6, 7]. O resultado gerado pelo modelo é apresentado abaixo:

Objetivos:
analisar *a linguagem SmellDSL*
com a finalidade de *investigar seu efeito*
no que diz respeito a *métricas de corretude, taxa de erro e esforço de especificação*
do ponto de vista de *desenvolvedores de software*
no contexto *de especificação de anomalias de software*

Particularmente, este estudo concentra-se em analisar atributos de corretude, erro, e esforço dos desenvolvedores ao utilizar a SmellDSL para especificar anomalias de código. Com o intuito de avaliar a sua efetividade em enunciar as irregularidades de código através de um estudo empírico realizado com 35 desenvolvedores. Métricas de irrefutabilidade foram apresentadas ao conferir atributos de corretude acima ou igual a 50%, atributos de erro abaixo de 50% e esforço de no máximo 15 minutos para resolver cada questão do teste de especificação de *code smells* com 8 questões. Valores escolhidos pelo autor para comprovação. Com isso, o estudo se concentra nas seguintes questões de pesquisa:

- **QP01:** Qual é a taxa de corretude de desenvolvedores especificando anomalias de código utilizando a SmellDSL?

- **QP02:** Qual é a taxa de erro de desenvolvedores especificando anomalias de código utilizando a SmellDSL?
- **QP03:** Quanto de esforço investido, em tempo, os desenvolvedores especificando anomalias de código utilizando a SmellDSL tiveram?

4.2 Formulação da Hipótese

Foram formuladas três hipóteses com objetivo de avaliar métricas de esforço de especificação, taxa de corretude e taxa de erro de desenvolvedores quando submetidos a um teste prático de especificação de irregularidades de códigos utilizando a SmellDSL.

Seguindo o objetivo do estudo, esta avaliação compara os resultados obtidos com esses valores hipotéticos. Para testar as hipóteses formuladas, foi utilizado o método do teste t de uma amostra e as ferramentas Winks e IBM SPSS Statistics foram utilizadas em conjunto para calcular o teste t. Por fim, as hipóteses formuladas ajudaram-nos na tomada de decisões sobre a SmellDSL, através dos dados coletados de cada participante.

[itemsep=0pt,parsep=0pt]**Hipótese Nula 1, H_{1-0}:** Espera-se que os participantes obtenham uma taxa de acerto por tarefa experimental inferior a 0,5 (50%) $\mathbf{H_{1-0}}$**:** Taxa de Corretude (CR) < 50% **Hipótese Alternativa 1, H_{1-1}:** Espera-se que os participantes obtenham uma taxa de acerto por tarefa experimental igual ou superior a 0,5 (50%) $\mathbf{H_{1-1}}$**:** Taxa de Corretude (CR) >= 50% **Hipótese Nula 2, H_{2-0}:** Espera-se que os participantes obtenham uma taxa de erro por tarefa experimental superior a 0,5 (50%) $\mathbf{H_{2-0}}$**:** Taxa de erro (ER) > 50% **Hipótese Alternativa 2, H_{1-1}:** Espera-se que os participantes obtenham uma taxa de erro por tarefa experimental inferior ou igual a 0,5 (50%) $\mathbf{H_{2-1}}$**:** Taxa de erro (ER) <= 50% **Hipótese Nula 3, H_{3-0}:** Espera-se que os participantes realizem cada questão experimental em tempo superior a 15 minutos $\mathbf{H_{3-0}}$**:** Esforço (SE) > 15min **Hipótese Alternativa 3, H_{3-1}:** Espera-se que os participantes realizem cada questão experimental tempo igual ou inferior a 15 minutos $\mathbf{H_{3-1}}$**:** Esforço (SE) <= 15min

A formulação das hipóteses analisadas estão dispostas na Tabela 2 com suas representações de valores verificados. Cada métrica verificada nas hipóteses foram representadas como variáveis dependentes na Seção 4.3.

Ao realizar os testes de hipóteses, estarão sendo produzidos conhecimentos empíricos com a finalidade de avaliar a efetividade da SmellDSL com assertividade, utilizando as métricas propostas em um contexto real de desenvolvimento de software.

Table 2: Hipóteses analisadas

Hipótese Nula		Hipótese Alternativa	
H_{1-0}	Atributos de corretude(CR) < 50%	H_{1-1}	Atributos de corretude(CR) >= 50%
H_{2-0}	Atributos de erro(ER) >= 50%	H_{2-1}	Atributos de erro(ER) < 50%
H_{3-0}	Esforço de especificação (SE) > 15min	H_{3-1}	Atributos de erro (SE) <= 15min

CR = Atributos de corretude; *ER* = Atributos de erro; *SE* = Esforço de especificação; Fonte: elaborado pelo autor.

A SmellDSL foi avaliada através de um estudo inicial. Os valores hipotéticos foram definidos permitindo uma avaliação inicial, partindo do raciocínio de que a DSL para ser considerada minimamente funcional deveria, no mínimo, apresentar taxa de acerto maior ou igual a 50%, taxa de erro menor ou igual a 50%. O valor de 50% não é respaldado por estudos empíricos disponíveis na literatura. A justificativa foi que a DSL deveria apresentar taxa de acerto igual ou superior a 50% nesta primeira avaliação inicial. Estudos empíricos mais robustos serão realizados para verificar uma taxa mais elevada. Hoje a ferramenta não possui restrições de comparação com outras ferramentas. Como próximo passo e trabalho futuro, é planejado comparar a SmellDSL com outras ferramentas existentes.

4.3 Variáveis do Estudo e Método de Quantificação

Variável independente: A variável independente da hipótese formulada é a linguagem específica de domínio estudada: "*SmellDSL*". Este será (ver Seção 4.4) submetido à testes para a coleta de dados através das variáveis dependentes definidas abaixo.

