Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Histrategy: uma técnica para a customização guiada de estratégias para a detecção de *bad smells*

Mário Hozano Lucas de Souza

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Evandro de Barros Costa
(Orientador)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

S729h Souza, Mário Hozano Lucas de.

Histrategy: uma técnica para a customização guiada de estratégias para a detecção de *bad smells* / Mário Hozano Lucas de Souza.– Campina Grande, 2017.

110 f. : il. color.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2017.

"Orientação: Prof. Dr. Evandro de Barros Costa".

Referências.

1. Qualidade de Software. 2. Anomalias de Código – *Bad Smells*. 3. Compreensão e Manutenção de Programas. I. Costa, Evandro de Barros. II. Título.

CDU 004.416.6(043)

STRATEGY: UMA TÉCNICA PARA A CUSTOMIZAÇÃO GUIADA DE ESTRATÉGIAS PARA A DETECÇÃO DE BAD SMELLS"

MARIO HOZANO LUCAS DE SOUZA

TESE APROVADA EM 02/06/2017

EVANDRO DE BARROS COSTA, Dr., UFAL
Orientador(a)

ROHIT GHEYI, Dr., UFCG
Examinador(a)

TIAGO LIMA MASSONI, Dr., UFCG
Examinador(a)

ALESSANDRO FABRICIO GARCIA, Dr., PUC-RIO
Examinador(a)

PATRICK HENRIQUE DA SILVA BRITO, Dr., UFAL
Examinador(a)

BALDOINO FONSECA DOS SANTOS NETO, Dr., UFAL
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Anomalias de código conhecidas como *bad smells* indicam estruturas em código que podem prejudicar a compreensão e manutenção de programas. A ausência de uma definição clara para os *bad smells* contribui para que diferentes interpretações sejam consideradas, fazendo com que desenvolvedores possuam uma noção particular do que são tais anomalias. Nesse sentido, algoritmos de aprendizagem de máquina têm sido utilizados para customizar a detecção de *bad smells* a partir de um conjunto de avaliações. Entretanto, tal customização não é guiada a partir das diferentes heurísticas utilizadas pelos desenvolvedores para a detecção de *smells*. Como consequência tal customização pode não ser eficiente, exigindo um esforço considerável para obter uma alta efetividade. Esse trabalho apresenta um extensivo estudo que investiga o quão similar os desenvolvedores detectam *smells* em código, e analisa fatores que podem influenciar em tal detecção. As conclusões desse estudo motivaram a criação de uma técnica de customização guiada para melhorar a eficiência na detecção de *smells*. Essa técnica, definida como *Histrategy*, guia a customização a partir de um conjunto limitado de estratégias para detectar um mesmo tipo de *smell*. A partir de um estudo experimental que envolveu 62 desenvolvedores e 8 tipos de *bad smell*. Os resultados indicaram que a *Histrategy* apresentou performance superior a 6 algoritmos de aprendizagem de máquina, utilizados em abordagens não guiadas. Por fim, os resultados confirmaram que a customização guiada foi capaz assistir desenvolvedores com estratégias de detecção eficazes e eficientes.

Abstract

Bad smells indicate poor implementation choices that may hinder program comprehension and maintenance. Their informal definition allows developers to follow different heuristics to detect smells in their projects. In such context, machine learning algorithms have been adapted to customize smell detection according to a set of examples of smell evaluations. However, such customization is not guided (i.e. constrained) to consider alternative heuristics used by developers when detecting smells. As a result, their customization might not be efficient, requiring a considerable effort to reach high effectiveness. This work presents an extensive study concerning how similar the developers detect smells in code, and investigate which factors may influence in such detection. The findings of this study lead to the creation of *Histrategy*, a guided customization technique to improve the efficiency on smell detection. *Histrategy* considers a limited set of detection strategies, produced from different detection heuristics, as input of a customization process. The output of the customization process consists of a detection strategy tailored to each developer. The technique was evaluated in an experimental study with 62 developers and eight types of code smells. The results showed that *Histrategy* is able to outperform six widely adopted machine learning algorithms, used in unguided approaches. Finally, the results confirmed that the guided customization was able to support developers with effective and efficient detection strategies.

Agradecimentos

Agradeço à minha esposa, Elizabete, que ao longo dos últimos anos suportou comigo diversos momentos difíceis e adiou diversos planos para que eu pudesse realizar o curso de doutorado.

Aos meus pais, José Ferreira e Elza, aos meus irmãos e à Teófila (minha segunda mãe) pela base familiar necessária para enfrentar os desafios da vida.

Aos meus orientadores, Evandro Costa e Baldoino Fonseca, que me deram a oportunidade de trabalhar com eles e não mediram esforços para favorecer a condução das atividades da pesquisa realizada.

Agradeço à UFAL, pelo apoio financeiro, bem como os professores do Campus Arapiraca, que continuaram as atividades acadêmicas que deixei de conduzir durante meu afastamento da instituição.

Por fim, agradeço a todas as pessoas que contribuíram direta ou indiretamente para que eu chegasse até aqui.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Durante o ciclo de vida de um software diversas intervenções são realizadas por desenvolvedores a fim de lidar com sua evolução. Devido às necessidades do mercado atual, tais intervenções acontecem com restrições de tempo mais severas, exigindo entregas em períodos de tempo cada vez mais curtos. Consequentemente, a qualidade do código é frequentemente negligenciada, permitindo a introdução de problemas estruturais no mesmo [71].

Estudos anteriores indicam que a maior parte do custo total de um software está dedicado à sua manutenção [13] e que desenvolvedores dispendem mais tempo para entender um código do que para modificá-lo [2]. Assim, atividades relacionadas à preservação e melhoramento da qualidade do software tem se tornado cada vez mais importantes durante o seu ciclo de vida. Nesse contexto, a reestruturação do código conhecida como refatoramento (*refactoring*) tem se apresentado como uma importante atividade no processo de desenvolvimento de sistemas, diminuindo custos durante a evolução [74; 32; 63; 48]. Tal atividade, definida por Opdyke [68], prevê o melhoramento da estrutura interna de um programa sem a alteração do seu comportamento externo.

Nesse contexto, um trabalho liderado por Fowler [32] apresentou um catálogo que define um conjunto de anomalias de código conhecidas como *bad smell* (ou *code smell*), que caracterizam situações em que um programa pode ser melhorado através de refatoramento. Em seu catálogo, Fowler ressalta os aspectos negativos relacionados a 22 tipos de *bad smell* e indica algumas consequências que os mesmos podem trazer na evolução de um software. Mais recentemente, outros trabalhos já apresentam evidências que além de dificultar a compreensão [1] e prejudicar a manutenção [32], o crescimento na incidência de *bad smells* pode,

também, levar à degradação da arquitetura [67] e à propensão a falhas [46].

1.1 Problemática

Apesar do trabalho de Fowler ter importante contribuição para o entendimento do impacto negativo que os *bad smells* causam à qualidade de um software, a identificação destas anomalias em sistemas reais gera divergência de opiniões entre desenvolvedores. De fato, a ausência de uma definição clara para os *bad smells* conhecidos contribui para que diferentes interpretações sejam consideradas, fazendo com que desenvolvedores possuam uma noção particular do que são tais anomalias. Nesse sentido, Fowler ressaltou que seu catálogo não incluía como objetivo a indicação de critérios precisos para a identificação de *smells*, mas sim apresentar exemplos de situações em que refatoramentos podem ser aplicados para melhorar a qualidade de um sistema [32, página 63]. Assim, embora desenvolvedores concordem que a presença de *bad smells* em um projeto prejudica a qualidade do software, não se pode garantir que os mesmos reconheçam *smells* de forma similar [54; 29; 46].

Como exemplo, considere o *bad smell* conhecido por *Long Method*, como um método muito longo e que, por isso, dificulta seu entendimento e prejudica uma possível manutenção. A consequência de tal anomalia parece coerente com sua definição, uma vez que os principais modelos que investigam a qualidade de software definem a capacidade de entendimento como aspecto qualitativo para favorecer a manutenção [62; 40; 21; 22]. Entretanto, a detecção de um *Long Method* em um sistema comum pode envolver alguns questionamentos que resultam em reflexões subjetivas, tais como:

- Como julgar se um método é longo ou não?
- É possível julgar através do número de linhas?
- A partir de quantas linhas um método pode ser considerado um *Long Method*?
- Essa contagem deve considerar linhas em branco e/ou de comentários?
- A utilização de uma determinada tecnologia pode obrigar a utilização de métodos longos? Com isso estes casos deveriam ser considerados *bad smells*?

- Existe algum outro sintoma que deve ser considerado para julgar se um método é uma instância de *Long Method*?

Considerando que desenvolvedores distintos podem ter noções de qualidade diferentes, é de se esperar que estes respondam os questionamentos supracitados de forma individualizada e distinta. Tomando como parâmetro apenas a questão ligada ao número de linhas de código que torna um método comum em um exemplo de *Long Method*, é possível encontrar em diversos sites e fóruns especializados, desenvolvedores com opiniões distintas [23; 81; 82; 80; 16; 6]. Com isso, a detecção de *Long Method* para estes desenvolvedores não deveria ser tratada da mesma forma. Ademais, assim como ocorre com a detecção de *Long Methods*, o entendimento de outros *bad smells* apresentam características semelhantes, requerendo soluções que apresentem resultados distintos para cada desenvolvedor.

Embora diversos trabalhos tenham apresentado técnicas para a detecção de diversos tipos de *bad smell* [61; 65; 55; 69; 7], poucos estudos consideraram o que os desenvolvedores realmente entendem sobre tais anomalias. Consequentemente, muitas das soluções propostas não apresentaram mecanismos capazes de lidar com diferentes opiniões dos desenvolvedores sobre quando um determinado trecho de código apresenta um *bad smell*. Em contrapartida, abordagens que oferecem mecanismos para adaptar tal detecção introduzem novas dificuldades para executar tal customização.

Nesse contexto, algumas ferramentas que utilizam regras de detecção baseadas em métricas de software têm permitido que desenvolvedores indiquem diretamente a forma que os *smells* devem ser detectados [65; 73; 17; 39]. Porém, esta capacidade introduz uma rotina de configuração adicional que aumenta o esforço do desenvolvedor [26] e prejudica o processo de detecção [53].

Em contrapartida, trabalhos mais recentes introduziram mecanismos inteligentes para favorecer a definição de estratégias de detecção a partir de exemplos. Assim, algoritmos de aprendizagem baseados em redes *bayesianas* [45], árvores de decisão [4] e mecanismos baseados em SVM[1] [55] têm sido adaptados para produzir estratégias de detecção a partir de exemplos validados previamente por desenvolvedores. De maneira geral, tais abordagens apresentam bons resultados ao promover uma detecção mais precisa, porém introduzem

[1] *Support Vector Machine*

grande esforço para validar um conjunto de exemplos suficiente para definir estratégias de detecção efetivas [7].

1.2 Objetivo

Diante da dificuldade em promover a detecção de *bad smells* considerando o entendimento do desenvolvedor sobre tais anomalias, faz-se necessária a criação de uma abordagem capaz de detectar tais anomalias considerando percepções dos desenvolvedores. Assim, este trabalho teve como objetivos principais (i) apresentar um estudo exploratório que investigue o entendimento dos desenvolvedores sobre *bad smells*, e (ii) propor uma abordagem capaz de auxiliar o desenvolvedor a detectar *bad smells* sensíveis à intuição do indivíduo de forma efetiva e eficiente.

Com isso, incluem-se como objetivos específicos os seguintes pontos:

1. Avaliar a divergência observada quando diferentes desenvolvedores detectam instâncias de *bad smell* sobre um mesmo conjunto de trechos de código;

2. Investigar como a detecção de *bad smells* é influenciada através de fatores relacionados à atuação e experiência do desenvolvedor;

3. Avaliar como o entendimento do desenvolvedor pode auxiliar na promoção da detecção customizada;

4. Apresentar uma solução capaz de promover uma detecção de *bad smells* efetiva e eficiente para os desenvolvedores;

5. Implementar e disponibilizar a solução proposta, permitindo sua utilização de forma integrada no processo de desenvolvimento de um software.

A partir das conclusões de um estudo que investigou o entendimento de 75 desenvolvedores que avaliaram instâncias de 15 diferentes tipos de *bad smell* foi criada uma técnica de que oferece uma customização guiada para a detecção de *smells*. Essa técnica, definida como *Histrategy*, considera o entendimento do desenvolvedor para promover, de forma eficiente, a adaptação de estratégias de detecção eficazes. Nesse contexto, a *Histrategy* foi

avaliada juntamente com outras seis abordagens de customização que definem o estado da arte na detecção de *smells*. Os resultados dessa avaliação indicaram que as estratégias de detecção propostas pela *Histrategy* apresentaram maior efetividade e eficiência que as abordagens existentes. Assim, espera-se que os resultados desse trabalho possam colaborar com um melhor entendimento sobre como os *bad smells* são compreendidos pelos desenvolvedores. Ademais, espera-se que a técnica proposta possa apoiar a detecção de *bad smells* considerando os entendimentos distintos dos desenvolvedores ao analisar tais anomalias.

1.3 Organização do Documento

O restante deste documento está organizado como a seguir:

- No **Capítulo 2** é realizada uma contextualização acerca de *bad smells* e das abordagens de customização existentes.
- No **Capítulo 3** é apresentada uma investigação sobre o entendimento dos desenvolvedores sobre *bad smells*. Essa investigação é realizada através de um experimento envolvendo desenvolvedores que detectam anomalias em sistemas reais.
- No **Capítulo 4** é apresentada a abordagem proposta nesse trabalho para minimizar os problemas supracitados na Seção 1.1. Nesse contexto sua arquitetura é apresentada juntamente com uma descrição detalhada de seu funcionamento, incluindo o algoritmo de aprendizagem para a produção de regras personalizadas.
- No **Capítulo 5** é introduzida uma série de estudos que focadas em responder as questões de pesquisa supracitadas. Neste capítulo são detalhadas a operação, metodologia e resultados de cada estudo. Ameaças a validade e discussões acerca de cada investigação são incluídas no final.
- No **Capítulo 6** serão apresentados os trabalhos relacionados que investigam o entendimento dos desenvolvedores sobre *bad smells*. Por fim são apresentadas as abordagens que representam o estado da arte na detecção de *bad smells*, destacando aquelas que apresentam um processo customizado de acordo com o entendimento do desenvolvedor.

- Por fim, no **Capítulo 7** são apresentadas as considerações finais desse trabalho, contemplando suas contribuições, limitações e trabalhos futuros.

Capítulo 2

Contextualização

Nesse capítulo é realizada uma contextualização a partir da apresentação de alguns tipos de *bad smell* (Seção 2.1). Em seguida são apresentadas algumas abordagens conhecidas que visam detectar essas anomalias (Seção 2.2). Por fim, na Seção 2.3, é fornecida uma visão geral sobre como as abordagens existentes lidam com a customização da detecção para desenvolvedores.

2.1 *Bad Smells*

Fowler apresentou o termo *bad smell* para caracterizar estruturas em código que precisavam ser refatoradas [32] a fim de melhorar a qualidade do projeto. De fato, estudos anteriores apresentaram evidências que a presença de tais estruturas em um dado programa, pode prejudicar sua compreensão [1], degradar a arquitetura [67] e aumentar sua propensão a falhas [46].

Visando ajudar desenvolvedores a reconhecer a presença de *bad smells*, Fowler apresentou um catálogo com 22 tipos dessas anomalias, sugerindo alguns refatoramentos para removê-las [32]. Não obstante, apesar de fornecer subsídios para o entendimento geral de cada tipo de anomalia, o catálogo apresentado não forneceu mecanismos para detectar precisamente os *bad smells* em código. A Tabela 2.1 fornece uma breve descrição de cada um dos tipos de *smell* apresentados no catálogo de Fowler [32].

Embora a descrição de cada tipo de anomalia, juntamente com os exemplos apresentados possam ajudar os desenvolvedores a perceber as situações que degradam a qualidade do

Tabela 2.1: *Bad Smells* apresentados no catálogo de Fowler [32]

Nome	**Descrição**
Duplicated Code	Ocorre quando uma mesma estrutura de código aparece em mais de um lugar.
Long Method	Um método que é muito longo e possui muitas responsabilidades.
God Class	Se refere a classes que tendem a centralizar a inteligência do sistema.
Long Parameter List	Um método que possui muitos parâmetros.
Feature Envy	Se refere a métodos que usam muito mais dados de outras classes do que da classe em que estão definidos.
Data Clumps	Ocorre quando diferentes partes de código apresentam groups idênticos de variáveis.
Primitive Obsession	Ocorre quando tipos primitivos de dados são usados em demasia no software.
Switch Statements	Acontece quando os casos de um comando *switch/case* apresentam duplicação de código.
Speculative Generality	Ocorre quando um código é criado para favorecer funcionalidades futuras que nunca serão implementadas.
Temporary Field	Verifica-se em um objeto que possui uma variável de instância que apenas é utilizada em certas circunstâncias.
Message Chains	Ocorre quando um objeto cliente requisita um objeto que é requisitado por outro.
Middle Man	Ocorre quando grande parte dos métodos de uma classe delega a execução para métodos de outra classe.
Inappropriate Intimacy	Verifica-se quando uma classe utiliza inadequadamente as estruturas internas de uma outra classe.
Data Class	Caracteriza-se por classes que possuem apenas atributos e métodos *get/set*.
Refused Bequest	Ocorre quando uma subclasse praticamente não utiliza os dados e métodos herdados da classe pai.
Divergent Change	Ocorre quando uma classe é comumente modificada de diferentes formas e por diferentes razões.
Shotgun Surgery	É o oposto de *Divergent Change*. Ocorre quando uma modificação em uma determinada classe requer uma série de pequenas mudanças em outras classes.
Parallel Inheritance Hierarchies	Caracteriza-se como um caso especial de *Shotgun Surgery*. Sempre que uma determinada classe é modificada uma outra classe também precisa ser alterada.
Lazy Class	Verifica-se em classes que não fazem nada no sistema e deveriam ser eliminadas.
Comments	Ocorre quando um comentário é utilizado para "justificar" uma implementação ruim e que deve ser refatorada.
Alternative Classes with Different Interfaces	São classes que fazem coisas similares mas possuem assinaturas diferentes.
Incomplete Library Class	Ocorre quando as bibliotecas de classes utilizadas não estão completas.

projeto de um software, a detecção destes *bad smells* não foi discutida no catálogo de Fowler [32]. Nesse ponto, o autor deixou claro que seu catálogo não incluía como objetivo a indicação de critérios precisos para a identificação de *smells*, mas sim apresentar exemplos de situações em que refatoramentos podem ser aplicados para melhorar a qualidade de um sistema [32, página 63].

2.2 Detecção de *Bad Smells*

Diante da relevância e da popularização do termo *bad smell*, diversos trabalhos propuseram técnicas focadas na detecção destas anomalias, a fim de assistir desenvolvedores a garantir a qualidade dos sistemas construídos por eles. A seguir são apresentadas as principais propostas para a detecção de tais anomalias, abordando como essa atividade pode ser realizada de forma manual e automática.

2.2.1 Abordagens de Detecção Manual

A detecção de *bad smells* através de inspeções dos desenvolvedores é discutida em [84]. Nesse trabalho, os autores propõem uma técnica de inspeção do software que busca aumentar a efetividade dos revisores a partir de algumas diretrizes usadas para examinar um software Orientado a Objetos (OO) na busca por problemas no projeto. Tais diretrizes foram concebidas pelos autores como um novo tipo de técnica de inspeção que tenta rastrear informações principais do projeto através de seus requisitos e documentos. Por fim, os autores ainda realizaram um estudo onde 44 estudantes aplicaram a técnica proposta na detecção de problemas de projeto em um sistema orientado a objetos relacionado ao contexto financeiro. Os resultados apontaram que os desenvolvedores reconheceram avanços para a detecção de problemas em projetos OO, mas a técnica ainda carece de artefatos semânticos que ajudem no julgamento das anomalias encontradas.

Apesar das técnicas manuais permitirem que a detecção seja realizada de forma coerente com a percepção do desenvolvedor sobre *bad smells*, o que tornaria a detecção mais precisa, esse tipo de técnica não favorece a repetição do processo em outras situações parecidas. Assim, as abordagens manuais, de maneira geral, tornam-se tarefas não-repetíveis, não-escaláveis e que consomem muito tempo para a realização [59].

2.2.2 Técnicas de Detecção Automática

Diante das dificuldades relacionadas à detecção manual, pesquisadores investigaram técnicas que pudessem realizar a identificação de *smells* de forma automática. Assim, Kataoka *et al.* propuseram uma abordagem para a detecção de *bad smells* através da descoberta de invariantes no código [44]. Exemplos de invariantes detectados por esta abordagem incluem parâmetros que não são utilizados no método ou que são sempre constantes, assim como instruções de retorno constantes ou em função de outros métodos. Como esta técnica utiliza uma coleção de testes para identificar os invariantes, a análise se torna dependente de um cenário de execução. Além disso, mesmo com as ferramentas para a criação automática de testes, é impossível garantir que os testes utilizados forneçam uma cobertura razoável para a detecção de todos os invariantes. Por fim, percebe-se que a maioria dos exemplos de *smells* indicados por Fowler [32] não apresentam invariantes em sua implementação, tornando a abordagem inadequada para a detecção dessas anomalias.

O trabalho apresentado por van Emden e Moonen [85] utiliza um meta-modelo extraído do software analisado para detectar *smells* automaticamente. O meta-modelo representa o software analisado em uma estrutura de árvore que pode ser navegada para encontrar sintomas encontrados em *bad smells* direta ou indiretamente. Por fim, os autores apresentam uma ferramenta que permite a visualização desta árvore com atributos que permitem o desenvolvedor analisar com mais detalhes os resultados gerados na detecção de alguns dois tipos de *bad smells* definidos pelos autores: *Instanceof*, que descreve o uso do operador Java de mesmo nome, e *Typecast*, que permite a conversão de um tipo de objeto para outro. Nesse trabalho, nenhum *bad smell* do catálogo de Fowler [32] é investigado.

Nesse cenário, métricas de software que já eram investigadas para a aferição de qualidade [21; 22; 10; 11; 19; 52], foram utilizadas, também, para identificar as partes de código que precisavam ser refatoradas. Em [59], Marinescu investigou o uso de métricas de software para a identificação de *bad smells* utilizando uma técnica que evoluiu para o conceito de estratégias de detecção. Segundo o autor, uma **estratégia de detecção** consiste em uma expressão quantificável de um regra segundo a qual os fragmentos de um projeto que estão em conformidade com a regra podem ser detectados no código fonte. A criação dessa regra é realizada em um processo de quatro passos que parte da análise das descrições informais dos *bad smells*, passa pela seleção de métricas apropriadas para a detecção e

conclui com a execução e validação da estratégia resultante. Além de observar as métricas definidas [10], o autor investiga outros conjuntos de métricas propostos na literatura [18; 15], compondo regras para a detecção de dois tipos de *bad smell*: *Data Class* e *God Class*.

Em um outro trabalho [60], Marinescu detalhou o processo para a definição das estratégias de detecção partindo da seleção de métricas, seguindo para a definição dos limiares de aceitação e, por fim, a composição da regra de detecção. O autor destaca que existe um grande desafio relacionado à definição de limiares apropriados para compor a regra de detecção. Afinal, a escolha deles vai determinar a acurácia da regra na identificação de *smells* para os desenvolvedores. Este processo de definição da estratégia de detecção é exemplificado para a identificação de *God Classes* que é avaliada com mais outras estratégias para a detecção de nove tipos diferentes de *bad smells* em duas versões de um mesmo sistema de médio porte. Apesar do trabalho indicar que humanos avaliaram a precisão dos resultados produzidos em 70%, não foi reportado quantos desenvolvedores participaram do processo nem a experiência deles com a detecção das anomalias.

As definições das estratégias de detecção propostas por Marinescu foram reunidas em um livro escrito juntamente com outros autores [51]. Nesse livro, 11 estratégias são discutidas detalhando o processo de entendimento da anomalia, seleção de métricas apropriadas para a detecção e definição limiares utilizados na composição das regras. O livro ainda reúne um catálogo detalhando todas as métricas utilizadas para a composição das estratégias de detecção.

O trabalho apresentado em [66] segue a abordagem de Marinescu para a criação de estratégias de detecção para dois tipos de *bad smell*. O autor utilizou as regras propostas para detectar anomalias em um sistema de pequeno porte e outro de médio porte. A avaliação da proposta foi feita com humanos apenas com a detecção sobre o sistema de pequeno porte. Entretanto, o trabalho não indica quantos desenvolvedores participaram da análise e nem descreveram as habilidades que eles possuíam para fazer tal trabalho. Apenas foi indicado que as análises seguiram as determinações informais sobre *bad smells* apresentadas no catálogo de Fowler [32].

Embora as estratégias de detecção tenham suprimido as dificuldades observadas na detecção manual, tais estratégias apresentam alguns problemas relacionados à detecção para diferentes desenvolvedores. Afinal, enquanto a automatização é satisfeita através de regras

compostas por métricas que podem ser computadas a partir do código fonte, a detecção em si ignora o entendimento individual de cada desenvolvedor a respeito dos *bad smells* avaliados. Por exemplo, em algumas ferramentas [73; 17] a estratégia de detecção para identificar *Long Method* tem sido definida como:

$$MLOC(m) > \alpha \rightarrow LongMethod(m)$$

onde *MLOC* denota a métrica de software que computa o número de linhas do método *m* e α define um número inteiro com o limiar necessário para se considerar um método comum como *Long Method*, a partir do seu número de linhas. Contudo, a utilização dessa estratégia de detecção limita a percepção dos desenvolvedores acerca dessa anomalia, fazendo com que a detecção seja realizada dentro de um padrão pré-estabelecido pela métrica *MLOC* com o limiar definido. Nesse caso, as diferentes percepções dos desenvolvedores poderiam ser mapeadas com métricas e limiares diferentes para a detecção de métodos longos.

Tal conclusão é evidenciada no estudo apresentado por Mäntylä *et al.* [58], que investigou a relação entre avaliações realizadas por 12 desenvolvedores confrontando-as com os valores das principais métricas utilizadas para detectar três tipos de *bad smell*. Nesse caso, os autores concluíram que não existia uma correlação significante entre as anomalias reportadas pelos humanos em comparação com os valores das métricas aferidas sobre os módulos avaliados. Mesmo para a detecção de *Long Parameter List* onde, teoricamente, a detecção através da métrica *NPARAM*, que computa o número de parâmetros em cada método traduziria de forma mais direta a definição do *bad smell*, os desenvolvedores produziram análises conflitantes, indicando limiares de detecção diferentes.

2.3 Customização da Detecção de *Bad Smells*

Como forma de minimizar as limitações relacionadas à definição de estratégias gerais para a detecção de *bad smells*, ferramentas mais recentes, tais como inFusion [39], JDeodorant [42], Checkstyle [17] e PMD [73], têm criado mecanismos que permitem a configuração dos limiares da regra de detecção. Assim, os desenvolvedores podem, manualmente, indicar os limiares adequados à sua percepção sobre a anomalia que será detectada. Entretanto, apesar de flexibilizar a detecção para desenvolvedores diferentes, esta capacidade introduz uma

rotina de configuração adicional que aumenta o esforço do desenvolvedor [26] e prejudica o processo de detecção [53]. Afinal, essa tarefa obriga o desenvolvedor a ter um conhecimento adicional sobre o funcionamento das estratégias de detecção, das métricas utilizadas e dos limiares definidos para a detecção de anomalias. Em um primeiro momento isso pode parecer simples ao se analisar a estratégia supracitada para a detecção de métodos longos. Entretanto, em estratégias mais complexas, envolvendo mais de uma métrica, tal rotina se torna extremamente difícil devido à complexidade envolvida na composição da regra de detecção. Como exemplo, pode-se citar a estratégia de detecção para detecção de anomalias do tipo *Feature Envy*, que utiliza a seguinte regra de detecção

$$(ATFD(m) > 5 \wedge LAA(m) < \frac{1}{3} \wedge FDP(m) \leq 5) \rightarrow FeatureEnvy(m)$$

onde a métrica *ATFD* (*Access to Foreign Data*) denota o número de atributos de classes não relacionadas que são acessadas diretamente ou através de métodos de acesso, *LAA* (*Locality of Attribute Accesses*) indica o número de atributos definidos na classe, divido pelo total de variáveis acessadas na mesma, e *FDP* (*Foreign Data Providers*) é definido pelo número de classes nas quais possui atributos acessados pela classe aferida.

Similarmente, Moha *et al.* propuseram um *framework* que permite especificar as características de um *bad smell* com mais atributos [65]. Nesse *framework*, denominado DECOR, a detecção é realizada seguindo as características expressas em uma linguagem específica de domínio (DSL[1]), que permite descrever as características das anomalias utilizando, além das métricas e limiares, informações relacionadas à estrutura do projeto (como nomes de classes, métodos, atributos entre outros) e características do paradigma OO (como a verificação de polimorfismo, composição, agregação e herança). Novamente, essa abordagem obriga o desenvolvedor a ter um conhecimento adicional sobre a linguagem de especificação da detecção de *bad smells*.

Customização por aprendizagem de máquina

Em [49], Kreimer propõe uma técnica que usa a classificação manual para treinar um algoritmo de aprendizado de máquina baseado em árvores de decisão. Nesse caso, os nós da árvore determinam as métricas utilizadas na detecção, as arestas indicam os limiares utili-

[1] do Inglês *Domain-specific Language*

zados e as folhas determinam se um trecho de código que respeita as restrições indicadas pelo caminho da árvore é um *smell* ou não. Essa técnica objetiva adaptar a árvore de decisão utilizada para a detecção de *smells*, a partir de um conjunto de exemplos de *smell* anotados pelos desenvolvedores. A avaliação da abordagem é realizada com um único desenvolvedor na análise de dois *smells* (*Long Method* e *God Class*) em dois projetos de código aberto. A avaliação foi realizada em duas fases, treino e teste. Primeiramente, na fase de teste, o desenvolvedor analisou 20 trechos de código de cada projeto e anomalia, indicando se o *smell* analisado estava presente ou não. Este conjunto de análises foi utilizado para produzir a árvore de decisão que seria utilizada na segunda fase, para detectar *smells* em outros 20 trechos de código. Por fim, o autor observou que a árvore de decisão criada obteve uma precisão em torno de 86%. Apesar de apresentar uma precisão elevada, a avaliação realizada não reportou se muitos *smells* deixaram de ser reportados pela ferramenta. Além disso, a ferramenta objetiva promover uma detecção personalizada de acordo com o entendimento de cada pessoa, entretanto a avaliação apresentada utiliza apenas um único desenvolvedor para verificar a efetividade da ferramenta. Por fim, o trabalho reporta que a construção de uma árvore de decisão efetiva se consegue com um conjunto mínimo de 100 análises, fazendo com que a utilização da ferramenta dependa de uma etapa inicial onde o usuário forneça esses dados, tornando a atividade de detecção mais custosa, e que demanda tempo e experiência do desenvolvedor.

O uso de técnicas de aprendizagem de máquina também foi explorado em [55]. Neste trabalho, a abordagem, denominada SMURF, utiliza um mecanismo baseado em uma Máquina de Vetores de Suporte (SVM[2]) para detectar *smells*. Este mecanismo utiliza como entrada os valores das métricas associadas às classes incluindo a indicação se a classe contém uma anomalia ou não. Esses dados são utilizados para treinar o classificador *SVM*, habilitando-o para uma detecção futura. A partir do uso desse classificador, o desenvolvedor pode concordar ou não com cada smell reportado, fazendo com que essa análise seja adicionada ao conjunto de dados de treino para melhorar o classificador produzido. Os autores avaliaram o *SMURF* na detecção de quatro tipos de *smells* em três projetos de código aberto, e compararam os resultados com uma implementação do *DECOR* [65]. Os resultados do experimento mostraram vantagem do *SMURF* em todos os cenários de detecção, atingindo uma precisão acima de

[2]do Inglês *Support Vector Machine*

66% e detectando pelo menos 57% dos *smells* presentes no estudo. Percebeu-se que, assim como no trabalho descrito anteriormente, esta abordagem requer uma base de treino para poder ser utilizada efetivamente, introduzindo um novo trabalho para o desenvolvedor antes de executar a detecção. Por fim, o trabalho não deixou claro se as bases de exemplos foram criadas e avaliadas individualmente para cada desenvolvedor que validou a existência dos *smells*.

Mais recentemente, Fontana *et al.* [7] apresentaram um grande estudo que comparou e experimentou seis algoritmos de aprendizagem de máquina para a detecção de *bad smells*. No estudo foram utilizadas diferentes configurações dos algoritmos *J48*, *JRip*, *Random Forest*, *SMO*, *Naive Bayes* e *SVM*. Durante a validação, os autores consideraram um conjunto com mais de 1.900 trechos de código contendo *bad smells* manualmente validados por diferentes desenvolvedores. Os resultados reportaram que todas as técnicas avaliadas apresentaram alta acurácia, principalmente com os algoritmos baseados em árvores de decisão (*J48* and *Random Forest* [64]). Os autores reportaram ainda que as técnicas avaliadas precisaram de centenas de exemplos para atingir uma acurácia de pelo menos 95%. Entretanto, o estudo não avaliou o desempenho dos algoritmos individualmente para cada um dos desenvolvedores que criou a base de *smells* anotados.

Capítulo 3

Avaliando a Concordância dos Desenvolvedores na Classificação de *Bad Smells*

As definições informais e subjetivas dos diversos tipos de *bad smells* [32] permitem que desenvolvedores reconheçam tais anomalias de forma diferente [7]. Como consequência, os desenvolvedores podem concordar ou discordar sobre a ocorrência de um determinado *bad smell* ao analisar um trecho de código específico. Nesse contexto, a concordância (ou discordância) entre os desenvolvedores tem impacto direto na forma como as ferramentas de detecção existentes podem ajudá-los.

Ao se considerar que, em geral, os desenvolvedores concordam entre si ao avaliar instâncias de *bad smells* a detecção automatizada promovida por ferramentas existentes, tais como [60; 51; 66; 45; 31; 14; 65], poderia auxiliar grande parte dos desenvolvedores a partir da indicação precisa dos trechos que contêm anomalias. Afinal, tais ferramentas adotam estratégias de detecção universais, que apresentam os mesmos resultados para todos os desenvolvedores que as utilizem. De outro modo, caso a discordância entre desenvolvedores seja comum, os benefícios destas ferramentas seriam reduzidos, fazendo com que seus resultados reportassem anomalias em que desenvolvedores discordam da existência de *bad smells*. Como consequência, uma nova validação manual teria que ser feita a partir dos resultados apresentados, exigindo tempo e esforço dos utilizadores dessas ferramentas.

Como observado no Capítulo 6, poucos trabalhos apresentaram estudos que investigam

quão similar os desenvolvedores detectam *smells* em código. Em particular, não existem evidências que indiquem que certos fatores podem influenciar desenvolvedores a apresentarem opiniões convergentes ao detectar instâncias dessas anomalias. Diversos fatores relacionados às características dos desenvolvedores podem desempenhar um papel importante no processo de detecção de *bad smells*. Tais fatores podem incluir características relacionadas à sua formação ou de acordo com as experiências obtidas no desenvolvimento de software. Independentemente do perfil de desenvolvedor, uma característica mais pessoal também pode influenciar os desenvolvedores a detectarem *bad smells* diferentemente. Por exemplo, cada desenvolvedor pode seguir uma heurística própria para detectar *bad smells* independente dos outros fatores supracitados.

Neste capítulo é apresentado um estudo que visa investigar a concordância entre desenvolvedores ao detectar instâncias de 15 tipos de *bad smells* diferentes. Além disso, o estudo também busca analisar se certos fatores relacionados aos desenvolvedores também influenciam na concordância entre eles. O estudo conta com 75 desenvolvedores que avaliaram a presença de *bad smells* em um grande conjunto de trechos de código extraídos de projetos reais. Ao todo, mais de 2.700 avaliações foram coletadas e analisadas, permitindo a avaliar a concordância entre os desenvolvedores com significância estatística. Além disso, foi analisada a influência de 6 fatores sobre a concordância dos desenvolvedores.