Variáveis dependentes: Três variáveis dependentes foram geradas para efetivar a análise das hipóteses propostas ao aplicar aos participantes o estudo de oito tarefas experimentais. Cada uma com um pedido de especificação de anomalia de código, que deve ser especificado utilizando a SmellDSL.

- **Taxa de corretude (CR)**: a especificação da SmellDSL é considerada correta caso esteja em conformidade com os requisitos solicitados. A taxa de corretude representa a média das especificações corretas das regras de cada tarefa, assumindo valores de 0 (zero) a 1 (um), em que valores próximos de zero significam uma taxa de corretude baixa e valores próximos de um representam uma taxa de corretude elevada.
- **Taxa de erro (ER)**: esta métrica busca medir a taxa de erro ao especificar uma anomalia de código. Será considerado erro qualquer não conformidade da tarefa realizada com os requisitos solicitados. Assume valores de 0 (zero) a 1 (um), em que zero significa baixa taxa de erro e um significa alta taxa de erro.
- **Esforço de especificação (SE)**: esta métrica busca medir a taxa de erro ao especificar uma anomalia de código, medindo o esforço necessário (em tempo) para especificar uma anomalia utilizando a SmellDSL para cada tarefa proposta. O estudo quantifica a métrica considerando o esforço investido para completar cada questão.

4.4 Processo Experimental

O Processo experimental deste estudo é dividido em cinco etapas e organizado em três fases: treinamento e familiaridade, execução do teste experimental e avaliação qualitativa e quantitativa. Todos os participantes realizaram o teste experimental individualmente e todos os dados referentes a suas execuções foram coletados. A Figura 6 apresenta o processo experimental utilizado no experimento.

- **Treinamento e familiaridade**: Essa é a primeira fase do estudo e tem como objetivo treinar os participantes para que eles se familiarizem com a SmellDSL. Para isso, todos os participantes receberam o devido treinamento sintático e contextual, além de informações técnicas para realizar uma especificação de anomalia de código. Um estudo inicial

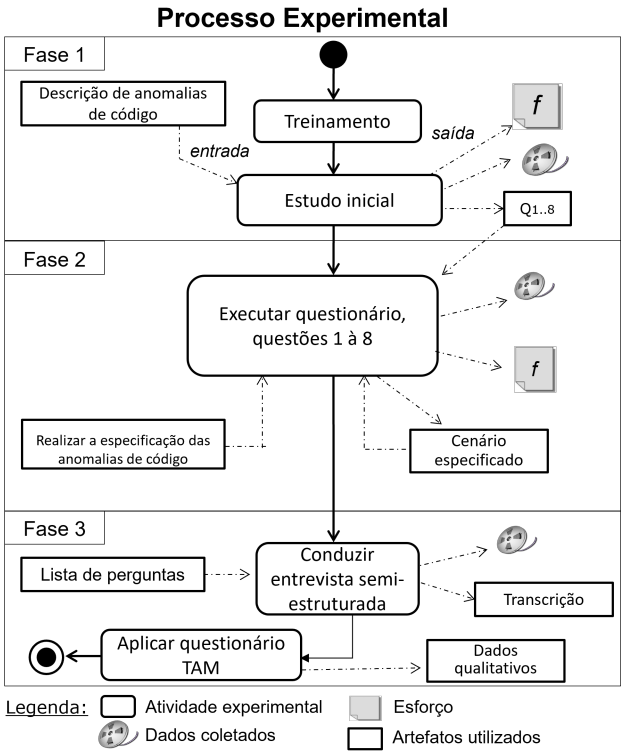

Figure 6: Processo experimental

foi executado previamente ao estudo real ter início para garantir que o treinamento proposto cumpre seu objetivo.

- **Execução do teste experimental**: Essa fase tem foco em executar o teste experimental com os participantes. Para isso, foi elaborado uma série de 8 questões contendo requisitos das anomalias que precisam ser especificadas utilizando a SmellDSL. Os dados dos exercícios realizados, assim como uma gravação da tela do participante foram coletados ao final do teste.
- **Avaliação qualitativa e quantitativa**: A última fase do processo é a coleta de dados quantitativos e qualitativos dos participantes. Para isso, foi executado uma análise estatística e testes de inferência nos dados coletados após os participantes finalizarem a execução das tarefas experimentais. Por fim, foi realizada uma entrevista semi estruturada com cada um dos participantes e aplicado um questionário TAM (Technology Acceptance Model) que é amplamente utilizado no meio acadêmico para avaliar aceitação de um objeto de estudo segundo [10]

Ao total, 8 cenários foram especificados para cada participante, totalizando em 280 casos avaliados utilizando a SmellDSL. Por exemplo, o cenário 1 apresenta uma tarefa descrevendo que o participante deve especificar um *Smelltype Bloater* e um *Smell LargeClass* do tipo *Bloater*. Nas tarefas 1-5 os participantes realizaram o exercício do zero, escrevendo todo processo de especificação, enquanto nos exercícios 6-8 foram realizados manutenção em especificações existentes.