O restante deste capítulo está estruturado como segue. Na Seção 3.1 é apresentado o projeto do estudo, indicando questões de pesquisa, seleção dos indivíduos, procedimento e métodos de análise dos dados. Na Seção 3.2 são apresentados os resultados obtidos com a realização do estudo. Na Seção 3.3 destacam-se as principais discussões obtidas a partir dos resultados apresentados. A Seção 3.4 descreve as principais ameaças à validade do estudo, enquanto a Seção 3.5 apresenta as conclusões do estudo.

3.1 Projeto do Estudo

O estudo apresentado visa avaliar a concordância entre desenvolvedores ao detectar *bad smells* em código, a partir da análise de trechos de código que contêm instâncias de diferentes tipos de anomalias. Ademais, o estudo investiga se alguns fatores relacionados aos desenvolvedores poderia influenciá-los a compartilhar uma mesma percepção sobre *bad smells*.

Nesse sentido, duas questões principais guiaram o estudo:

- **Q1**: **Os desenvolvedores concordam ao avaliar *bad smells* em código?** A motivação para essa questão de pesquisa consiste em investigar se, em geral, os desenvolvedores concordam ao detectar *bad smells* em códigos de projetos reais. Além disso, é avaliado o grau de tal concordância a fim de verificar quão diferente eles avaliam a presença de tais anomalias sobre um mesmo conjunto de exemplos de código.

  Particularmente, essa análise se torna difícil porque requer a participação de diversos desenvolvedores com diferentes características a fim de criar uma amostra relevante. Afinal, tais requisitos podem ter reduzido as conclusões de estudos anteriores que realizaram investigações similares [56; 57]. Assim o estudo foi pensado para realizar uma investigação mais detalhada a fim de incrementar o conhecimento sobre quão similar os desenvolvedores detectam *smells* em código.

- **Q2**: **O que faz os desenvolvedores concordarem ao avaliar *bad smells* em código?** A motivação para essa questão visa analisar se certos fatores podem influenciar os desenvolvedores a concordarem ou discordarem ao detectar *bad smells*. Particularmente, é verificado se os desenvolvedores detectam tais anomalias de maneira similar, ao serem agrupados de acordo com características próprias em comum.

  Os fatores analisados foram escolhidos com base em trabalhos anteriores que investigaram a relação dos mesmos com a concordância dos desenvolvedores ao detectar *bad smells*. Embora análises similares acerca de fatores relacionados à atuação e experiência tenham sido realizadas em outros trabalhos [56; 57; 76; 75], espera-se que o estudo apresentado possa confirmar as conclusões observadas a partir de uma investigação mais robusta, conforme mencionado anteriormente. Além desses fatores, o estudo investiga como o julgamento dos desenvolvedores é influenciado pelas heurísticas que eles formulam para detectar *smells*. Os resultados dessa investigação podem revelar o potencial e as limitações das técnicas de detecção de *smell* customizadas a partir dos fatores analisados.

Nas seções seguintes serão detalhados os principais componentes envolvidos nesse estudo, descrevendo os componentes utilizados, operação e os métodos de análise dos resultados.

Tabela 3.1: *Bad Smells* investigados no estudo

Duplicated Code	Long Method	God Class	Long Parameter List	Feature Envy
Data Clumps	Primitive Obsession	Switch Statements	Speculative Generality	Temporary Field
Message Chains	Middle Man	Inappropriate Intimacy	Data Class	Refused Bequest

3.1.1 Desenvolvedores e *Bad Smells*

O estudo contou com desenvolvedores com diferentes atuações e experiências visando investigar quão diferente os desenvolvedores detectam *bad smells*. Nesse sentido, eles foram classificados de acordo com duas categorias de atuação. A primeira indica desenvolvedores com atuação na **academia**, incluindo estudantes de graduação, mestrado e doutorado, bem como professores que conduzem pesquisas relacionadas com a detecção de *bad smells*. A segunda contou com profissionais que trabalham com desenvolvimento de software na **indústria**. Nos casos em que o participante tenha experiência tanto na academia quanto na indústria, foi considerada a atuação onde o mesmo tenha trabalhado por um período de tempo maior.

Independentemente da atuação em que o desenvolvedor foi enquadrado, todos os participantes do estudo tiveram algum contato com a detecção de *bad smells* em código. Essa restrição foi considerada no estudo a fim de evitar que desenvolvedores sem conhecimentos sobre a detecção de *bad smells* pudessem responder as perguntas realizadas no experimento de forma aleatória, devido à falta de conhecimento no assunto.

Os desenvolvedores avaliaram trechos de código de 15 diferentes tipos de *bad smell*, descritos na Tabela 3.1. Estes *bad smells* foram escolhidos devido ao fato de estarem relacionados a diferentes escopos em um código fonte. Por exemplo, enquanto instâncias dos *smells God Class* e *Data Class* estão relacionados às classes de um software, instâncias de *Long Method* e *Long Parameter List* estão mais relacionadas a métodos individuais de uma classe.

3.1.2 Trechos de Código

Para cada tipo de *bad smell* os desenvolvedores avaliaram 15 trechos de código com diferentes características. Nesse caso, os trechos de código foram escolhidos visando apresentar uma variedade de exemplos suspeitos de apresentar o tipo de *smell* em análise. Por exemplo,

no caso dos *bad smells Long Method* e *Long Parameter List*, que são comumente detectados através de métricas de software, como linhas de código e número de parâmetros, foram escolhidos exemplos com diferentes combinações dessas métricas. Os métodos escolhidos para avaliação de *Long Methods* apresentavam tamanho que variavam de 42 a 192 linhas de código. Já os trechos de código relacionados ao *bad smell Long Parameter List* apresentaram métodos e construtores contendo de 5 a 9 parâmetros.

O Código Fonte 3.1[1] apresenta um exemplo de trecho de código analisado pelos desenvolvedores que avaliaram a presença de *Long Parameter List*. Embora todo o arquivo que contém a classe `RegularFrameOffsetBuilder` tenha sido apresentado para os desenvolvedores que avaliaram esse tipo de *bad smell*, a análise da presença de *Long Parameter List* deveria ser realizada considerando apenas o construtor da classe definido na *linha 8*.

Os trechos de código apresentados para os desenvolvedores foram selecionados a partir de relatórios apresentados em estudos anteriores [65; 55; 69; 72] e identificados por abordagens [51; 66; 39; 73; 17; 42; 34] que são capazes de detectar os *bad smells* investigados no estudo. Ao final, os trechos de código selecionados foram extraídos de cinco projetos de código aberto escritos em Java: GanttProject [33] (2.0.10), Apache Xerces [5] (2.11.0), ArgoUML [83] (0.34), jEdit [43] (4.5.1) e Eclipse [24] (3.6.1). Estes projetos foram escolhidos por já serem investigados em estudos anteriores que relataram uma variedade de *bad smells* suspeitos que permitiriam a execução do estudo.

3.1.3 Operação

O experimento realizado no estudo foi projetado para coletar a opinião dos desenvolvedores sobre a existência de *bad smells* nos trechos de código selecionados. Tais opiniões foram coletadas a partir de uma série de avaliações realizadas através de uma aplicação baseada na Internet a qual foi construída exclusivamente para esse estudo.

Antes de iniciar as avaliações, foram registrados nome, e-mail e atuação (academia ou indústria) dos desenvolvedores envolvidos no experimento. Tais desenvolvedores foram divididos aleatoriamente em grupos de 12 desenvolvedores, incluindo 6 participantes da *academia* e 6 da *indústria*. Nesse sentido os 12 desenvolvedores de um mesmo grupo foram respon-

[1]Para favorecer a apresentação foram omitidas algumas partes da classe como importações, atributos e métodos

Código Fonte 3.1: Trecho de código para análise de *Long Parameter List*

```
1  package net.sourceforge.ganttproject.chart;
2  // Imports
3
4  class RegularFrameOffsetBuilder {
5
6    // Atributos de classe
7
8    RegularFrameOffsetBuilder(
9          TimeUnitStack timeUnitStack, GPCalendar calendar, TimeUnit topUnit,
10         TimeUnit bottomUnit, Date startDate,
11         int bottomUnitWidth, int chartWidth, float weekendDecreaseFactor, Date endDate) {
12       myTimeUnitStack = timeUnitStack;
13       myCalendar = calendar;
14       myStartDate = startDate;
15       myTopUnit = topUnit;
16       myBottomUnit = bottomUnit;
17       myBottomUnitWidth = bottomUnitWidth;
18       myChartWidth = chartWidth;
19       myWeekendDecreaseFactor = weekendDecreaseFactor;
20       myEndDate = endDate;
21   }
22
23   // Metodos da classe
24 }
```

sáveis por avaliar trechos de código de um determinado tipo de *bad smell*. Esse critério foi utilizado para criar grupos heterogêneos a fim de analisar, separadamente, as avaliações realizadas pelos desenvolvedores da academia e da indústria.

Após definir os grupos responsáveis por avaliar cada tipo de *bad smell*, foram enviados e-mails para cada desenvolvedor, incluindo uma URL e uma mensagem de convite. Ao abrir essa URL, o desenvolvedor pôde aceitar o convite e iniciar a participação no experimento, que foi realizado em três etapas conforme ilustrado na Figura 3.1. As principais atividades do experimento são detalhadas a seguir.

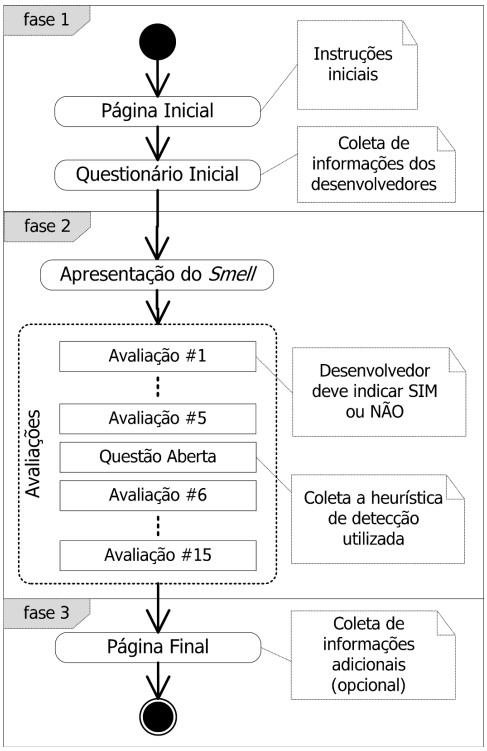


Figura 3.1: Fluxo da participação dos desenvolvedores

Página Inicial - A participação do desenvolvedor se inicia na etapa 1, onde é apresentada uma página de boas vindas contendo uma breve descrição sobre o experimento. Nesse ponto, o desenvolvedor é informado que a sua participação pode ser interrompida a qualquer momento, e continuada, através do acesso da mesma URL que foi enviada por e-mail. Em seguida, o desenvolvedor deve responder um questionário inicial, onde ele avalia sua experiência em (i) desenvolvimento de software, (ii) na linguagem de programação Java e (iii) na detecção de *bad smells*. Assim como na verificação da atuação, esse procedimento foi realizado a fim de analisar se essas características podem influenciar na opinião dos de-

senvolvedores ao detectar os *smells* investigados. Nesse caso, a auto-avaliação foi utilizada porque esse procedimento tem sido bastante utilizado e considerado adequado para estimar a experiência em programação, como observado em [79].

Apresentação do *Bad Smell* - Após completar o questionário da etapa inicial, o participante é conduzido para a etapa 2, onde ele realiza as avaliações de acordo com o tipo de *bad smell* atribuído ao grupo que ele pertence. No início dessa etapa, o participante é apresentado a uma descrição do *bad smell* em análise. Esta descrição é baseada em uma definição informal definida no catálogo de Fowler [32]. As definições providas nesse catálogo ajudam os desenvolvedores a: (i) entender como os *bad smells* se apresentam em um programa, e (ii) definir estratégias para detectá-los. Diferentemente do estudo apresentado em [57], nenhum exemplo do *bad smell* analisado é apresentado ao participante a fim de evitar a introdução de um viés no entendimento do desenvolvedor acerca do *smell* estudado. Caso contrário, os participantes poderiam ser influenciados pelas características do exemplo apresentado ao avaliar os trechos de código do experimento.

Avaliações - Em seguida, o desenvolvedor inicia uma sequência de 15 avaliações dos trechos de código relacionados ao *bad smell* introduzido na atividade anterior. Em cada avaliação, o código fonte de arquivos Java contendo o trecho de código que deve ser analisado é apresentado ao desenvolvedor. Nesse cenário o desenvolvedor deve analisar e indicar se o *smell* investigado está se manifestando no trecho de código destacado no código apresentado. Durante a avaliação o desenvolvedor pode navegar no código fonte a fim de inspecionar outras partes do projeto analisado. Ao fim de cada avaliação, o desenvolvedor deve responder **SIM** ou **NÃO** indicando se o trecho de código contém o *bad smell* investigado.

Além de prover as respostas (SIM ou NÃO), o desenvolvedor deve responder a uma questão aberta que é apresentada após a quinta avaliação. Nessa resposta o desenvolvedor deve informar a heurística que ele utilizou para indicar se os trechos de código apresentados continham o *bad smell* investigado. Essas heurísticas foram coletadas a fim de investigar as diferentes formas que os desenvolvedores detectam instâncias de um mesmo tipo de *bad smell*.

Página Final - Após completar as avaliações, o desenvolvedor é conduzido para um a página final, já na etapa 3. Nessa página, o desenvolvedor pode reportar problemas e dificuldades ocorridas durante a execução do experimento.

Para garantir uma amostra de dados maior, que permitisse uma análise mais robusta dos dados coletados, os participantes puderam avaliar mais de um tipo de *bad smell* analisado no estudo. Nesses casos o desenvolvedor repetiu as atividades da fase 2 de acordo com o número de tipos de *smell* que avaliou. Para garantir que as avaliações de um único participante não tivesse grande representação na amostra coletada, o estudo limitou as avaliações de um mesmo desenvolvedor para até três tipos de *bad smell* diferentes.

3.1.4 Questões Abertas

Conforme descrito na seção anterior, os desenvolvedores precisaram responder a uma questão aberta reportando a heurística utilizada para detectar os *smells* analisados. Essa questão foi introduzida após a quinta avaliação por dois motivos principais. Primeiro, porque após desenvolvedor realizar algumas avaliações, espera-se que ele esteja mais preparado para responder a heurística que utilizou para responder às avaliações anteriores. Como segundo motivo, compreende-se que uma longa série de avaliações pode tornar a experiência tediosa e cansativa, assim a inserção de uma questão no meio das avaliações poderia evitar a realização de uma grande série de avaliações, mantendo a atenção do desenvolvedor durante todas as fases do experimento. Assim, essas questões também foram utilizadas como ponto de controle para identificar se o desenvolvedor está sendo negligente em suas avaliações.

As respostas para as questões abertas têm um papel importante no experimento, uma vez que elas podem ajudar a analisar quão similar os desenvolvedores detectam um mesmo tipo de *bad smells*. Embora os desenvolvedores tenham sido introduzidos com uma única definição para cada tipo de *bad smell* durante a etapa das avaliações, é possível que os mesmos utilizem formas diferentes para detectar os *smells*. Assim, a partir dessas respostas, é possível aplicar um procedimento de *coding* [78] a fim de identificar os desenvolvedores que utilizaram a mesma heurística para detectar um determinado tipo de **bad smell**. Por fim, é possível investigar se esses desenvolvedores apresentam uma grande concordância em suas avaliações.

3.1.5 Análise dos Dados

A fim de responder às questões de pesquisa definidas para esse estudo, foi utilizada uma medida que computa a concordância entre as avaliações dos desenvolvedores. Tal concordância é calculada através do *Fleiss' Kappa*, uma medida utilizada para verificar a concordância entre múltiplos avaliadores [27]. Tal medida reporta um número menor ou igual a **1**, indicando a concordância aferida. Caso o valor de *Kappa* seja igual a **1**, então os avaliadores apresentaram uma concordância perfeita. Em outro caso, a discordância é mais presente à medida que esse valor seja mais distante de **1**.

Além do valor do *Kappa*, a análise realizada nesse estudo considerou as categorias propostas por Landis e Koch [50], como descrito na Tabela 3.2. Tais categorias têm sido utilizadas em trabalhos anteriores relacionado à detecção de *bad smells* visando verificar o nível de concordância a partir do valor aferido para a medida *Kappa*.

Tabela 3.2: Classificação de Landis e Koch para os valores de *Kappa*

Valor de Kappa	Nível de concordância
< 0,00	Pobre
0,00 - 0,20	Ligeira
0,21 - 0,40	Justa
0,41 - 0,60	Moderada
0,61 - 0,80	Substancial
0,81 - 1,00	Perfeita

Considerando o uso da medida de *Kappa* juntamente com as categorias propostas por Landis e Koch, é possível responder às questões de pesquisa definidas no estudo. Para responder a questão **Q1**, foi realizado o seguinte procedimento para cada tipo de *bad smell* investigado:

1. Foram coletadas as avaliações realizadas pelos 12 desenvolvedores responsáveis por avaliar 15 trechos de código relacionados a um único tipo de *smell*. Tais avaliações produziram uma matriz de 12x15 contendo as respostas (SIM ou NÃO) de acordo com cada avaliação.

2. A partir da matriz de avaliação, foi computado o valor de *Kappa* visando obter o grau de concordância entre os 12 desenvolvedores. A partir desse valor, foi verificada a classificação da força da concordância de acordo com as categorias descritas na Tabela 3.2

Após obter o valor de *Kappa* e suas classificações, é possível responder à questão **Q1** a partir da análise da força da concordância verificada. Nesse caso, foram considerados com concordância boa os casos onde o valor de *Kappa* estava relacionado com as três melhores classificações apresentadas na Tabela 3.2, quando a concordância é *Moderada*, *Substancial* ou *Perfeita*. Por outro lado, uma concordância fraca foi observada quando o valor de *Kappa* está relacionado com as três categorias restantes: *Pobre*, *Ligeira* e *Justa*.

Para responder à questão de pesquisa **Q2**, foram investigados se alguns fatores puderam influenciar em uma maior concordância entre os desenvolvedores. Nesse contexto, foram realizadas análises que consideraram os fatores coletados durante o experimento que estão descritos na Tabela 3.3. Assim, a concordância foi avaliada a partir de um subgrupo de desenvolvedores definido por cada fator analisado. O procedimento para a criação desses subgrupos está descrito na Seção 3.2.

Tabela 3.3: Fatores de influência investigados no estudo

#	Fator	Descrição
1	Atuação na academia	indica se o desenvolvedor é da academia
2	Atuação na indústria	indica se o desenvolvedor é da indústria
3	Exper. com Desenv. de Software	indica (em uma escala de 1 a 10) a experiência do desenvolvedor em desenvolvimento de software
4	Experiência com Java	indica (em uma escala de 1 a 10) a experiência do desenvolvedor com a linguagem de programação Java
5	Experiência com *Bad Smells*	indica (em uma escala de 1 a 10) a experiência do desenvolvedor com a detecção de *bad smells*
6	Heurística de Detecção	indica a heurística extraída a partir das questões abertas

3.1.6 Execução e Preparação dos Dados

A execução do experimento ocorreu durante um período de 2 meses com início na segunda semana de janeiro, no ano de 2016. Ao todo, 98 desenvolvedores foram convidados e 75 destes completaram todas as fases do estudo e foram considerados no estudo. Após coletar os dados produzidos durante a execução, foi realizada uma série de procedimentos a fim de garantir a qualidade dos dados analisados no experimento. Nesse contexto, as avaliações dos desenvolvedores foram avaliadas, juntamente com todas as respostas às questões abertas, visando identificar qualquer problema que poderia prejudicar a análise dos dados.

A partir das respostas, foi observado que um desenvolvedor indicou que não havia compreendido a definição do *bad smell Inappropriate Intimacy* e, por isso, respondeu **NÃO** em todas as avaliações relacionadas a esse *smell*. Tais avaliações foram desconsideradas da análise dos dados. Em outro caso, um desenvolvedor atribuiu respostas vazias para todas as questões abertas e respondeu **NÃO** para todas as avaliações realizadas por ele. Em contato realizado após o experimento, esse desenvolvedor indicou que estava muito ocupado com outras atividades e não teve tempo suficiente para realizar o experimento de forma adequada. Esse desenvolvedor também foi desconsiderado do experimento. Por fim, outros desenvolvedores substituíram as avaliações desconsideradas nos casos reportados.

Ao final do experimento foram obtidas respostas de 75 desenvolvedores com diferentes experiências em desenvolvimento de software, linguagem de programação Java e detecção de *bad smells*. Em uma escala de 1 a 10, as experiências dos participantes estão ilustradas na Figura 3.2. Nesse contexto, percebeu-se uma variação nos níveis de experiência (eixo x) onde uma grande maioria indicou experiência maior que 3. Tais fatos permitiu a análise dos dados considerando desenvolvedores com diferentes níveis de experiência e com, pelo menos, certo nível de conhecimento sobre o contexto investigado.

3.2 Resultados

Nas seções seguintes são apresentados os principais resultados que ajudam a responder as questões de pesquisa definidas no estudo detalhado nesse capítulo. Na Seção 3.2.1, é apresentada a concordância aferida entre as avaliações dos desenvolvedores que participaram do estudo. Tais resultados ajudam a responder à questão de pesquisa **Q1**. Os resultados des-

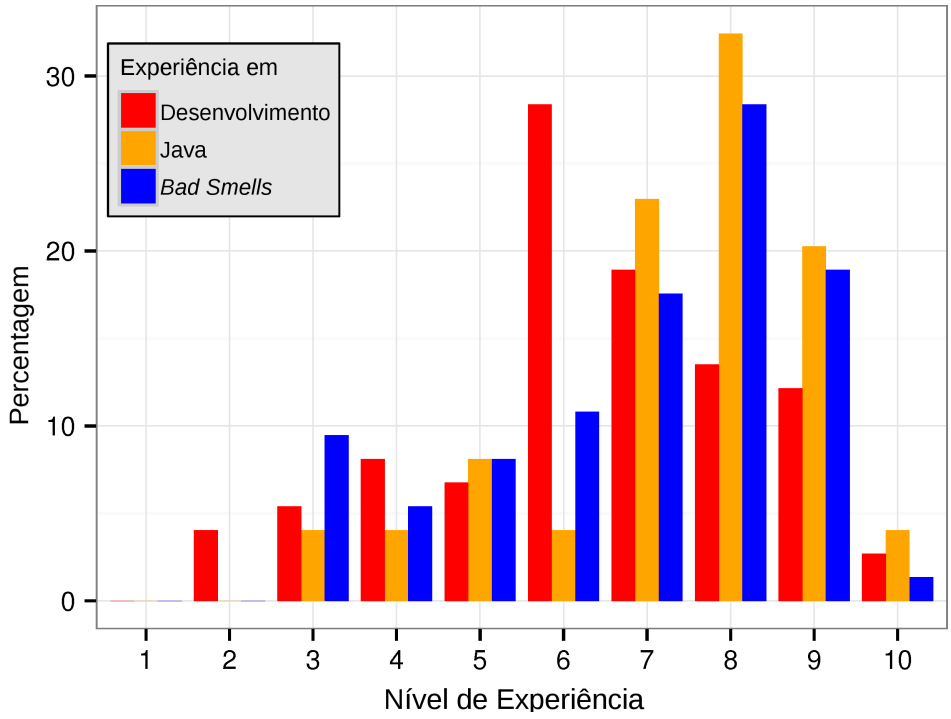


Figura 3.2: Níveis de experiência dos desenvolvedores

critos nas seções seguintes ajudam a responder a questão **Q2**, uma vez que eles descrevem a concordância entre os desenvolvedores considerando os diferentes fatores relacionados a eles, tais como *atuação* (Seção 3.2.2), *experiências* (Seção 3.2.3) e *heurísticas* reportadas nas questões abertas (Seção 3.2.5). Além disso, na Seção 3.2.5, são investigados quais desses fatores são predominantes entre os desenvolvedores que apresentaram maior similaridade em suas avaliações.

3.2.1 Concordância entre Desenvolvedores

Nesta seção são descritas as concordâncias aferidas a partir das avaliações dos desenvolvedores sobre um mesmo conjunto de trechos de código. Considerando o procedimento descrito na Seção 3.1.3, para cada tipo de *bad smell* um grupo de 12 desenvolvedores avaliou um conjunto de 15 trechos de código. Cada desenvolvedor reportou se cada trecho de código continha o *bad smell* analisado pelo grupo. Tais avaliações foram utilizadas para aferir a concordância entre esses desenvolvedores.

Tomando como exemplo as avaliações do grupo responsável por avaliar a presença do *bad smell Long Method*, a Tabela 3.4 apresenta como cada um dos 12 desenvolvedores (iden-

tificados nas colunas) avaliaram os 15 trechos de código apresentados (indicados nas linhas). Cada célula interna da Tabela representa a avaliação (**SIM** ou **NÃO**) dada como resposta de cada desenvolvedor à seguinte pergunta: *O trecho de código indicado contém uma instância do bad smell Long Method?*

Tabela 3.4: Avaliações dos desenvolvedores na detecção de *Long Method*

Trechos de Código	Desenvolvedores											
	41	42	43	45	46	47	49	51	64	74	81	95
#1	■	■	■	■	■	■	■	■	■	■	■	■
#2	■	■	■	■	■	■	■	■	■	■	■	■
#3	■	■	■	■	■	■	■	■	■	■		■
#4	■	■	■	■	■	■	■	■	■	■		■
#5	■	■		■	■	■	■	■	■	■	■	■
#6		■	■	■	■	■		■	■	■	■	■
#7								■			■	
#8	■	■	■		■	■	■	■	■	■		
#9	■	■	■	■	■	■			■	■		
#10		■							■		■	■
#11	■	■	■	■		■					■	
#12		■		■		■			■			
#13	■	■							■		■	
#14	■			■		■						■
#15								■		■	■	

Células em cinza representam respostas SIM, enquanto as células em branco representam respostas NÃO.

As avaliações apresentadas na Tabela 3.4 indicam que todos os desenvolvedores concordam que os trechos de código #1 e #2 contêm o *bad smell Long Method*. Além disso, para os trechos de código #3, #4 e #5 apenas um desenvolvedor reportou uma resposta diferente dos outros. Para todos os trechos de código restantes, os desenvolvedores apresentaram 2 ou mais diferenças em suas avaliações. Considerando as colunas dos desenvolvedores, pode-se observar que nenhum par de desenvolvedores apresentou as mesmas avaliações para os 15 trechos de código avaliados. Nesse sentido, os desenvolvedores {45;47} realizaram avaliações similares, exceto para o trecho de código #8. Similarmente, os pares de desenvolvedores {42;64} e {46;74} também apresentaram apenas uma resposta diferente ao avaliar os trechos de código #11 e #15, respectivamente.

Para as avaliações relacionadas ao *bad smell Long Method* foi calculado o valor de *Kappa* visando aferir a concordância entre os desenvolvedores. Com esse valor foi possível verificar o nível de concordância a partir das categorias descritas na Tabela 3.2. Esse procedimento

foi repetido para todos os tipos de *bad smell* investigados no estudo.

Na Figura 3.3 são ilustrados os valores de *Kappa* obtidos para cada tipo de *bad smell* analisado no estudo, bem como o nível de concordância relacionado a cada valor. Nesse caso, os valores de *Kappa* estão anexados ao lado da barra associada com cada tipo de *bad smell*, enquanto o nível de concordância está representado através das colunas em cinza que estão destacadas ao fundo da figura. Em todo o estudo, apenas valores de *Kappa* que são estatisticamente significantes são apresentados, considerando um $p\text{-}value < 0.05$. Nesse caso, os *p-value* associados com os valores de *Kappa* apresentados na Figura 3.3 variaram entre 0.000 e 0.041.

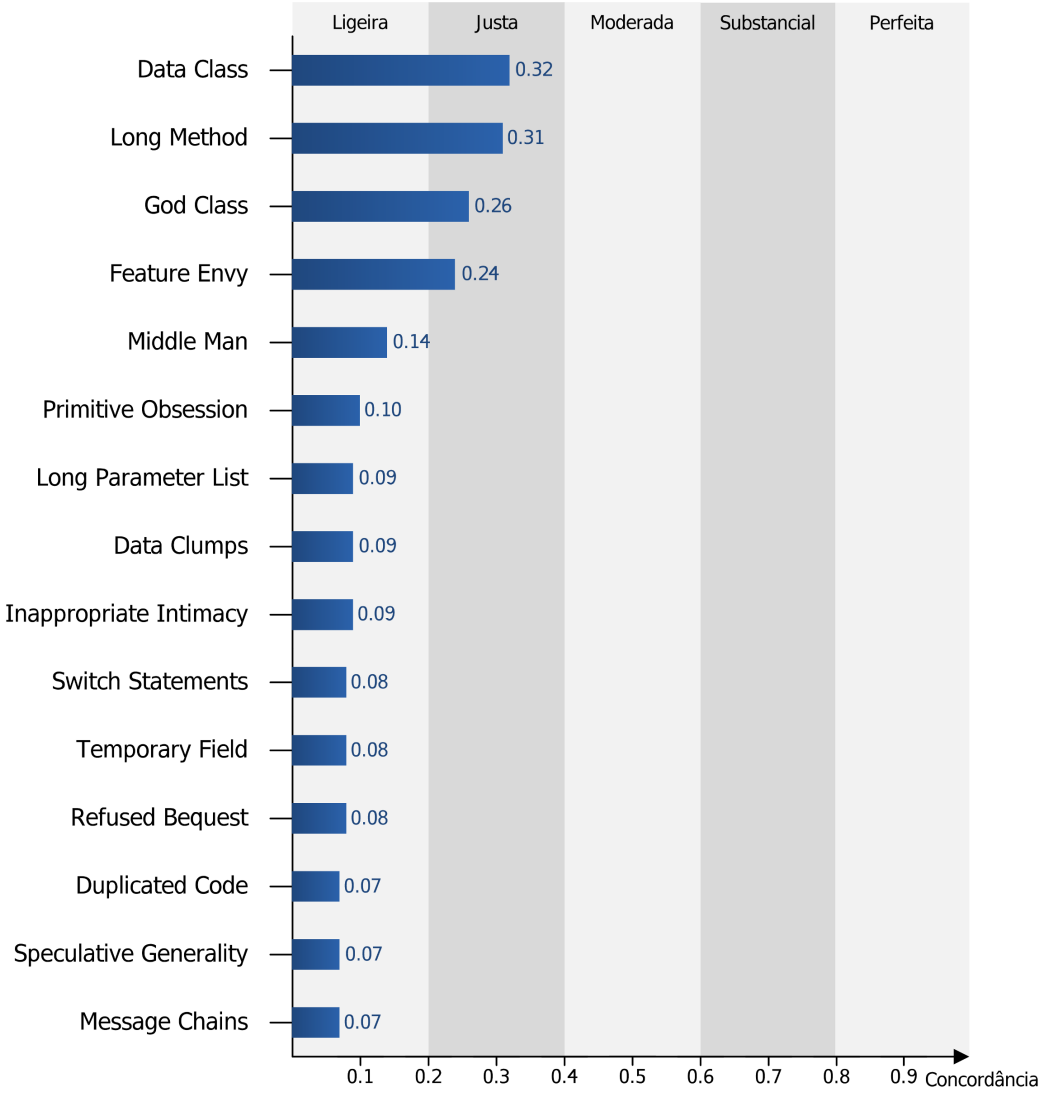


Figura 3.3: Concordância entre desenvolvedores para cada tipo de *bad smell*

De acordo com os resultados apresentados na Figura 3.3, observa-se que os desenvolve-

dores alcançaram os maiores valores de *Kappa* nas avaliações dos *bad smells Data Class*, *Long Method*, *God Class* e *Feature Envy*. Entretanto, estes valores indicam níveis de concordância ainda muito baixos. Afinal, nesses, casos, o *Kappa* aferido apresentou valores que variaram entre 0.24 e 0.32, não alcançando níveis de concordância maiores que atingissem uma categoria *Moderada* ou superior. Por exemplo, o valor de *Kappa* aferido para as avaliações de *Long Method* apresentou um dos quatro melhores valores que atingiram um nível de concordância *Justa*. Não obstante, conforme apresentado na Tabela 3.4, é possível observar que mesmo nesse caso houve (i) uma discordância consistente entre os pares de desenvolvedores que avaliaram esse *bad smell*, e (ii) uma discordância consistente entre a maioria dos trechos de código avaliados. Com exceção de *Data Class*, a concordância aferida para os outros *bad smells* apresentou valores ainda piores do que o aferido para *Long Method*. Na maioria dos casos os valores aferidos apresentaram concordância com nível em uma categoria pior que *Justa*. Esses casos apresentaram uma concordância *Ligeira* com valores de *Kappa* abaixo de 0.15, onde a maioria desses valores ficaram abaixo de 0.10.

Os resultados obtidos sugerem que os desenvolvedores tendem a discordar ao detectar *bad smells* uma vez que os níveis de concordância entre as suas avaliações foram muito baixos, não alcançando concordância *Substancial* nem *Moderada*. Essa tendência pode ser influenciada por fatores envolvidos na detecção dos *bad smells*, que incluem a atuação e experiência dos desenvolvedores. Além disso, a ausência (e talvez a impossibilidade) de uma definição formal para os *bad smells*, podem contribuir que os desenvolvedores detectem um mesmo tipo de *bad smell* utilizando formas diferentes. Consequentemente, os desenvolvedores tendem a discordar em suas avaliações. Nas seções seguintes a concordância entre os desenvolvedores é analisada a partir dos fatores descritos na Tabela 3.3. Os resultados apresentados nessas seções permitem investigar se esses fatores podem levar os desenvolvedores a alcançarem uma maior concordância em suas avaliações.

3.2.2 Concordância a partir dos Fatores de Atuação dos Desenvolvedores

Conforme descrito na Seção 3.1.3, os participantes do experimento foram enquadrados segundo a sua atuação, indicando uma participação mais efetiva na *Academia* ou na *Indústria*.

Tal informação foi utilizada para investigar se os desenvolvedores de mesma atuação apresentaram uma concordância mais elevada.

Na análise descrita na seção anterior, para cada tipo de *bad smell* analisado no estudo, foi calculada a concordância entre o grupo de 12 desenvolvedores responsáveis por avaliar tal **smell**. Nessa seção, visando investigar a influência da atuação dos desenvolvedores na concordância entre suas avaliações, os participantes do mesmo grupo foram divididos em 2 subgrupos. O primeiro subgrupo foi composto apenas com desenvolvedores que apresentaram uma atuação mais efetiva na *Academia*. Já o segundo subgrupo reuniu os desenvolvedores com atuação na *Indústria*. Em seguida, foi calculada a concordância entre as avaliações dos desenvolvedores pertencentes a cada subgrupo. Finalmente, a concordância aferida para os subgrupos foi comparada com a concordância observada no grupo com 12 desenvolvedores, ilustrada na Figura 3.3. Tal comparação é importante para verificar se o fator de atuação dos desenvolvedores pode influenciar em uma maior concordância quando comparados com o grupo geral.

Fator: Atuação do desenvolvedor - Academia

A Figura 3.4 ilustra a concordância observada nas avaliações feitas pelos desenvolvedores com atuação na *Academia*. Na figura foi utilizada uma *linha vertical laranja* para representar a concordância geral observada na Figura 3.3. A representação da concordância geral irá ajudar a analisar se os desenvolvedores da *Academia* apresentam uma concordância maior que aquela aferida de maneira geral envolvendo todos os desenvolvedores. Além disso, a fim de identificar o nível de cada concordância, são apresentadas no topo da figura as classificações descritas na Tabela 3.2.

De acordo com os resultados coletados é possível observar que os desenvolvedores da *Academia* obtiveram uma concordância que variou de 0.14 até 0.36. Enquanto o nível de concordância relacionado aos *smells Data Clumps* e *Long Method* está classificado como *Ligeiro*, os demais *bad smells* apresentaram uma concordância *Justa*.