Table 3: Questões experimentais

	Smelltype			Smell									
Questão	*Bloaters*	Dispensables	*Couplers*	LargeClass	LongParameterList	DuplicateCode	LongMethod	FeatureEnvy	Features	Symptom	Treatment	Rule	Maintenance
Q01	●	○	○	●	○	○	○	○	●	○	○	○	○
Q02	●	○	○	○	●	○	●	○	●	○	○	○	○
Q03	○	●	○	○	○	●	○	○	●	●	●	●	○
Q04	○	●	○	○	○	○	○	●	●	●	●	●	○
Q05	○	○	●	○	○	○	○	●	●	●	●	●	○
Q06	○	○	●	○	○	○	○	●	●	●	●	●	●
Q07	●	●	○	●	●	○	○	●	●	●	●	●	●
Q08	●	●	○	●	●	○	○	●	●	●	●	●	●

Legenda: (●) *Suporta* (○) *Não Suporta*

O experimento proposto abrangeu a definição de alguns tipos de *smells* e *Smelltypes*. Cada questão foi responsável por conter a especificação de um *smell* ou *Smelltype* específico, e testar conceitos de especificação utilizando as estruturas implementadas pelas SmellDSL, conforme a Tabela 3. Optou-se por iniciar os dois primeiros exercícios testando apenas as estruturas de *smell*, *Smelltype* e *feature*, para facilitar a aproximação do participante com a linguagem, incluindo especificações utilizando rules, *treatment* e sintomas a partir do exercício três em diante. Por exemplo, no exercício um, foi pedido que os participantes descrevessem um *Smelltype Bloaters* e um *smell* chamado Large Class que possui a *feature* ClassNumber, scaletype nominal e com threshold de HIGH, MEDIUM e LOW. Já o exercício cinco, último exercício de criação de especificações que não abrangeu manutenções de código, apresentou um problema muito mais complexo, pedindo aos participantes para especificar o *Smelltype Couplers*, uma regra condicional utilizando a estrutura *rule*, e um *smell FeatureEnvy*, que possui um *symptom* e um *treatment* além de quatro *features*, sendo elas: NumVar, NumMeth, NumVarNoUsed, NumMethNoUsed, referindo-se a quantidade de variáveis, quantidade de métodos, quantidade de variáveis não utilizadas e quantidade de métodos não utilizados, respectivamente. Esse exemplo serve para ilustrar a escala de dificuldade de especificação dos exercícios propostos no teste experimental. Observando a Seção de resultados, é possível verificar que por mais que a curva de dificuldade seja crescente entre os exercícios, alguns participantes apresentaram mais tempo de esforço de especificação nos exercícios iniciais, devido a estarem se familiarizando e realizando tarefas mais complexas, como o exercício cinco, de forma mais simples e direta já que haviam se acostumado com a ferramenta SmellDSL.

4.5 Contexto e seleção de participantes

A avaliação foi realizada com 35 participantes, sendo eles quatro estudantes e 31 desenvolvedores de software de diversas empresas Brasileiras. Foi enviado um e-mail para alunos da Universidade do Vale do Rio dos Sinos e do Instituto Federal do Mato Grosso (IFMT), selecionando aqueles com experiencia em desenvolvimento e modelagem de software. Alguns participantes já completaram a graduação e outros já completaram ou estão cursando o mestrado. Optou-se por incluir estudantes além de desenvolvedores profissionais para verificar como pessoas de diferentes níveis de escolaridade e experiência lidaram com o treinamento e teste experimental,

visando coletar dados de perfis heterogêneos. A condução do experimento foi similar a um teste de laboratório. Cada participante recebeu o mesmo nível de treinamento e teve o mesmo acesso as questões.

4.6 Procedimento de Análise

Análise descritiva. Foi realizada a análise descritiva para analisar a distribuição, dispersões, tendências como médias e medianas dos dados coletados de cada métricas selecionada para cada tipo de arquitetura definida pela variável independente.

Inferência estatística. Foi realizada a análise estatística para o teste de hipóteses. O nível de significância para os testes das hipóteses foi = 0,05. As análises estatística foram realizadas para testar as hipóteses de cada métrica selecionada e em cada cenário de teste executado. Para testar as hipóteses H1, H2 e H3 foi aplicado o Teste t-student. Como resultado, cada métrica foi analisada e comparada para verificar se a hipótese nula foi rejeitada e a hipótese alternativa aceita.

Para avaliar as hipóteses com relação aos dados coletados foi utilizado o **WINKS (*IBM SPSS Statistics*)** [2]. Nos cenários propostos cada desenvolvedor teve um desempenho diferente, isso deve-se a diversos fatores, experiencia com programação, entendimento da SmellDSL e outros diversos fatores. Um participante pode ter um desempenho pior devido ao seu tempo de execução e capacidade de abstração dos cenários propostos. Porém, na maioria dos cenários o tempo foi calibrado para ter uma média de 15 minutos para sua codificação em geral, tempo este que se mostrou bem regulado, conforme a Seção 5 apresenta.

5 RESULTADOS

Esta Seção apresenta os resultados obtidos após analisar os dados obtidos pelo processo experimental descrito na Seção 4. As descobertas são derivadas de processamentos numéricos dos dados coletados e representações gráficas dos aspectos dos resultados obtidos. A Seção 5.1 apresenta informações a respeito dos dados de perfil dos participantes entrevistados. A Seção 5.2 apresenta os dados obtidos através dos testes de hipóteses, discorrendo informações a respeito dos resultados obtidos para cada questão proposta através dos procedimentos de análise informados na Seção 4.6. As Seções 5.3, 5.4 e 5.5 apresentam a análise descritiva e inferências estatísticas realizadas para responder as questões QP01, QP02 e QP03, respectivamente. A Figura 7 apresenta valores de taxa de corretude e taxa de erro nos oito cenários especificados pelos participantes. A Figura 8 apresenta os valores da taxa de erro dos participantes por questão experimental. A Figura 9 apresenta os esforços investidos por cenário, discriminando cada cenário individualmente.