Com exceção de *God Class* e *Feature Envy*, foi percebida uma diferença significante entre os valores de concordância entre os desenvolvedores da *Academia* e em geral. De fato, houve um decremento na concordância relacionada à detecção do *smell Long Method* quando comparada à concordância correspondente aferida para todos os desenvolvedores.

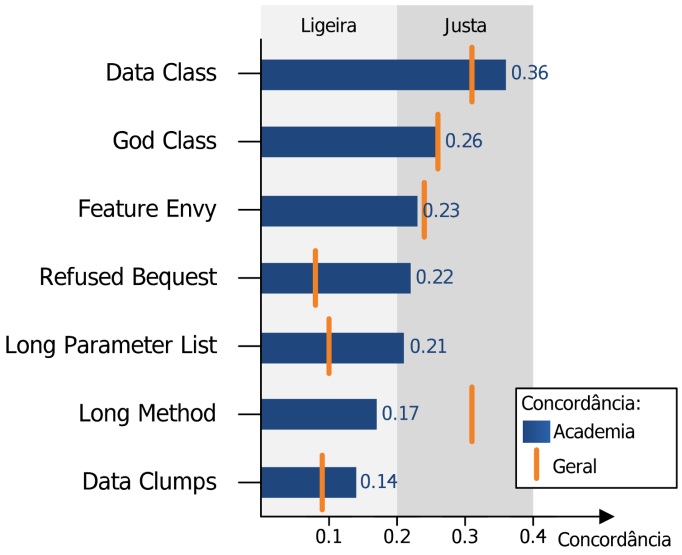


Figura 3.4: Concordância a partir do fator de atuação - Academia

Nesse caso houve, inclusive, uma mudança no nível da concordância que decresceu de *Justa* para *Ligeira*. Este resultado sugere que os desenvolvedores da *Academia* tendem a discordar mais entre si do que quando juntos aos demais desenvolvedores ao avaliar a presença do *smell Long Method*.

Por outro lado, os resultados também apontaram que os desenvolvedores da *Academia* concordaram mais ao detectar os *smells Data Class*, *Data Clumps*, *Refused Bequest* e *Long Parameter List*. De fato, considerando as classificações descritas na Tabela 3.2, os desenvolvedores da *Academia* apresentaram melhores níveis de concordância do que os desenvolvedores em geral ao detectarem os *smells Refused Bequest* e *Long Parameter List*. Esses resultados indicam que os desenvolvedores da Academia tendem a concordar mais nesses *smells* quando comparados aos desenvolvedores em geral.

Em resumo, os resultados indicaram que os desenvolvedores da *Academia* tendem a concordar mais do que os desenvolvedores em geral para a maioria dos bad *smells* indicados na Figura 3.4. De fato, para quatro tipos de *smells* (*Data Class*, *Refused Bequest*, *Long Parameter List* e *Data Clumps*) os desenvolvedores da *Academia* apresentaram uma concordância mais elevada do que os desenvolvedores em geral. Para os demais tipos de *bad smell*, apenas a detecção de *Long Method* apresentou um decremento significante da concordância dos desenvolvedores da *Academia* quando comparados com todos os desenvolvedores que avaliaram esse *smell*. Este resultado foi ligeiramente surpreendente, uma vez que se esperava a

detecção desse *smell* apresentasse uma maior concordância do que outros como *God Class* e *Data Class*. Afinal, o escopo do *bad smell Long Method* está associado a um método de uma classe, enquanto os *smells God Class* e *Data Class* afetam uma classe inteira, que pode ser composta por diversos métodos. Desse modo, esperava-se que o escopo reduzido em que os *Long Methods* são observados favorecesse uma opinião uniforme durante suas avaliações. Por fim, também foi observado que mesmo nos casos em que os desenvolvedores da *Academia* tenham alcançado uma concordância mais elevada que os desenvolvedores em geral na maioria dos *smells*, os níveis de concordância atingidos ainda apresentaram valores baixos, variando entre as categorias *Ligeira* e *Justa*.

Fator: Atuação do desenvolvedor - Indústria

A Figura 3.5 ilustra a concordância observada nas avaliações realizadas pelos desenvolvedores com atuação na *Indústria*. Nota-se que tais desenvolvedores alcançaram uma concordância *Ligeira* nas avaliações relacionadas aos *smells God Class* e *Temporary Field*, uma concordância *Justa* nas avaliações de *Data CLass* e *Feature Envy*, e, finalmente, uma concordância *Moderada* nas avaliações de *Long Method*.

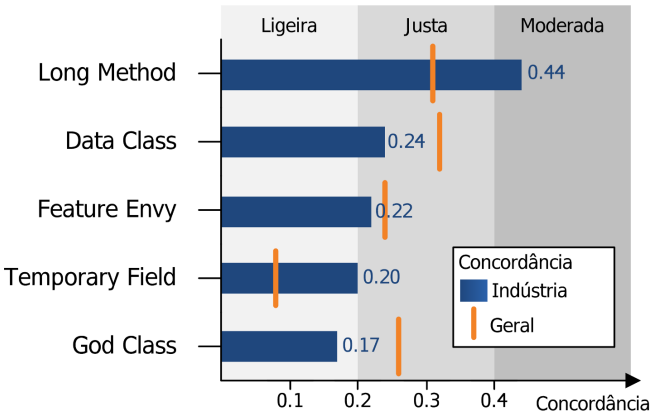


Figura 3.5: Concordância a partir do fator de atuação - Indústria

Além disso, quando se analisa apenas os casos relacionados aos *smells Long Method* e *Temporary Field*, é percebido um acréscimo na concordância. Nesses casos, o maior aumento foi observado para o *smell Long Method*, onde o nível de concordância passou de *Justo* (para os desenvolvedores em geral) para *Moderado*. Entretanto, os resultados também indicaram um decréscimo na concordância relacionada aos *smells God Class*, *Feature Envy*

e *Data Class*. Em particular, verificou-se que o nível de concordância das avaliações de *God Class* passou de *Justo* para *Ligeiro*.

Portanto, os resultados analisados não apresentou evidências que a atuação na *Indústria* pode ajudar os desenvolvedores a alcançar uma maior concordância ao detectar *bad smells*. A influência desse fator permitiu tanto um acréscimo quando um decréscimo da concordância quando comparada com os valores aferidos para os desenvolvedores em geral. Por outro lado, os valores aferidos para os desenvolvedores da *Indústria* apresentaram uma interessante relação com os resultados aferidos com os desenvolvedores da *Academia*. Enquanto os desenvolvedores da *Indústria* tenderam a discordar ao analisar os casos de *Data Class*, os acadêmicos alcançaram sua maior concordância ao avaliar esse *smell*. Esse cenário sugere que os desenvolvedores da *Indústria* divergem mais que os desenvolvedores da *Academia* sobre o que constitui um smell de *Data Class*.

Por fim, os resultados relacionados a *Long Method* também apresentaram tendências opostas. Enquanto os desenvolvedores da *Academia* apresentaram uma tendência em discordar nas avaliações, os desenvolvedores com atuação na *Indústria* alcançaram sua maior concordância, atingindo um nível *Moderado*. Tal resultado pode ter sido influenciado pelas regras de estilo de código que muitas vezes são impostas nas companhias. Assim, os desenvolvedores da *Indústria* podem ter analisados os trechos de código seguindo tais regras.

3.2.3 Concordância dos Desenvolvedores a partir de Fatores Relacionados à Experiência

Após analisar se a atuação dos desenvolvedores pode influenciar na concordância entre eles, investigou-se uma relação similar entre a concordância e a experiência dos desenvolvedores. Nesse caso, foi analisada a concordância entre os desenvolvedores mais experientes que compuseram cada grupo de 12 desenvolvedores. A experiência de cada desenvolvedor foi definida levando em consideração auto-avaliação reportada por eles. Nesse caso, os desenvolvedores considerados como mais experientes foram aqueles que apresentaram um nível de experiência maior que a mediana dos 12 desenvolvedores que avaliaram um mesmo tipo de *bad smell*. Esse procedimento permitiu a criação de subgrupos com, pelo menos, quatro desenvolvedores experientes, os quais permitiram a investigação da concordância a partir de

suas avaliações.

Fator: Experiência - Desenvolvimento de Software

A Figura 3.6 descreve a concordância entre os desenvolvedores mais experientes com desenvolvimento de software. Foi verificado que tais desenvolvedores alcançaram níveis de concordância que variaram de 0.13 a 0.43, onde a maior concordância alcançou um nível *Moderado*.

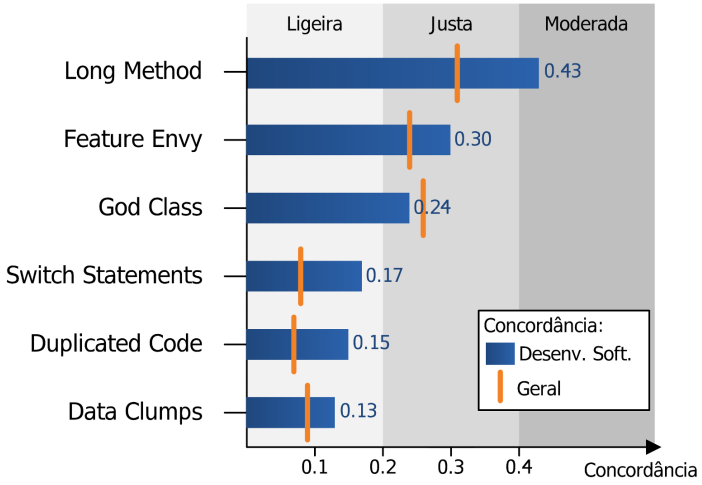


Figura 3.6: Concordância a partir do fator de experiência - Desenvolvimento de Software

Apenas para o *bad smell God Class*, foi verificado um decréscimo na concordância quando comparada com os valores obtidos por todos os desenvolvedores. Entretanto, esse decréscimo não foi suficiente para mudar o nível de concordância. Para os demais tipos de *bad smells*, verificou-se um incremento na concordância. Em particular, para *Long Method*, a concordância cresceu da categoria *Justa* para *Moderada*, alcançando o melhor resultado para essa análise.

Para os *bad smells* ilustrados na Figura 3.6 observou-se uma tendência positiva: os desenvolvedores mais experientes tenderam a incrementar os níveis de concordância quando comparados com os valores obtidos com todos os desenvolvedores. Entretanto, uma melhora significativa foi observada em apenas um caso, com o *smell Long Method*, onde os desenvolvedores experientes alcançaram uma concordância *Moderada*. Os níveis de concordância calculados para os demais *smells* apresentaram valores ruins, com níveis de categoria *Justa* ou *Ligeira*.

Fator: Experiência - Linguagem Java

O estudo realizado também analisou a concordância obtida ao considerar desenvolvedores com experiência na linguagem de programação *Java*, conforme ilustrado na Figura 3.7. Nessa análise os desenvolvedores mais experientes apresentaram níveis de concordância muito baixos, que variou entre as categorias *Ligeira* e *Justa*.

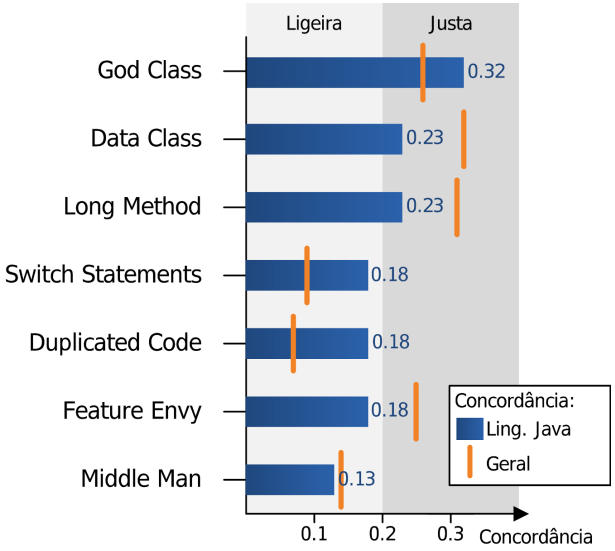


Figura 3.7: Concordância a partir do fator de experiência - Linguagem Java

Embora tenha sido observado um decréscimo na concordância relacionada aos *bad smells* *Data Class*, *Long Method*, *Feature Envy* e *Middle Man*, os resultados apontaram um acréscimo na concordância dos *smells God Class*, *Switch Statements* e *Duplicated Code*. Portanto, não foi possível verificar um resultado consistente que evidencie que a experiência em *Java* pode influenciar na concordância entre os desenvolvedores. Em outras palavras, o fator relacionado à experiência na linguagem *Java* não apresentou evidências que indiquem uma maior concordância entre os desenvolvedores.

Fator: Experiência - Detecção de *Bad Smells*

Finalmente, foi analisada a influência da experiência em detecção de *bad smells* na concordância dos desenvolvedores. A Figura 3.8 ilustra a concordância obtida a partir das avaliações dos desenvolvedores que apresentaram maior experiência na detecção de *bad smells*.

De acordo com os resultados, observa-se um decréscimo na concordância relacionada aos *smells Data Class* e *God Class*. Para este último, o nível de concordância mudou da

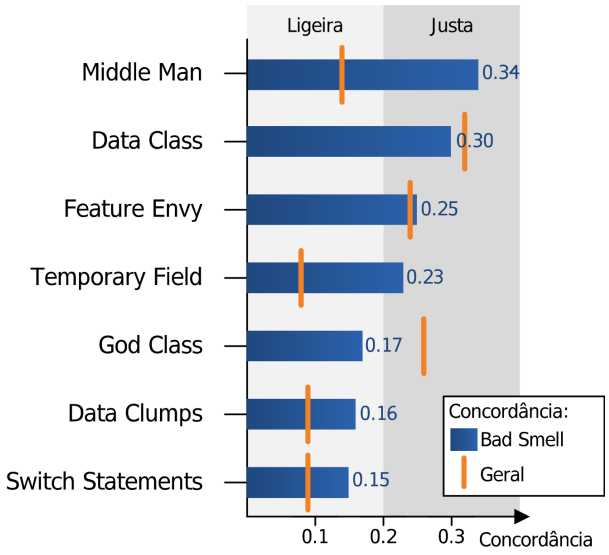


Figura 3.8: Concordância a partir do fator de experiência - *Bad Smell*

categoria *Justa* para *Ligeira*. Para os demais *smells* da Figura 3.8, notou-se um acréscimo na concordância aferida, principalmente para os casos relacionados aos *smells Middle Man* e *Temporary Field*, onde o nível de concordância mudou da categoria *Ligeira* para *Justa*.

Os resultados apresentados indicaram que não existiu uma uniformidade na influência da experiência em *bad smells* na concordância. Pode ser observado que enquanto os desenvolvedores experientes nesse fator apresentaram uma concordância menor para os casos de *God Class* e *Data Class*, a concordância para os demais *smells* foi acrescida. Considerando que estes desenvolvedores compõem o grupo mais experiente em detecção de *bad smells*, esperava-se que eles pudessem alcançar uma concordância melhor para todos os casos analisados, incluindo aqueles que afetam classes inteiras (*God Class* e *Data Class*). Não obstante, os resultados contraditaram a expectativa ao revelar uma concordância baixa para os *smells* ilustrados na Figura 3.8 quando comparados com a concordância geral. Além disso, percebeu-se que a maioria dos casos onde a concordância foi mais elevada estão relacionado a *smells* que afetam um escopo reduzido. De fato, o *smell Switch Statement* afeta apenas uma estrutura do tipo *switch/case*, *Data Clumps* afeta um conjunto de parâmetros ou atributos e *Temporary Field* está relacionado a um único atributo. Por outro lado, os casos onde a concordância foi menor estão relacionados a classes. Talvez o número reduzido de linhas nos trechos de código analisados tenha favorecido uma avaliação mais uniforme entre os desenvolvedores.

3.2.4 Concordância a partir das Heurísticas dos Desenvolvedores

Além de analisar a *atuação* (Seção 3.2.2) e a *experiência* (Seção 3.2.3) dos desenvolvedores, o estudo investigou se as heurísticas, reportadas pelos desenvolvedores ao detectar os *bad smells*, influenciaram a concordância. Tais heurísticas, foram extraídas a partir das respostas dos desenvolvedores às *questões abertas* providas durante o experimento, como descrito na Seção 3.1.4. Nesse cenário, quatro especialistas em *bad smells* (dois estudantes de doutorado e dois estudantes de mestrado) aplicaram uma técnica de *coding* [78] para analisar as respostas dos desenvolvedores. Esses especialistas realizam pesquisas sobre *bad smells* a mais de dois anos e incluem a análise de várias anomalias em diversos contextos. A análise realizada pelos especialistas foi conduzida com o objetivo de reconhecer as heurísticas adotadas pelos desenvolvedores ao detectar os *smells* investigados.

Por exemplo, a partir das avaliações relacionadas à detecção do *smells Long Method*, os especialistas reconheceram três diferentes heurísticas de detecção: (*H1*) essa heurística considera apenas o *número de linhas de código em um método* para julgar se o mesmo é uma instância de *Long Method*. Nesse caso as respostas dos desenvolvedores foram similares a *Eu considerei métodos com um número de linhas maior que 30*; (*H2*) nessa heurística um *Long Method* é detectado ao considerar o tamanho e a estrutura do método analisado. Nesses casos, as respostas dos desenvolvedores foram similar a *Eu verifiquei o número de linhas de código e o número de estruturas de controle*; (*H3*) nessa heurística considera que um método é uma instância de *Long Method* se o mesmo pode ser dividido em métodos menores. Tal heurística representa respostas similares a *Eu olhei métodos que poderiam ser divididos em métodos menores*. Nessa heurística os respondentes não indicaram nenhum atributo específico mensurável nem uma estrutura da linguagem.

Na Tabela 3.5 estão descritas as três heurísticas reconhecidas pelos especialistas identificando os desenvolvedores que reportaram cada heurística. Pode-se observar que 6 desenvolvedores detectaram o *smell* considerando a heurística *H1*. Três desenvolvedores consideraram a heurística *H2* e outros três desenvolvedores detectaram *Long Methods* utilizando a heurística *H3*.