5.1 Análise do Perfil dos Participantes

A Tabela 4 descreve os dados informativos a respeito do perfil dos participantes entrevistados. Nela é possível verificar que a maioria (45.7%) dos participantes está na faixa de 18 a 25 anos, 31.4% estuda ou estudou Sistemas da Informação e 42.9% estão graduados. O interessante sobre estes dados é verificar a heterogeneidade dos perfis dos participantes entrevistados, e que por mais que a maioria (65.7%) sejam programadores, quase todos apresentaram contextos de experiência diferentes. Conforme apresentado na Seção 5.2, nota-se que os participantes que possuem *background* similar, acabaram por descrever opiniões muito parecidas durante a execução da entrevista semi-estruturada. Por fim, 34.3% dos participantes possuem mais de 8 anos de experiência na área de tecnologia, 37.1% possuem de 5 a 6 anos modelando *software* e 28.6% possuem de 5 a 6 anos de experiência desenvolvendo *software*.

Table 4: Perfil dos participantes

Características	Descrição	%
Idade	18 - 25 anos	45.7%
	26 - 35 anos	31.4%
	36 - 45 anos	20%
	> 45 anos	2.9%
Educação acadêmica	Engenharia da Computação	5.7%
	Sistemas da Informação	31.4%
	Análise e desenvolvimento de sistemas	14.3%
	Ciência da Computação	25.7%
	Outros	22.9%
Educação	Técnico	17.1%
	Graduação	42.9%
	Mestrado	5.7%
	Outros	34.3%
Cargo atual	Programador	65.7%
	Analista	11.4%
	Gerente	11.4%
	Outros	11,5%
Tempo de experiência	< 2 anos	20%
	2 - 4 anos	20%
	5 - 6 anos	25.7%
	> 8 anos	34.3%
Tempo de experiência modelando software	< 2 anos	17.1%
	< 2 anos	31.4%
	2 - 4 anos	17.1%
	5 - 6 anos	37.1%
	7 - 8 anos	2.9%
	> 8 anos	11.4%
Desenvolvimento de software	< 2 anos	17.1%
	2 - 4 anos	17.1%
	5 - 6 anos	28.6%
	7 - 8 anos	20%
	> 8 anos	17.1%

5.2 Teste de Hipóteses

Foram realizados testes estatísticos com os dados coletados através das métricas de performance para avaliar se os mesmos são estatisticamente significantes. Para isso, foram estipulados valores hipotéticos, escolhidos pelo autor, definindo uma avaliação inicial dos dados se baseando na justificativa de que uma DSL, para ser considerada minimamente utilizável, deve ao menos atingir uma taxa de corretude igual ou maior a 50%, uma taxa de erro menor ou igual a 50% e que o tempo de esforço mínimo requirido para especificar uma anomalia de código seja menor ou igual a 15 minutos, conforme mencionado anteriormente. Estes valores hipotéticos para realizar a avaliação de uma linguagem específica de domínio não foram encontrados em nenhum estudo na comunidade acadêmica, com isso em mente, este estudo busca enriquecer a acadêmia providenciando análises concretas a respeito de uma DSL e visando abrir as portas para que mais estudos desse tipo sejam realizados.

A Tabela 5 apresenta os resultados para as hipóteses formuladas. Os valores das médias de corretude, erro e esforço indicam que todas hipóteses nulas do trabalho foram rejeitadas, conferindo que todas hipóteses alternativas foram comprovadas.

A Tabela 6 apresenta os resultados das médias de corretude, erro e esforço de especificação para cada uma das questões experimentais propostas para os participantes.

[2] https://www.alanelliott.com/TEXASOFT/

Table 5: Resultados das hipóteses testadas.

Hipótese	Valor Hipotético	Qtd Questões	Média	Desvio Padrão	t	Grau de Liberdade (DF)	p-value
H1 (QP1)	50	8	98.20	1.0195	133.74	7	**0.015**
H2 (QP2)	50	8	1.78	1.0204	-133.65	7	**0.015**
H3 (QP3)	15	8	5.0325	1.3661	-20.71	7	**0.001**

Table 6: Média dos resultados por questão experimental

Questões experimentais	Taxa de corretude	Taxa de erro	Esforço de Especificação
Questão 1	96.1905	3.80952	4.05
Questão 2	98.0952	1.90476	4.42
Questão 3	99.4286	0.57143	7.07
Questão 4	97.7143	2.28571	5.4
Questão 5	98.2857	1.71429	7.11
Questão 6	98.2857	1.71429	4.11
Questão 7	99.4286	0.57143	3.6
Questão 8	98.2857	1.71429	4.4

Considerando que as hipóteses possuem o objetivo de responder as questões propostas na Seção 4, optou-se por estruturar essa Seção de forma a introduzir os resultados das hipóteses testadas nas tabelas acima e abrir caminho para que as próximas seções apresentem o resultado para cada hipótese de forma aprofundada.

5.3 QP01: SmellDSL e Taxa de Corretude

Após coletar os resultados a respeito da taxa de corretude para responder a questão QP01 e aplicar a análise descritiva na amostra dos resultados da tabela 5, foi apresentado uma média de taxa de corretude de 98.20 (98.20%) e um desvio padrão de 1.0195. O valor hipotético para a aprovar a hipótese é de 50 (50%). A análise descritiva inclui a aplicação de um teste t, com um valor crítico de 133.74 e 7 graus de liberdade.

O resultado obtido após a execução do teste de t-student informa que a taxa de corretude [Média = 98.20876, DP = 1.019515] foi estatisticamente significativa a um nível de significância de 0.05 (t = 133,74, df = 7, p = 0.015) para o valor de teste igual a 0.5, com uma diferença média de 0.015 e um intervalo de confiança de 95%. A hipótese nula que sugeria uma taxa de corretude inferior a 50 pôde ser rejeitada. Por conseguinte, o resultado do teste indica que a taxa de corretude média foi significativamente superior a 50. Com base nos resultados, a hipótese nula de que a taxa de corretude seria inferior a 50 pôde ser rejeitada.