O mesmo procedimento também foi aplicado pelos especialistas com o objetivo de reconhecer as heurísticas relacionadas aos outros tipos de *bad smell*. Na Tabela 3.6 é indicado o número de heurísticas reconhecidas pelos especialistas para cada tipo de *smell*. Nota-se que

Tabela 3.5: Heurísticas para a detecção de *Long Method* reportadas nas questões abertas

#	Heurística	Desenvolvedores (ID)
H1	Análise exclusiva do tamanho do método	41, 42, 46, 51, 64, 95
H2	Análise do tamanho e das estruturas de controle de um método	47, 49, 74
H3	Análise sobre quando um método pode ser dividido em métodos menores	43, 45, 81

os desenvolvedores reportaram de 3 (como para os casos de *Middle Man* e *Long Method*) até 11 (como no caso de *Temporary Field*) diferentes heurísticas para detectar os *smells*. Este fato indica que, mesmo apresentando apenas uma única definição de cada *smell*, os desenvolvedores detectaram um mesmo tipo de *smell* de diferentes formas. Mesmo para *smells* que afetam um escopo limitado no código, como *Temporary Field* (atributo) e *Switch Statements* (comando *switch/case*), os desenvolvedores reportaram um elevado número de heurísticas (11 e 8, respectivamente). Particularmente, o caso mais inesperado está relacionado ao fato do *smell Temporary Field*, onde 12 desenvolvedores reportaram 11 diferentes heurísticas de detecção.

Tabela 3.6: Número de Heurísticas reconhecidas para cada *bad smell*

Bad Smell	Número de Heurísticas
Data Class	4
Primitive Obsession	5
Long Method	3
God Class	9
Middle Man	3
Long Parameter List	4
Data Clumps	8
Switch Statements	8
Refused Bequest	5
Duplicated Code	3
Message Chains	3
Feature Envy	4
Temporary Field	11
Inappropriate Intimacy	9
Speculative Generality	8

Para cada tipo de *bad smell*, foi verificada a heurística mais citada e, a partir dela, verificou-se a concordância entre as avaliações dos desenvolvedores que a reportaram. Por exemplo, considerando novamente o *smell Long Method*, a heurística *H1* foi a mais citada

pelos desenvolvedores, uma vez que foi reportada nas respostas dos desenvolvedores 41, 42, 46, 51, 64 e 95. A partir das avaliações desses desenvolvedores foi observada uma concordância igual a 0.45 (*Moderada*). Esse foi o maior valor de *Kappa* se comparado com os valores correspondentes derivados das análises descritas nas seções anteriores. Após aplicar o mesmo procedimento para os outros *smells*, foram obtidos os níveis de concordância ilustrados na Figura 3.9.

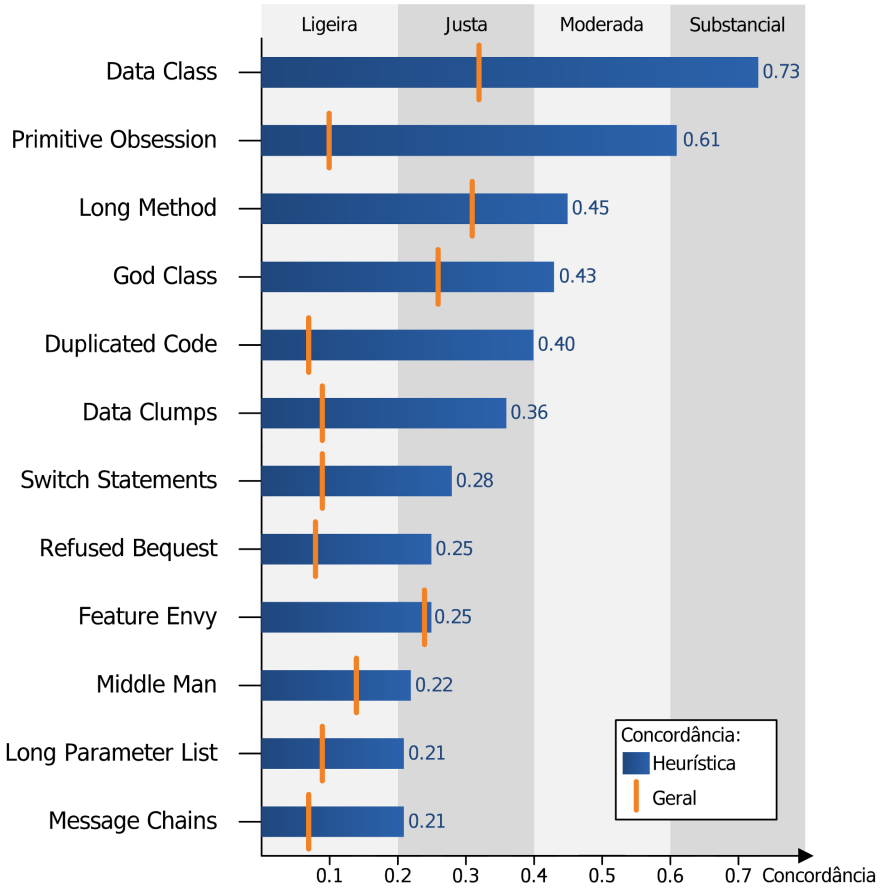


Figura 3.9: Concordância dos desenvolvedores que reportaram a heurística mais citada

Ao analisar os níveis de concordância ilustrados na Figura 3.9, observa-se um consistente acréscimo na concordância aferida quando comparada com a amostra geral. De fato, os níveis de concordância foram maiores para todos os 12 tipos de *bad smell* onde foi possível obter um valor de *Kappa* estatisticamente significante. Nesse contexto apenas a concordância relacionada aos *smells Temporary Field*, *Inappropriate Intimacy* e *Speculative Generality* não puderam ser calculadas com a devida significância estatística. O alto número de heurísticas envolvidas na detecção desses *smells* dificultaram a obtenção de valores estatisticamente

significantes para a concordância aferida.

Já para os casos apresentados, é possível observar que, na maioria dos casos, o resultado aferido permitiu uma mudança no nível da concordância. Por exemplo, enquanto os desenvolvedores que reportaram a heurística mais citada para detectar *Data Class* e *Primitive Obsession* alcançaram uma concordância *Substancial*, a concordância em geral para esses *smells* foram aferidas como *Justa* e *Ligeira*, respectivamente. Da mesma forma, a concordância relacionada aos *smells Long Method* e *God Class* alcançaram uma concordância *Moderada*, frente à concordância *Justa* observada para os desenvolvedores em geral. A única exceção ocorreu para o *smell Feature Envy*, onde a concordância foi classificada como *Justa* em ambos os cenários. Não obstante, mesmo nesse caso, houve um incremento na concordância de 0.24 (em geral) para 0.25 (desenvolvedores que reportaram a heurística mais citada). Portanto, entre todos os fatores analisados, o uso comum da heurística mais citada apresentou a maior influência no aumento da concordância.

Embora os desenvolvedores que seguram uma mesma heurística tenham apresentado um aumento significativo na concordância, tal resultado poderia ser ainda melhor se fosse analisado como cada desenvolvedor aplicou a heurística adotada. Afinal, uma heurística pode ser operacionalizada de diferentes formas pelos desenvolvedores. Por exemplo, [60; 51; 66] propõem "estratégias de detecção" para implementar heurísticas através de regras compostas por métricas e limiares. Nesse contexto, assumindo que os desenvolvedores 42 e 46 (que reportaram a mesma heurística para detectar *Long methods* de acordo com o tamanho do método) fizessem suas avaliações analisando métrica *LOC*[2] (que indica o número de linhas de código), não é possível garantir que eles adotaram o mesmo limiar para determinar uma instância de *Long Method*. De fato, analisando as avaliações desses desenvolvedores, percebeu-se que o desenvolvedor 46 considerou como *Long Method* todos os métodos com mais de 92 linhas de código, enquanto o desenvolvedor 42, indicou como *smell* todos os métodos com mais de 56 linhas. Tal critério subjetivo, onde desenvolvedores podem adotar diferentes heurísticas, métricas e limiares ao detectar *bad smells*, tem um impacto direto na concordância entre os desenvolvedores.

[2] *Lines of Code* - Linhas de Código

3.2.5 Fatores presentes nos Agrupamentos de Desenvolvedores

Após analisar a influência da *atuação* e *experiência* dos desenvolvedores, assim como as *heurísticas* utilizadas por eles para detectar *bad smells*, foi realizada uma outra análise com o objetivo de verificar quais fatores são mais predominantes entre os agrupamentos de desenvolvedores que apresentaram as avaliações mais similares. Como resultado dessa análise, espera-se confirmar se os fatores analisados nas seções anteriores também estão presentes nesses agrupamentos.

A execução dessa análise aconteceu com o seguinte procedimento. Primeiro, para cada *bad smell*, foi gerada uma matriz representando as avaliações dos 12 desenvolvedores sobre os 15 trechos de código relacionados ao *smell*. A partir dessa matriz foi produzido um diagrama de agrupamentos hierárquicos conhecido como *dendograma* para identificar o nível de similaridade entre as avaliações dos desenvolvedores. Para isso, foi utilizada a ferramenta estatística R[3] na geração dos dendogramas a partir das avaliações dos desenvolvedores para cada tipo de *smell* analisado.

Por exemplo, a matriz apresentada na Tabela 3.4 representa as avaliações dos 12 desenvolvedores sobre os 15 trechos de código relacionados ao *smell Long Method*. Tal matriz foi usada como entrada da ferramenta *R* gerando o dendograma ilustrado na Figura 3.10.

De acordo com o dendograma produzido, as *folhas* representam os desenvolvedores (indicados por um número identificador) e os valores descritos no *eixo-y* representam a distância em termos do máximo número de diferenças entre as avaliações realizadas pelos desenvolvedores pertencentes aos agrupamentos vizinhos. Por exemplo, o agrupamento composto pelos desenvolvedores {45;47} diferem em apenas uma avaliação das 15 realizadas.

A geração dos dendogramas ajudou a identificar os agrupamentos de desenvolvedores com as menores diferenças entre suas avaliações. Nesse sentido, foi possível verificar se os fatores analisados nas seções anteriores também estavam presentes em tais agrupamentos. A Tabela 3.7 resume os principais resultados dessa análise.

A primeira coluna da Tabela 3.7 descreve os agrupamentos com as menores diferenças entre as avaliações dos desenvolvedores. A segunda coluna indica o número de desenvolvedores que compuseram cada agrupamento e a terceira coluna apresenta o número de avaliações diferentes para cada agrupamento. Por exemplo, no dendograma ilustrado na Fi-

[3]http://www.r-project.org/

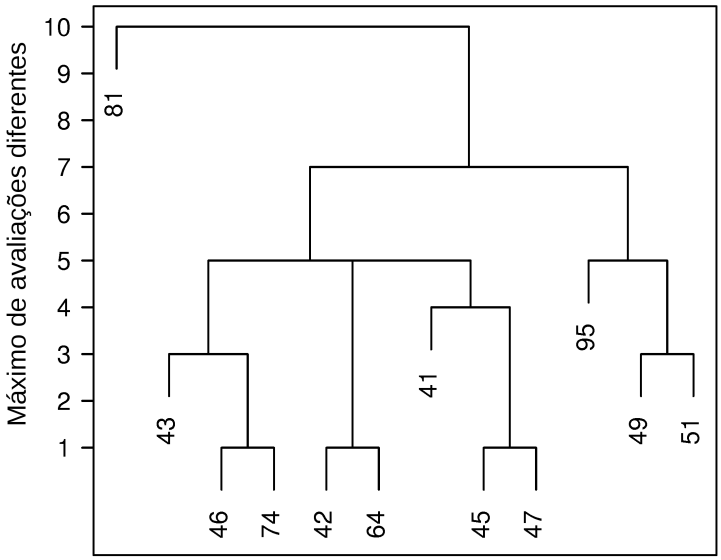


Figura 3.10: Agrupamento hierárquico baseado nas similaridades das avaliações de *Long Methods*

gura 3.10, existem três agrupamentos compostos por três pares de desenvolvedores ({46;74}, {42;64} and {45;47}). Para cada agrupamento, o dendograma representa que existiu apenas uma diferença entre as avaliações dos desenvolvedores, que foi a menor diferença entre todos os agrupamentos identificados para este *smell*. Na Tabela 3.7, esses três agrupamentos estão representados como *LM1*, *LM2* e *LM3*, respectivamente. Observa-se ainda que, assim como ocorreu para o *smell Long Method*, outros quatro *smells* apresentaram a ocorrência de múltiplos agrupamentos: *Middle Man* (2 agrupamentos), *Duplicated Code* (2 agrupamentos) e *Speculative Generality* (2 agrupamentos).

As quarta e quinta colunas indicam se os desenvolvedores que pertencem ao agrupamento identificado apresentam a mesma *atuação* (na *Academia* ou na *Indústria*, respectivamente). As três colunas seguintes reportam se os desenvolvedores do agrupamento pertencem ao grupo dos mais experientes considerando a experiência em desenvolvimento de software, Java e detecção de *bad smells*. Finalmente, a última coluna indica se os desenvolvedores do agrupamento seguiram a mesma heurística ao avaliar o *smell* correspondente.

De acordo com os resultados, observou-se que apenas em 3 dos 20 agrupamentos os desenvolvedores apresentaram uma atuação na *Academia*. O número de agrupamentos compostos por desenvolvedores da *Indústria* foi ainda menor, indicando apenas 2 agrupamentos

Tabela 3.7: Agrupamento de desenvolvedores e fatores de influência

Bad Smell	# Desenv.	Dif.	Atuação		Experiência			Heurística
			Academia	Indústria	Desenvolvimento	Java	Bad Smell	
DCl	3	1	SIM	NÃO	NÃO	NÃO	NÃO	SIM
LM1	2	1	NÃO	NÃO	NÃO	NÃO	NÃO	NÃO
LM2	2	1	NÃO	NÃO	NÃO	NÃO	NÃO	SIM
LM3	2	1	NÃO	NÃO	SIM	SIM	NÃO	NÃO
GC	2	0	NÃO	NÃO	NÃO	SIM	NÃO	NÃO
FE	2	2	NÃO	NÃO	NÃO	NÃO	NÃO	SIM
MM1	2	2	NÃO	NÃO	NÃO	NÃO	NÃO	NÃO
MM2	2	2	SIM	NÃO	NÃO	SIM	SIM	SIM
PO	3	2	NÃO	NÃO	NÃO	NÃO	NÃO	SIM
LPL	2	1	NÃO	NÃO	NÃO	NÃO	NÃO	SIM
DCm	2	0	NÃO	NÃO	NÃO	NÃO	SIM	SIM
II	2	3	NÃO	SIM	SIM	SIM	NÃO	NÃO
SS	2	1	NÃO	NÃO	NÃO	SIM	SIM	SIM
TF	2	1	NÃO	NÃO	NÃO	NÃO	SIM	NÃO
RB	2	2	NÃO	NÃO	NÃO	NÃO	NÃO	SIM
Dup1	2	0	NÃO	NÃO	SIM	SIM	SIM	SIM
Dup2	2	0	NÃO	NÃO	NÃO	NÃO	NÃO	SIM
MC	2	1	NÃO	SIM	SIM	SIM	NÃO	SIM
SG1	2	1	SIM	NÃO	NÃO	SIM	SIM	SIM
SG2	2	1	NÃO	NÃO	NÃO	NÃO	NÃO	NÃO
Total de SIM (%)			15	10	20	40	30	65

DCl-Data Class, LM-Long Method, GC-God Class, FE-Feature Envy, MM-Middle Man, PO-Primitive Obsession, LPL-Long Parameter List, DCm-Data Clumps, II-Inappropriate Intimacy, SS-Switch Statements, TF-Temporary Field, RB-Refused Bequest, Dup-Duplicated Code, MC-Message Chains, SG-Speculative Generality

que representam apenas 10% dos agrupamentos analisados. Considerando os fatores de experiência, os resultados foram ligeiramente melhores. Os desenvolvedores mais experientes em desenvolvimento de software compuseram 20% de todos os agrupamentos, enquanto os mais experientes em Java e detecção de *bad smells*, foram percebidos em 40% e 30% de todos os agrupamentos analisados, respectivamente. Finalmente, os desenvolvedores que reportaram a mesma heurística ao detectar o *smell* analisado, compuseram 65% dos agrupamentos.

Os resultados indicam que os fatores relacionados à atuação dos desenvolvedores não foram predominantes nos agrupamentos. Embora os fatores relacionados à experiência tenham sido mais predominantes que os fatores relacionados à atuação, nenhum deles chegou a estar presente em metade dos agrupamentos analisados. Por outro lado, verificou-se que o fator relacionado à heurística estava presente em 65% de todos os agrupamentos. Esse cenário

endorsa os resultado descritos na Seção 3.2.4, onde os desenvolvedores que reportaram a mesma heurística apresentaram os maiores níveis de concordância

3.3 Discussões

Nesta seção, os resultados do estudo são discutidos com o objetivo de responder suas questões de pesquisa. Além disso, nessa seção são reportados os principais achados do estudo que estão relacionados com trabalhos anteriores.

3.3.1 Q1: Os desenvolvedores concordam ao avaliar *bad smells* em código?

Os resultados descritos na Seção 3.2.1 indicam que os níveis de concordância entre as avaliações dos desenvolvedores sobre os *smells* analisados no estudo são muito baixos. De fato, os níveis de concordância em 4 (*Data Class*, *Long Method*, *God Class* and *Feature Envy*) dos 15 *smells* investigados foram classificados como *Justa* (com concordância variando entre 0.24 e 0.32), enquanto para os demais *smells* a concordância apresentou a categoria *Ligeira*, com valores variando entre 0.07 e 0.14.

Nesse contexto, estudos anteriores investigaram a concordância entre desenvolvedores que detectaram *bad smells* [56; 57]. Mäntylä [56] reportou uma alta concordância nas avaliações dos desenvolvedores relacionadas à detecção de *Long Method* e *Long Parameter List*. Baseado nesse achado, ele concluiu que tais *smells* são "simples", porque são facilmente detectados pelos desenvolvedores. O estudo apresentado em [57] reforça tal conclusão ao reportar que 5 desenvolvedores apresentaram uma concordância perfeita ao avaliarem a existência de *Long Method* em um único trecho de código.