A Figura 7 ilustra de forma gráfica a taxa de acertos dos participantes nas questões propostas, e a Tabela 6 ilustra as médias de corretude para cada questão exeperimental. Com estes dados, é possível verificar que a taxa de corretude é muito próxima a 100% (98.2%) em todas as questões, tendo a questão 3 e a questão 7 como as mais acertadas pelos participantes (99.43%).

5.4 QP02: SmellDSL e Taxa de Erro

Verificando as 280 respostas geradas pelos participantes e aplicando a análise descritiva da taxa de erro, seguindo a Tabela 5, é possível uma média amostral de 1.78 (1.78%) e um desvio padrão de 1.0204. O valor hipotético para a taxa de erro é 50. Ao aplicarmos um teste t com 7 graus de liberdade e um valor crítico de -133.74 (t= 133.74), pode-se avaliar a significância estatística da discrepância entre a média amostral e o valor hipotético.

O cálculo do estatístico de teste t segue a fórmula padrão, resultando em um valor específico. A comparação entre esse valor calculado e o valor crítico, determinará se é possível rejeitar a hipótese nula de que a taxa de erro de especificação é superior ao valor hipotético. O nível de significância p estabelecido para esta análise é de 0.015. sinalizando que a hipótese nula foi rejeitada, indicando uma diferença estatisticamente significativa na taxa de erro.

O resultado do teste t-student também mostra que a taxa de erro [Média = 1.78, DP = 1.020448] foi estatisticamente significativa a um nível de 0.05 de significância (t = -133.74, df = 7, p = 0.015) para o valor de teste igual a 50, com uma diferença média de 0.015 e Intervalo de Confiança de 95%. A hipótese nula que sugeria uma taxa de erro superior a 50 (50%) pode ser rejeitada.

A Figura 7, em conjunto com a Tabela 6 apresentam o percentual de erro dos participantes em cada cenário. Com elas, é possível notar que a maioria dos erros está presente nas primeiras questões (questão 1 e questão 2), fato este, que será discorrido mais aprofundadamente na Seção 5.6. Além desses dados, para visualizar os dados de taxa de erro de forma gráfica, pode-se visualizar a Figura 8.

5.5 QP03: SmellDSL e Esforço de especificação

A média do tempo gasto pelos participantes foi de 5.0325 (aproximadamente 5 minutos), com um desvio padrão de 1.3661. Esses dados sugerem que, em média, os participantes conseguiram completar as questões dentro do limite estabelecido de 15 minutos. O desvio padrão fornece uma medida da variabilidade nos tempos de realização, indicando quão dispersos estão os resultados entre os participantes.

Para uma análise estatística mais aprofundada, foi realizado um teste t com um valor crítico de 20.71 (t= 20.71), considerando 7 graus de liberdade e um nível de significância de 0.001. O valor calculado de t foi comparado com o valor crítico para determinar se a diferença entre a média amostral e o valor hipotético de 15 minutos é estatisticamente significativa.

O resultado do t-test também apresentou que a média do esforço [Média = 5.0325, DP = 1.3661] foi estatisticamente significativa a um nível de significância de 0.05 (t = -20.71, df = 7, p < 0.001) para o valor de teste igual a 15, com uma diferença média de 0.001 e intervalo de confiança de 95%. A hipótese nula que sugere um esforço de especificação igual ou superior a 15 minutos pôde ser rejeitada.

A Figura 9, em conjunto com a Tabela 6 ilustram que o esforço médio, em minutos, para cada questão não foi superior a 7 minutos, sinalizando a questão 3 (7.09 minutos) e a questão 5 (7.11 minutos) como as mais demoradas para serem realizadas.

5.6 Discussão de dados empíricos

Baseado nas observações dos resultados obtidos pela entrevista semi-estruturada, foram coletados dados empíricos de cada participante a respeito da sua opinião com o teste realizado e a linguagem SmellDSL.

Questões experimentais: Conforme apresentado anteriormente, foram desenvolvidas oito questões experimentais, com uma curva de dificuldade ascendente, abrangendo conceitos de anomalias de

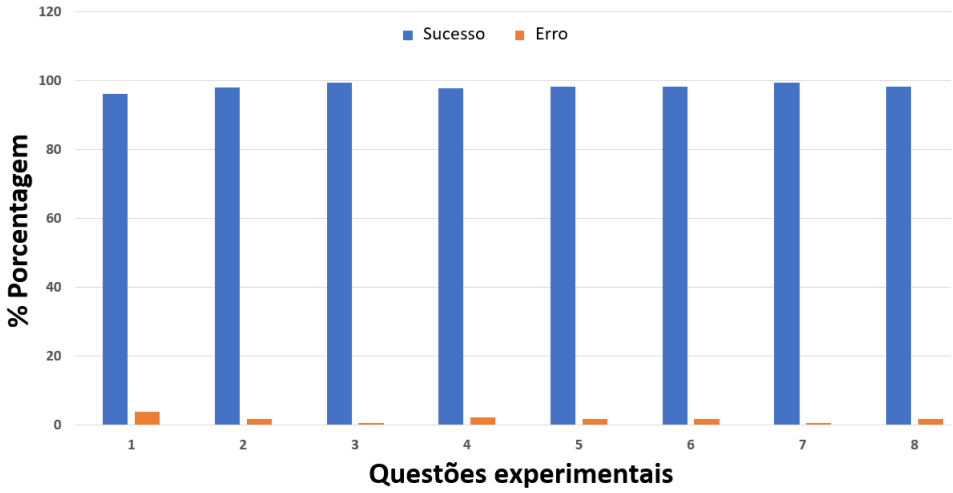

Figure 7: Taxa de corretude (QP1) e taxa de erro (QP2)