Apesar disso, os resultados apresentados na Seção 3.2.1 verificaram uma baixa concordância entre as avaliações dos desenvolvedores em todos os *bad smells* investigados, incluindo *Long Method* e *Long Parameter List*. A concordância relacionada aos *smell Long Method* foi considerada *Justa*, com valor de *Kappa* igual a 0.31; além disso, a concordância para *Long Parameter List* foi ainda menor, apresentando valor de *Kappa* igual a 0.092, o que é classificado como uma concordância *Ligeira*. Apesar desses resultados contraditarem

algumas conclusões reportados em estudos anteriores, os mesmos foram obtidos através de um estudo mais completo, envolvendo avaliações de 12 desenvolvedores sobre 15 trechos de código diferentes. Por outro lado, o estudo reportado em [57] calculou a concordância a partir das avaliações de, no máximo, 6 desenvolvedores sobre um único trecho de código. Diferentemente do estudo apresentado na Seção 3.2.1, os autores não obtiveram resultados estatisticamente significantes ao analisar a concordância entre os desenvolvedores.

De acordo com a baixa concordância entre as avaliações dos desenvolvedores ao detectarem *bad smells*, acredita-se que seja importante criar técnicas e ferramentas que permitam a customização da detecção de acordo com o entendimento de cada desenvolvedor. Outro ponto a ser considerado está relacionado a como a performance das abordagens de detecção propostas são calculadas. A maioria dessas abordagens [45; 65; 55; 70; 72] são avaliadas em um conjunto de dados (*oráculo* contendo instâncias de *smells* validado manualmente por "especialistas" em detecção de *bad smells*. Nesses cenários, a validação de tais abordagens considera as respostas desses especialistas como a verdade a ser seguida. Portanto, tais abordagens tendem a ser definidas de acordo com o ponto de vista desses "especialistas" e ignoram o entendimento de outros desenvolvedores sobre *bad smells*.

3.3.2 Q2: O que faz os desenvolvedores concordarem ao avaliar *bad smells* em código?

Os resultados descritos na Seção 3.2.2 indicaram que os desenvolvedores com atuação na *Academia* alcançaram uma concordância maior do que os desenvolvedores em geral em quatro dos sete tipos de *smell* em que foi possível calcular um resultado estatisticamente significante. Entretanto, os níveis das concordâncias obtidas não alcançaram categorias superiores à *Justa*. Considerando os desenvolvedores da *Indústria*, mesmo os resultados apontando que eles obtiveram uma concordância *Moderada* na avaliação de *Long Method*, os desenvolvedores aumentaram a concordância em apenas dois dos cinco tipos de *smell* apresentados.

Considerando a investigação sobre os fatores de experiência descritos na Seção 3.2.3, os resultados foram similares para os fatores relacionados à atuação do desenvolvedor reportados anteriormente. Para os três fatores de experiência analisados, apenas em poucos casos os níveis de concordância foram maiores que os aferidos para todos os desenvolvedores.

De fato, os desenvolvedores mais experientes em *desenvolvimento de software* alcançaram maiores níveis de concordância em cinco tipos de *smell*, e os mais experientes em *detecção de bad smells* alcançaram resultados superiores em apenas quatro tipos. Para os desenvolvedores mais experientes com a *linguagem Java* os resultados foram ainda piores, uma vez que os mesmos alcançaram valores superiores que os desenvolvedores em geral em apenas três dos sete tipos de *smell* analisados. Ao analisar as categorias das concordâncias aferidas, observou-se que apenas os mais experientes em desenvolvimento de software foram capazes de apresentar uma concordância *Moderada* relacionada ao smell *Long Method*. Para todos os demais cenários avaliados, a categoria da concordância não foi superior à *Justa*.

De acordo com discussões acima descritas, nota-se mesmo nos casos em que os desenvolvedores puderam elevar a concordância quando agrupados de acordo com os fatores de atuação e experiência, tal acréscimo foi observado em casos isolados. Além disso, apenas em 2 casos, relacionados à detecção de *Long Method*, foi observada uma concordância *Moderada*. Para os demais 30 dos 32 casos investigados, onde foi possível calcular valores estatisticamente significantes de *Kappa*, a concordância apresentou níveis baixos que não foram superiores à categoria *Justa*. Tais resultados sugerem que os fatores relacionados à atuação e experiência não influenciam os desenvolvedores a concordarem com a detecção de *bad smells*. Afinal, ao agrupar os desenvolvedores de acordo com sua atuação e experiência, eles não puderam, consistentemente, apresentar níveis de concordância maiores que os desenvolvedores em geral.

Em um estudo anterior realizado em [56], Mäntylä analisou se estudantes de mestrado com experiência em desenvolvimento de software poderiam favorecer a concordância em suas avaliações. Os resultados não indicaram nenhuma relação relevante entre a experiência dos estudantes e a concordância nas avaliações deles. Tal conclusão também foi confirmada nos resultados apresentados na Seção 3.2. Entretanto, o estudo apresentado nessa seção realizou uma investigação mais ampla, envolvendo uma grande variedade de fatores e tipos de *smell*.

Em contraste aos fatores de atuação e experiência, os resultados descritos na Seção 3.2.4 indicaram que as heurísticas reportadas pelos desenvolvedores é um importante fator que ajuda a determinar a concordância. De fato, os desenvolvedores que reportaram a mesma heurística de detecção, alcançaram valores de concordância mais elevados do que os resul-

tados respectivos envolvendo todos os desenvolvedores em todos os 12 tipos de *smell*. Além disso, tais resultados alcançaram níveis de concordância que elevaram a categoria em 11 desses tipos analisados, atingindo uma concordância *Substancial* para os *smells Data Class* e *Primitive Obsession*, conforme ilustrado na Figura 3.9. Essa concordância *Substancial* indicou o nível mais elevado nesse estudo. Os resultados também indicaram que os desenvolvedores reportaram uma variedade de diferentes heurísticas para detectar os *smells*, resultando em uma média de, pelo menos, seis heurísticas para cada tipo. Esse alto número de heurísticas reportadas sugere que os desenvolvedores detectam *smells* de diferentes formas, o que ajuda a explicar a discordância observada nas análises anteriores.

Embora a investigação sobre as heurísticas tenha revelado que este fator possa ajudar a reconhecer uma maior concordância entre desenvolvedores, os resultados poderiam ser ainda melhor se outros aspectos fossem considerados. Como exposto na Seção 3.2.4, desenvolvedores que reportaram uma mesma heurística discordaram ao avaliar alguns trechos de código devido às diferentes formas que uma heurística pode ser operacionalizada. Em outras palavras, diferentes estratégias de detecção (envolvendo métricas e limiares) podem ser definidas para implementar uma mesma heurística visando determinar se um trecho de código apresenta um *smell* específico. Esse fato introduz ainda mais dificuldades na criação de uma técnica de customizada automática para identificar *bad smells*. Assim, tais técnicas deveriam permitir a customização da detecção em diferentes níveis, considerando heurísticas, métricas e limiares.

Apesar das ferramentas existentes [65; 73; 17; 39] para a detecção de *bad smells* permitirem a customização de suas técnicas, a partir da adaptação de limiares e da produção de regras de detecção, elas não estão adaptadas para lidar com diferentes heurísticas de forma automática. Além disso, a customização manual dos limiares e regras tende a aumentar o esforço do desenvolvedor [26] prejudicando o processo da detecção [53].

Finalmente, a investigação sobre os agrupamentos dos desenvolvedores também indicou que a heurística reportada pelos desenvolvedores foi o fator mais predominante entre os agrupamentos dos desenvolvedores que tiveram avaliações similares. Tal conclusão ratifica os resultados descritos na Seção 3.2.4, onde os desenvolvedores que apresentaram a heurística mais citada apresentaram uma maior concordância quando comparados com os desenvolvedores em geral. Esse cenário reforça a necessidade de novos estudos que visam

investigar técnicas de detecção capazes de customizar a detecção de *bad smells* automaticamente, baseando-se na heurística considerada pelos desenvolvedores. Talvez, essas técnicas possam mitigar o alto número de casos detectados indevidamente (falsos positivos) observados em resultados relacionados às técnicas de detecção existentes [28], incrementando a produtividade dos desenvolvedores na realização de tarefas que visam melhorar a qualidade do software [29; 26; 36].

3.4 Ameaças à Validade

Nesta seção, são apresentadas as ameaças à validade de acordo com os critérios de validade definidos em [86].

3.4.1 Validade de Construto

No experimento realizado, os desenvolvedores foram apresentados à uma típica definição de cada tipo de *bad smell*, sem apresentar exemplos relacionados ao mesmo. Embora se saiba que a apresentação de exemplos possa, eventualmente, ajudar alguns desenvolvedores a tender melhor cada *bad smell*, tal procedimento foi adotado visando evitar a situação onde a natureza de uma instância de um *smell* específico pudesse influenciar ou induzir as avaliações dos participantes. Em particular, os desenvolvedores poderiam ser encorajados a avaliar os *smells* considerando os mesmos aspectos observados no exemplo. Além disso, suas heurísticas de detecção poderiam ser fortemente baseadas nas características dos exemplos apresentados. Dessa forma, tentou-se deixar os desenvolvedores avaliar os *smells* de acordo com seus próprios entendimentos advindos das suas experiências.

De acordo com Fontana *et al.* [28], as abordagens existentes para a detecção de *bad smells*, como as utilizadas para selecionar os trechos de código utilizados no estudo apresentado [39; 73; 17; 42], inerentemente apresentam resultados não acurados (incluindo falsos positivos). Portanto, o uso de tais abordagens pode detectar suspeitos de *bad smells* de forma equivocada. Com o objetivo de mitigar esse problema, o estudo apresentado utilizou tais abordagens apenas com o o objetivo de selecionar os trechos de código que foram apresentados aos desenvolvedores. Mesmo que alguns destes casos tenham sido apresentados aos desenvolvedores, os mesmos puderam informar se concordavam ou não com a existên-

cia do *smell* analisado. Afinal, a análise realizada no estudo avaliou a concordância entre os desenvolvedores, independente se as abordagens de detecção reportaram se o trecho de código apresentava uma instância do *smell* analisado ou não.

Durante o experimento, os desenvolvedores avaliaram se um trecho de código continha uma instância de um *smell* específico. Nesse caso, o escopo básico de um trecho de código é dependente de cada tipo de *smell*. Por exemplo, o escopo básico de um *smell* candidato a *Long Method* é um método, enquanto o escopo básico de um *smell* candidato a *Data Class* é uma classe. Nesse contexto, pode-se argumentar que o escopo básico de um trecho de código não seja suficiente para apoiar a decisão do desenvolvedor sobre a presença de um *bad smell* no código. De fato, essa premissa foi considerada. Embora estudos anteriores [56; 76; 72] optaram apenas por mostrar o escopo básico de cada *bad smell* para os desenvolvedores, o estudo descrito nesse capítulo apresentou todo o arquivo Java que continha o trecho de código analisado. Além disso, o desenvolvedor poderia navegar no código fonte sempre que ele considerasse necessário para inspecionar outras partes do projeto analisado. Afinal, não é possível saber antecipadamente todas as partes do código que cada desenvolvedor analisou para realizar suas avaliações.

Uma outra ameaça está relacionada às avaliações dos desenvolvedores. O estudo permitiu os participantes avaliassem cada trecho de código reportando as opções **SIM** e **NÃO**. O provimento de apenas essas duas opções pode ser considerado como uma ameaça, uma vez que elas não permitem que os desenvolvedores informassem o grau de confiança em suas respostas. Entretanto, tal procedimento foi adotado para garantir que os desenvolvedores estivessem aptos a decidir sobre a existência de um *bad smell* e para permitir uma comparação mais precisa entre as avaliações dos desenvolvedores. Visando mitigar essa ameaça, foram criadas as questões abertas onde os desenvolvedores puderam descrever seu entendimento sobre *bad smells*, bem como as suas dúvidas sobre eles. Dessa forma, foi possível analisar como os desenvolvedores fizeram suas avaliações. A partir das respostas às questões abertas, foram identificados alguns desenvolvedores que não realizaram as avaliações de forma cuidadosa e/ou coerente. Tais avaliações foram removidas da análise, como detalhado na Seção 3.1.6.

3.4.2 Validade Interna

Cada desenvolvedor avaliou 15 trechos de código relacionados a um único *bad smell*. Esse procedimento foi realizado por um mesmo desenvolvedor até, no máximo, para três tipos de *smells* diferentes. Nesses casos um único participante foi responsável por avaliar 45 trechos de código. Esse número de avaliações pode tornar o processo de avaliação cansativo, fazendo com que o desenvolvedor realize as avaliações sem prestar a devida atenção durante o experimento. Para mitigar essa ameaça, a aplicação que apoiou a execução do estudo permitiu que os desenvolvedores pudessem interromper e continuar suas avaliações durante o experimento. Assim, os participantes puderam realizar um subconjunto de avaliações até o momento em que eles se sentissem cansados. Depois, eles puderam continuar continuar as avaliações no momento em que sentisse preparado para continuar.

A utilização do experimento baseado na internet, permitiu alcançar um grande número de desenvolvedores para avaliar os *bad smells* investigados. Não obstante, uma vez que os desenvolvedores participaram do experimento em locais e momentos diferentes, não foi possível ter um controle estrito para evitar que eles pudessem se comunicar uns com os outros enquanto realizaram as avaliações. Assim, os desenvolvedores poderiam combinar as respostas realizadas nas avaliações. Visando mitigar essa ameaça, a aplicação que favoreceu o experimento modificava a ordem que os trechos de código eram apresentados para os desenvolvedores. Além disso, o alto nível de discordância apresentado na Seção 3.2 sugere que os desenvolvedores não combinaram suas respostas.

Ao responder a questão de pesquisa **Q1** foram obtidos resultados estatisticamente significantes ao analisar todos os 15 tipos de *bad smells* envolvidos no estudo. Por outro lado, ao considerar a investigação dos fatores que ajudaram a responder a questão **Q2**, não foi possível verificar uma concordância estatisticamente significante para todos os *bad smells*. Com o objetivo de evitar conclusões indevidas, a análise desses fatores foi limitada para os casos em que resultados estatisticamente significantes foram obtidos.

Uma outra ameaça está relacionada com as heurísticas reportadas pelos desenvolvedores. Durante o estudo não foi realizada uma análise rigorosa para verificar se os desenvolvedores realmente seguiram as heurísticas relatadas por eles. Mesmo considerando que essa tarefa não está no escopo do trabalho, observou-se que os desenvolvedores realmente aplicaram as heurísticas reportadas por eles ao avaliar *Long Methods*, como detalhado na Seção 3.2.4.

Finalmente, a análise da influência da experiência dos desenvolvedores sobre a concordância levou em consideração a auto-avaliação reportada pelos participantes. Embora as informações fornecidas pelos desenvolvedores possam ser imprecisas quando questionados sobre suas experiências, estudo anterior [79] indica que esse procedimento é coerente para estimar a experiência dos desenvolvedores.

3.4.3 Validade Externa

No estudo realizado foram utilizados trechos de código de cinco projetos Java que possuem diferentes tamanhos e domínios. Tais projetos são bem conhecidos da comunidade e têm sido amplamente utilizados em trabalhos existentes relacionados à detecção de *bad smells* [65; 55; 69; 72]. Não obstante, os resultados apresentados não podem ser inteiramente estendidos a outros projetos uma vez que eles possuem muitas características distintas.

O experimento contou com 75 desenvolvedores de diferentes atuações e experiências, como apresentado na Figura 3.2. Embora o estudo tenha envolvido uma quantidade considerável de desenvolvedores, os resultados também não podem ser estendidos para outros desenvolvedores, uma vez que seus entendimentos sobre *bad smells* dependem de uma variedade de fatores subjetivos.

Os desenvolvedores avaliaram trechos de código visando detectar instâncias de 15 diferentes tipos de *bad smell*. Em geral, os desenvolvedores apresentaram uma alta discordância ao detectar esses *smells*. Entretanto, embora o experimento tenha considerado um conjunto representativo de tipos de *bad smell*, não é possível estender os resultados para outros tipos que não foram analisados no estudo.

3.5 Conclusões do Estudo

No Capítulo 3 foi apresentado um estudo que investigou a concordância dos desenvolvedores ao detectar *bad smells* em código. Além disso, foram investigados possíveis fatores que influenciaram a concordância dos desenvolvedores a partir de suas avaliações. Para isso, foi realizado um estudo envolvendo 75 desenvolvedores que avaliaram 15 tipos de *bad smells* em trechos de código de 5 projetos de código aberto. No total mais de 2.700 avaliações sobre os trechos de código foram realizadas, resultando em uma grande quantidade de dados

quantitativos e qualitativos.

Os dados coletados apresentaram uma grande discordância entre as avaliações dos desenvolvedores sobre todos os tipos de *bad smell* investigados no estudo. Tal discordância foi verificada, inclusive, para *smells* considerados como "simples" em trabalhos anteriores, tal como *Long Method* e *Long Parameter List*. Esses resultados apresentaram evidências que, em geral, os desenvolvedores detectam *bad smells* de formas diferentes, e seus julgamentos pessoais devem ser considerados em técnicas para a detecção automática de *bad smells*. Nesse contexto, a performance de algumas técnicas propostas não deveriam ser avaliadas a partir da comparação de resultados com um oráculo definido por "especialistas", como observado em trabalhos anteriores [45; 65; 55; 70; 72].

Os resultados obtidos com a investigação dos fatores que poderiam influenciar a concordância indicaram que características relacionadas à atuação e experiência dos desenvolvedores não fazem eles concordarem ao avaliar *bad smells*. Não obstante, verificou-se que os desenvolvedores que seguiram uma mesma heurística de detecção alcançaram uma alta concordância no estudo, atingindo níveis e concordância *Moderada* e *Substancial*.

Esses resultados revelaram que a heurística adotada pelos desenvolvedores pode ajudar a determinar como eles detectam *bad smells*. De fato, verificou-se que os desenvolvedores adotaram diferentes heurísticas para detectar um mesmo tipo de anomalia, que resultam na verificação de diferentes características do código avaliado. Não obstante, ao avaliar as heurísticas reportadas juntamente com as avaliações realizadas individualmente, verificou-se que, em alguns casos, os desenvolvedores não conseguiram expressar uma heurística condizente com todas suas avaliações. Esse cenário sugere que, apesar de conseguirem julgar se um determinado trecho de código está manifestado por alguma anomalia, os desenvolvedores apresentam dificuldades ao expressar a heurística utilizada para realizar tal julgamento.