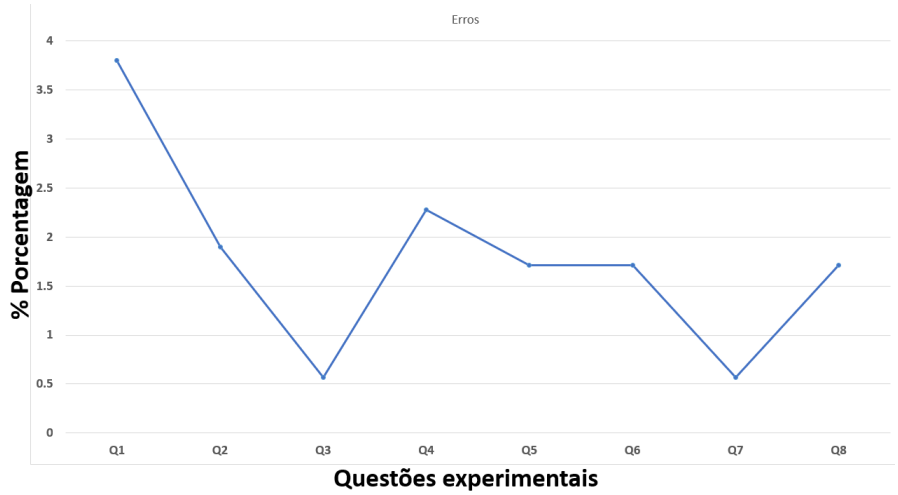

Figure 8: Taxa de erro por questão (QP2)

código e específicos da SmellDSL. Após coletar dados através de uma entrevista semi-estruturada e da análise da gravação de tela dos participantes, 65% responderam que acharam os exercícios massantes e que a descrição deles deveria ter sido feita de forma enunciada e descritiva, e não apresentado em uma forma de tabela, pois isso fez com que eles se tornassem repetitivo e não demonstrassem muito desafio no processo de pensamento para interpretar os problemas a serem especificados de forma que seria feito normalmente em um ambiente de especificação real.

Estrutura da SmellDSL: A respeito da sintaxe e estrutura da SmellDSL, foram coletados dados da opinião dos participantes ao especificar anomalias de código utilizando a ferramenta. 77% dos participantes informaram que não encontraram dificuldades ao especificar anomalias utilizando a SmellDSL, sinalizando o quão fácil é a utilização da ferramenta, porém 70% acharam a escrita da ferramenta muito verbosa e repetitiva, comentando que o código rapidamente se torna muito longo. 40% sinalizaram que a utilização da palavra *extends* para definir o *Smelltype* de um *smell* da a entender que é uma importação de *Smelltype* e não uma definição que precise ser colocada novamente em cada exercício. Esse ponto do *Smelltype* foi interessante de avaliar, pois a maioria dos erros coletados foram feitos durante a execução dos exercícios iniciais devido ao fato dos participantes acharem que o *Smelltype* funcionava como uma importação e não que precisava ser definido. Além desses fatores, 30% dos participantes acharam a sintaxe muito descritiva, sugerindo que a palavra threshold não precisasse ser usada na definição de valores de *features* e sugerindo que houvesse uma padronização de *case* para as especificações. 75% dos participantes apontaram que não há necessidade de repetir o nome da *feature* após inserir o operador de comparação ao definir condições na estrutura rules. Algumas opiniões isoladas também foram comentadas, como: permitir mais de um tratamento para os *smells*, seguir uma abordagem

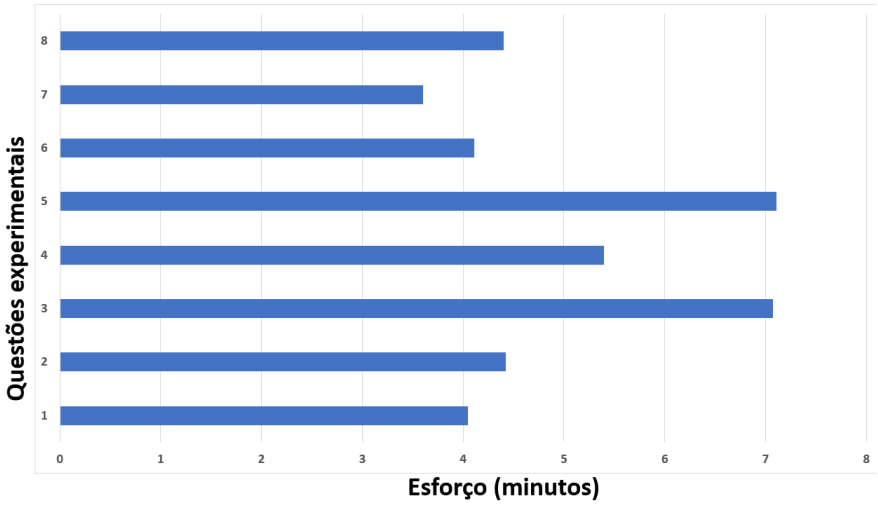

Figure 9: Esforço de especificação (QP3)

BDD ao invés da definição similar a uma orientação a objetos ou então seguir uma abordagem mais próxima a programação e não um meio termo. Um fato interessante de se observar é que as opiniões a respeito dos pontos citados acima geralmente foram apresentados por participantes cujo o *background* técnico era similar.

Analisando os dados coletados de forma empírica, nota-se que as questões propostas poderiam representar de forma mais clara um processo de pensamento para análise e especificação de anomalias seguindo o mundo real. Que provavelmente, se descritas de forma similar a um enunciado, gere mais dúvidas e faça o participante ir além no teste.

Além disso, ao analisar que a taxa de erro foi maior na primeira e segunda questão experimental, nota-se que por mais que as primeiras questões sejam as mais simples, muitos participantes acabaram se confundindo e cometendo erros por ser o seu primeiro contato com a SmellDSL. Correlacionando os dados da Figura 7 e da Figura 9 com os dados empíricos coletados, nota-se que o maior tempo de especificação foram justamente na questão três e cinco, devido a questão três ser a primeira a conter especificação de regras (*rules*) e a questão cinco conter regras complexas. Sinalizando que a forma que a SmellDSL especifica regras pode ser custosa e tomar um tempo considerável.

Ao coletar o feedback de muitos participantes a respeito da SmellDSL também nota-se que por mais que a facilidade de uso seja o seu ponto forte, ainda há bastante campo para evolução e reflexão de quais caminhos seguir para o futuro. Os dados empíricos coletados também servem para que a comunidade analise e desenvolva novos estudos, visto que não há nenhum estudo avaliando similar ao apresentado com foco em analisar linguagens específicas de domínio para realização de especificação de anomalias de código.

Este tipo de discussão, a respeito de dados coletados empiricamente, serve como forma de direcionamento para a perspectiva dos próximos estudos. Direcionando o autor e leitores a conduzir um experimento de forma a abranger lacunas deixadas pelo vigente trabalho em trabalhos futuros.

5.7 Limitações do estudo

O estudo experimental relatado é um estudo inicial que explora uma área ainda pouco investigada na literatura. Neste sentido, o estudo possui algumas limitações que precisam ser consideradas.

- Apenas 08 cenários foram considerados no estudo.
- Apenas 35 participantes implementaram os cenários propostos e nem todos eles têm experiência com qualidade de código, ou especificação de anomalias de código utilizando uma linguagem específica de domínio.
- Conforme argumentado no estudo, foi identificado que a maior proliferação de erros está concentrada nos cenários que possuíam uma elevada complexidade na especificação de anomalias de código. Sendo este um dos motivos pelos quais o estudo explora apenas a análise de especificações através da SmellDSL.
- A avaliação de outros modelos de DSL, além de ser inviável para o contexto de análise atual, poderia prejudicar a avaliação dos resultados. Essa dificuldade pode ser entendida pela grande diferença de conhecimento de programação entre os participantes, por se tratar de uma linguagem de especificação nova.
- Não existe uma base de dados explorada por outros trabalhos que permita a especificações de anomalias de códigos e suas possíveis variações para a construção de um fluxo de desenvolvimento e especificação para os cenários propostos. Para isso foi proposto cenários de implementação e manutenção de anomalias de código escolhidas pelo autor.

A ferramenta SmellDSL é 100% funcional e foi desenvolvida como um plugin da plataforma Eclipse, porém, também foram identificadas três limitações referentes a linguagem em si:

- Permite especificar as anomalias de código, mas não gera código para automatizar sua quantificação;
- Integra-se apenas com a plataforma Eclipse, com ideia de ser suportado em outras IDEs no futuro, como o VSCode, SpringTool Suite, entre outras;

- Possui uma gramática bem definida utilizando BNF, mas não há documentação da linguagem e nem exemplos, exceto os criados pelo autor na etapa de treinamento, para auxiliar os desenvolvedores;

6 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho apresentou um estudo empírico para avaliar a efetividade de uma linguagem específica de domínio através de métricas de corretude, erro e esforço de especificação. O objeto do estudo foi a linguagem SmellDSL e durante o experimento foram entrevistados 35 participantes e coletados dados empíricos por meio de um experimento controlado, além das métricas citadas acima a respeito da linguagem.

A partir dos resultados dos dados obtidos neste estudo, observou-se que a SmellDSL possui 98% de taxa de corretude, 2% de erro e média de 5 minutos de esforço de especificação para resolução das questões experimentais, comprovando todas hipóteses propostas pelo trabalho. Através desses resultados, este estudo contribui com conhecimentos empíricos para auxiliar a condução de testes relacionados a efetividade de especificação de anomalias de código utilizando linguagens específicas de domínio, auxiliando a comunidade científica, pois não há estudos relacionando os temas.

Para que este trabalho tenha uma abordagem mais completa, alguns trabalhos futuros foram mapeados. A primeira perspectiva futura, é considerar novas métricas para avaliar outros aspectos relacionados a especificação de anomalias de código. Como não há uma estrutura definida para conduzir um teste de efetividade de especificação de anomalias de código, o autor propôs métricas empíricas que visam justificar a efetividade através de corretude, erro e esforço, porém novas métricas podem ser propostas para avaliar pontos não abrangidos pelo trabalho.

Um estudo pode ser realizado seguindo as mesmas métricas estudadas no vigente trabalho, porém limitando-se somente a participantes que já possuam uma grande experiência profissional com anomalias de código e suas técnicas de identificação. Dessa forma é possível homogeneizar o perfil dos participantes, fazendo com que as respostas coletadas sejam fornecidas por especialistas no ramo e direcionando as respostas, pois conforme os dados coletados na Seção 5.6, as respostas dadas por participantes tendem a ser mais direcionadas quando possuem experiências em comum no seu currículo.

Outra perspectiva a se considerar é realizar uma nova análise após uma versão futura da SmellDSL ser implementada, já que o *feedback* empírico dos participantes foi coletado e pode ser utilizado como base para aprimorar a ferramenta, norteando o autor Robson Keemps a criar um *roadmap* de evolução para a ferramenta.

Por fim, um estudo futuro que definitivamente enriqueceria a comunidade acadêmica seria coletar dados de outras ferramentas que realizam especificação de anomalias de código e comparar seu conceito com a SmellDSL, propondo um novo estudo entre os participantes para coletar estes dados. Ou até mesmo, criar hipóteses de comparação para avaliar a identificação de anomalias de código baseadas em heurísticas e técnicas baseadas em especificações, sinalizando os pontos fortes e fracos de cada uma e suas diferenças em ambientes acadêmicos e profissionais.

Este trabalho é um estudo inicial de métricas de utilização de uma linguagem específica de domínio para especificar anomalias de código, servindo de ponto de partida e suporte para novos estudos mais aprofundados relacionados ao tema.

REFERENCES

[1] Diego Albuquerque, Bruno Cafeo, Alessandro Garcia, Simone Barbosa, Silvia Abrahão, and António Ribeiro. 2015. Quantifying usability of domain-specific languages: An empirical study on software maintenance. *Journal of Systems and Software* 101 (2015), 245–259.

[2] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* 107 (2015), 1–14.

[3] Hebert Cabane and Kleinner Farias. 2024. On the impact of event-driven architecture on performance: An exploratory study. *Future Generation Computer Systems* 153 (2024), 52–69.

[4] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.

[5] Leandro Ferreira D'Avila, Kleinner Farias, and Jorge Luis Victória Barbosa. 2020. Effects of contextual information on maintenance effort: a controlled experiment. *Journal of Systems and Software* 159 (2020), 110443.

[6] Kleinner Farias, Alessandro Garcia, and Carlos Lucena. 2013. Effects of stability on model composition effort: an exploratory study. *Softw Syst Model (2014) 13:1473–1494* (Jan. 2013), 13:1473—-1494. https://doi.org/10.1007/s10270-012-0308-2

[7] Kleinner Farias, Alessandro Garcia, Jon Whittle, Christina von Flach Garcia Chavez, and Carlos Lucena. 2014. Evaluating the effort of composing design models: a controlled experiment. *Softw Syst Model (2015) 14:1349–1365* (May 2014), 14:1349—-1365. https://doi.org/10.1007/978-3-642-33666-9_43

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. 2012. *Refactoring: Improving the Design of Existing Code.* Pearson Education. https://books.google.com.br/books?id=HmrDHwgkbPsC

[9] Giuliana Gabrielli. 2023. *Aprofundando em "Code Smells" e refatoração de código.* https://lab.vortx.com.br/tipos-de-code-smells-e-refactoring/#:~:text=Identificar%20e%20corrigir%20code%20smells,de%20entender%2C%20manter%20e%20evoluir.

[10] Andrina Granić and Nikola Marangunić. 2019. Technology acceptance model in educational context: A systematic literature review. *British Journal of Educational Technology* 50, 5 (2019), 2572–2593.

[11] Everton Guimaraes, S Vidal, A Garcia, JA Diaz Pace, and C Marcos. 2018. Exploring architecture blueprints for prioritizing critical code anomalies: Experiences and tool support. *Software: Practice and Experience* 48, 5 (2018), 1077–1106.

[12] Almas Hamid, Muhammad Ilyas, Muhammad Hummayun, and Asad Nawaz. 2013. A comparative study on code smell detection tools. *International Journal of Advanced Science and Technology* 60 (2013), 25–32.

[13] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software* 167 (2020), 110610.

[14] Benoît Langlois, Consuela-Elena Jitia, and Eric Jouenne. 2007. DSL classification. In *OOPSLA 7th workshop on domain specific modeling.*

[15] Isela Macia, Roberta Arcoverde, Alessandro Garcia, Christina Chavez, and Arndt Von Staa. 2012. On the relevance of code anomalies for identifying architecture degradation symptoms. In *2012 16Th european conference on software maintenance and reengineering.* IEEE, 277–286.

[16] Anderson Oliveira, Willian Oizumi, Leonardo Sousa, Wesley KG Assunção, Alessandro Garcia, Carlos Lucena, and Diego Cedrim. 2022. Smell Patterns as Indicators of Design Degradation: Do Developers Agree?. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering.* 311–320.

[17] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. 2017. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development* 5, 1 (2017), 1–28.

[18] Maluane Rubert and Kleinner Farias. 2021. On the Effects of Continuous Delivery on Code Quality: A Case Study in Industry. *Computer Standards and Interfaces* (2021), 103588.

[19] Matheus Segalotto, Willian Bolzan, and Kleinner Farias. 2023. Effects of Modularization on Developers' Cognitive Effort in Code Comprehension Tasks: A Controlled Experiment. In *XXXVII Brazilian Symposium on Software Engineering.* 206–215.

[20] Robson Keemps da Silva. 2022. SmellGuru: a machine learning-based approach to predict design problems.

[21] Stefan Slinger. 2005. Code smell detection in eclipse. *Delft University of Technology* (2005).

[22] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. 2014. *Refactoring for software design smells: managing technical debt.* Morgan Kaufmann.

[23] Anderson Uchôa, Caio Barbosa, Willian Oizumi, Publio Blenílio, Rafael Lima, Alessandro Garcia, and Carla Bezerra. 2020. How does modern code review impact software design degradation? an in-depth empirical study. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 511–522.

[24] Roger Denis Vieira and Kleinner Farias. 2020. Usage of Psychophysiological Data as an Improvement in the Context of Software Engineering: A Systematic Mapping Study. *SBSI'20: XVI Brazilian Symposium on Information Systems* (Nov. 2020), 1–8. https://doi.org/10.1145/3411564.3411580