Capítulo 4

Histrategy: uma técnica para a customização guiada de estratégias para a detecção de *bad smells*

Os principais resultados do estudo apresentado no Capítulo 3 reforçaram as evidências de que os desenvolvedores discordam entre si ao avaliarem *bad smells*. Além disso, em uma análise detalhada sobre os fatores que influenciam as avaliações dos desenvolvedores, os resultados descritos na Seção 3.2.4 indicaram que a heurística de detecção, reportada pelos desenvolvedores, pôde ajudar a determinar uma maior concordância entre os mesmos ao avaliaram os *smells* investigados no estudo. Nesse contexto, apesar dos *bad smells* serem apresentados aos desenvolvedores através de uma única definição informal, os desenvolvedores reportaram várias heurísticas de detecção diferentes para julgar se um trecho de código possui uma determinada anomalia.

Apesar das ferramentas de detecção existentes [73; 17; 39; 42; 70; 65] permitirem uma customização da detecção, elas são limitadas à adaptação de limiares ou à definição manual da estratégia de detecção, o que exige um conhecimento extra e esforço [53; 26]. Por outro lado, as abordagens existentes [55; 7; 4] que automatizam parte do processo de customização com algoritmos de aprendizagem de máquina, exigem uma grande quantidade de exemplos para produzir uma estratégia de detecção eficaz. Afinal, o processo de aprendizagem desses mecanismos é realizado de maneira não guiada, sem considerar as heurísticas existentes para a detecção.

Nesse capítulo é apresentada a *Histrategy*, uma técnica para a *customização guiada* de estratégias para a detecção de *bad smells*. *Histrategy* é guiada por um conjunto de heurísticas de detecção previamente definidas, que oferece uma customização a partir da percepção de cada desenvolvedor. Durante o processo de customização, *Histrategy* produz diversas estratégias de detecção e, após convergir com a percepção do desenvolvedor, avalia e seleciona a melhor estratégia produzida. Nesse caso, a melhor estratégia é definida a partir da acurácia obtida por todas as estratégias produzidas. Esse processo permite reduzir o número de exemplos requerido para produzir estratégias de detecção eficazes que sejam sensíveis à percepção de cada desenvolvedor.

A seguir, na Seção 4.1, é apresentada uma visão geral da *Histrategy*, destacando seus componentes e funcionamento. Em seguida, na Seção 4.2 é apresentado um exemplo de execução da *Histrategy* considerando a customização da detecção para *Long Method*.

4.1 *Histrategy*: visão geral

Nessa seção é apresentada Histrategy, uma técnica que implementa a customização guiada de estratégias para a detecção de *bad smells*. Essa técnica tem como objetivo principal oferecer estratégias de detecção eficazes através de um processo de customização eficiente. Com isso, a customização da *Histrategy* irá permitir a detecção de *bad smells* de forma sensível à percepção dos desenvolvedores. A Figura 4.1 oferece uma visão geral sobre o funcionamento da *Histrategy* que visa produzir estratégias de detecção efetivas para os desenvolvedores.

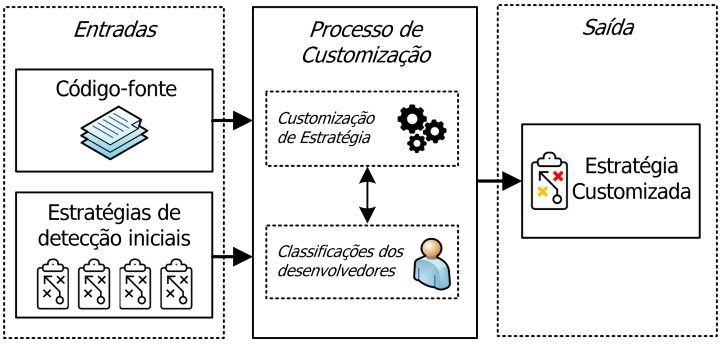


Figura 4.1: Visão geral do funcionamento da *Histrategy*

Inicialmente, *Histrategy* recebe como entrada o código fonte que deve ser analisado para a detecção de instâncias de *bad smell*. Esse código fonte pode ser tanto um sistema inteiro,

como uma subparte, composta por algumas classes e métodos. Além disso, *Histrategy* recebe um conjunto de estratégias de detecção iniciais que serão utilizadas para guiar a customização de acordo com a percepção dos desenvolvedores sobre um determinado tipo de *smell*. Tais estratégias definem diferentes formas de detectar um mesmo tipo de *bad smell*.

Após processar o código fonte e as estratégias recebidas como entrada, *Histrategy* inicia o processo de customização. Nesse ponto, as estratégias de detecção iniciais serão utilizadas para selecionar alguns trechos do código fonte que serão apresentados ao desenvolvedor, que deve classificar se os trechos apresentados contêm uma instância do *smell* ou não. Baseado na classificação do desenvolvedor, *Histrategy* tenta identificar qual estratégia inicial é mais adequada para detectar o *smell* analisado para o desenvolvedor. Além disso, *Histrategy* customiza as estratégias iniciais realizando as adaptações necessárias (alterando limiares e operadores) para que as mesmas classifiquem o smell analisado em conformidade com a classificação do desenvolvedor.

Por fim, após realizar algumas customizações de acordo com as avaliações do desenvolvedor, *Histrategy* apresenta a melhor estratégia de detecção considerada para o desenvolvedor. Nesse sentido, espera-se que a aplicação dessa estratégia possa, automaticamente, reconhecer novas instâncias de *bad smell* de acordo com a percepção do desenvolvedor que participou do processo de customização. Visando explicar o funcionamento da *Histrategy* de forma mais aprofundada, as seções seguintes descrevem detalhes dos componentes principais da customização guiada. Na Seção 4.1.1 são apresentadas algumas formas para a definição de um conjunto de estratégias iniciais que deve ser fornecido ao processo de customização, detalhado na Seção 4.1.2. Finalmente, na Seção 4.2, é apresentado um exemplo de execução da *Histrategy*, detalhando seu funcionamento no processo de customização guiada.

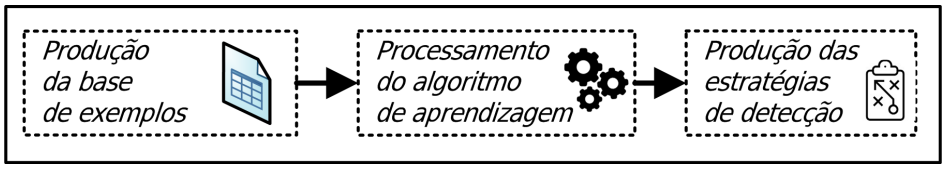
4.1.1 Estratégias de Detecção Iniciais

Diferentemente das técnicas definidas em trabalhos anteriores, a *customização guiada* oferecida pela *Histrategy* considera um conjunto de estratégias de detecção como entrada do processo de customização. Tal procedimento visa ajudar a customização da detecção a partir de um conjunto de diferentes formas de detectar um mesmo tipo de *bad smell*. Assim, a definição dessas estratégias de detecção iniciais tem um importante papel para o funcionamento da *Histrategy*. Nesse contexto, uma vez que uma variedade de estratégias podem ser

derivadas/extraídas de trabalhos anteriores, a construção desse conjunto inicial pode ser uma tarefa pre-processada e independente da customização em si, sem introduzir novos esforços para o desenvolvedor que será auxiliado pela *Histrategy*.

Trabalhos anteriores [60; 51; 66; 47; 55; 7] direcionaram esforços para definir estratégias de detecção capazes de identificar *bad smells*. Tais estratégias foram, principalmente, definidas através de dois procedimentos definidos na literatura. Esses procedimentos, ilustrados na Figura 4.2, são detalhados a seguir.

(a) a partir das heurísticas dos desenvolvedores



(b) a partir de algoritmos de aprendizagem

Figura 4.2: Produção das estratégias de detecção

Marinescu [60] apresentou uma metodologia para definir estratégias de detecção para *bad smells*. Em seu trabalho, essa metodologia foi detalhada através de um exemplo indicando os passos necessários para a definição de uma estratégia capaz de detectar instâncias de *God Class*. Nesses passos, ilustrados na Figura 4.2a, a criação da estratégia de detecção se inicia com a definição de heurísticas de detecção relacionadas com um conjunto de sintomas (como tamanho excessivo de classes, alta complexidade, alto acoplamento) que podem ser capturados através de métricas de software. Em seguida, é realizada uma seleção de métricas adequadas capazes de quantificar melhor cada um dos sintomas identificados no passo anterior. Por exemplo, a complexidade de uma classe pode ser quantificada através da métrica *WMC*, que define a soma da complexidade estatística de todos os métodos de uma classe [19]. Finalmente, no último passo, cada métrica selecionada é combinada com um limiar adequado que permite identificar quando um dado sintoma pode ser observado em um código de um software. Assim, o conjunto das métricas selecionadas juntamente com os limiares definidos compõem uma estratégia de detecção. Seguindo essa metodologia os

trabalhos publicados em [51; 66] propõem estratégias para detectar diferentes tipos de *bad smell*, como *God Class*, *Data Class* e *Temporary Field*.

Além da metodologia proposta por Marinescu [60], alguns trabalhos têm explorado o uso de algoritmos de aprendizagem de máquina para produzir estratégias de detecção [55; 47; 7]. Nesse contexto, as estratégias de detecção são definidas a partir de um processo de aprendizagem a partir de uma amostra de dados contendo instância de *smells* manualmente validados por desenvolvedores, conforme ilustrado na Figura 4.2b. A partir dessa amostra, algoritmos de aprendizagem tentam identificar as métricas de software, e seus respectivos limiares, que são sensíveis às classificações reportadas pelos desenvolvedores. Após esse processo de aprendizagem, o algoritmo apresenta um conjunto de estratégias de detecção que é capaz de classificar os trechos de código de maneira similar à realizada na amostra utilizada no aprendizado. Seguindo esse procedimento, os trabalhos apresentados em [55; 47; 7] propuseram estratégias para detectar *bad smells* como *God Class*, *Data Class* e *Feature Envy*.

4.1.2 Processo de Customização

Após receber o código fonte e o conjunto inicial de estratégias, *Histrategy* inicia seu processo de customização guiada de acordo com o processo ilustrado na Figura 4.3. Esse processo define um fluxo iterativo de acordo com os passos descritos a seguir:

Passo 1 - Seleção da *melhor* estratégia

A partir do conjunto de estratégias disponível, *Histrategy* escolhe uma estratégia que será utilizada para guiar o processo de customização. Tal estratégia, considerada como a *melhor* estratégia da iteração, é escolhida a partir da efetividade das estratégias obtidas inicialmente e/ou criadas em iterações anteriores. Nesse caso a efetividade de cada estratégia é medida a partir da acurácia obtida na classificação de trechos de código já avaliados pelo desenvolvedor. A estratégia que apresentar maior acurácia nessa iteração é escolhida como a *melhor* estratégia. Em casos onde duas ou mais estratégias apresentarem a mesma acurácia, ou quando ainda não existam trechos de código avaliados pelo desenvolvedor, a *melhor* estratégia é escolhida aleatoriamente.

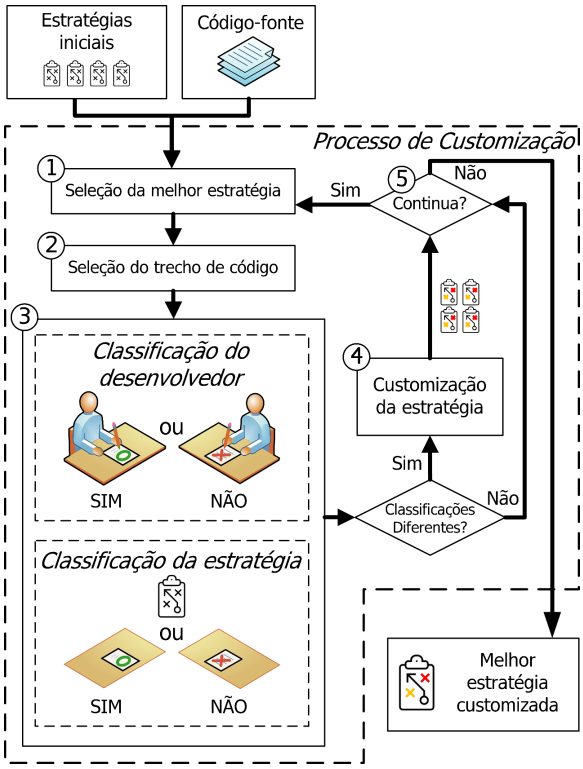


Figura 4.3: Processo de customização da *Histrategy*

Passo 2 - Seleção do trecho de código

Baseada na *melhor* estratégia, *Histrategy* seleciona um trecho do código fonte indicado como entrada. Para tal seleção, *Histrategy* identifica outros trechos de código que possuem valores de métricas próximos dos limiares para as métricas que compõem a *melhor* estratégia escolhida. Por exemplo, se a *melhor* estratégia é definida como $LOC > 80$, então a *Histrategy* pode selecionar um método que contém entre 79–80 linhas de código. No caso de mais de um trecho de código com tais características, *Histrategy* escolhe um deles aleatoriamente.

Passo 3 - Classificação do trecho de código

O trecho de código selecionado é apresentado ao desenvolvedor que deve classificá-lo indicando se o mesmo contém (ou não) o *bad smell* analisado. Após coletar a classificação do desenvolvedor, *Histrategy* também verifica como a *melhor* estratégia classifica o trecho de código selecionado. Em seguida, *Histrategy* analisa se a classificação realizada pelo desenvolvedor é igual à classificação realizada pela *melhor* estratégia, indicando a presença do *bad smell*.

Passo 4 - Customização da estratégia

As classificações realizadas pelo desenvolvedor e pela *melhor* estratégia apresentam um valor binário (**SIM** ou **NÃO**) indicando a presença do *smell* analisado no trecho de código apresentado. Com isso, quatro possíveis cenários podem ocorrer ao se comparar essas classificações, conforme ilustrado na Figura 4.4.

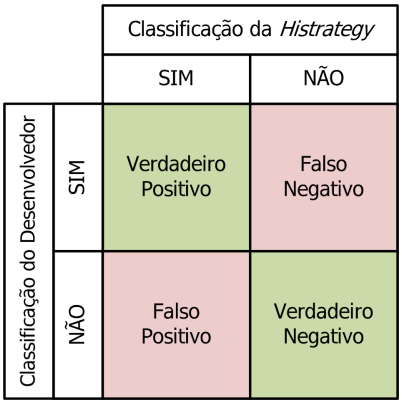


Figura 4.4: Cenários para as classificações da *Histrategy* e do desenvolvedor

Se as classificações forem convergentes, indicando *Verdadeiro-Positivo* ou *Verdadeiro-Negativo*, nenhuma customização é realizada. Nesses casos *Histrategy* considera que a *melhor* estratégia da iteração está em conformidade com a percepção do desenvolvedor. Entretanto, se tais classificações forem divergentes, a estratégia precisa ser customizada de modo que fique de acordo com a percepção do desenvolvedor. Tal customização consiste em um ajuste dos limiares da estratégia de modo que a mesma possa classificar o trecho de código apresentado conforme o desenvolvedor classificou. Nesse sentido, a customização realiza diferentes rotinas a depender do tipo de divergência apresentado.

Para exemplificar como as rotinas de customização são realizadas, considere a *melhor* estratégia como $MLOC \geq 80 \wedge CC \geq 8$, para a detecção de instâncias de *Long Method*, bem como os trechos de código *S1* e *S2* com suas métricas e valores expressos na Tabela 4.1.

Tabela 4.1: Trechos de código e suas métricas

Trecho de código	MLOC	CC
S1	75	9
S2	85	15

Se em uma dada iteração um desenvolvedor classifica que *S1* é uma instância de *Long Method*, a comparação com a avaliação da *melhor* estratégia apresenta um cenário de *Falso-Negativo*. Nesse caso, *Histrategy* identifica quais expressões que compõem a *melhor* estratégia fazem a mesma classificar *S1* como não-*smell*. Nesse exemplo, a expressão $MLOC \geq 80$ foi responsável por determinar a classificação **NÃO** na avaliação de *S1*. Assim, essa expressão é customizada para $LOC \geq 75$ de modo a permitir que a *melhor* estratégia possa classificar *S1* como o desenvolvedor o fez. Ao final, a estratégia customizada será definida como $MLOC \geq 75 \wedge CC \geq 8$.

Um outro cenário de customização ocorre quando a *melhor* estratégia indica que um dado trecho de código contém uma instância do *smell* analisado e o desenvolvedor discorda, resultando em um *Falso-Positivo*. Nesse cenário, a customização se torna pode se tornar mais complexa devido ao desconhecimento sobre o que levou o desenvolvedor a classificar o trecho como não-*smell*. Por exemplo, considere o caso em que o desenvolvedor indicou que o trecho de código *S2* não contém uma instância de *Long Method*, enquanto a *melhor* estratégia ($MLOC \geq 80 \wedge CC \geq 8$) o classificou como *smell*. Considerando que as métricas que compõem a *melhor* estratégia são apropriadas para detectar o *smell*, a partir do exemplo, não é possível saber qual limiar deveria ser modificado para customizar a estratégia de acordo com a percepção do desenvolvedor. Afinal, ele pode ter reportado que *S2* não é *smell* devido ao valor de $MLOC$, CC ou de ambos. Nesse cenário, *Histrategy* considera todas essas possibilidades e cria não apenas uma, mas três novas estratégias a partir dos ajustes de limiares. Portanto, em tal exemplo, as seguintes estratégias são criadas: (i) $MLOC \geq 85 \wedge CC \geq 8$, modificando apenas o limiar de $MLOC$; (ii) $MLOC \geq 80 \wedge CC \geq 15$, modificando apenas o limiar de CC; e, (iii) $MLOC \geq 85 \wedge CC \geq 15$, modificando ambos os limiares de $MLOC$ e CC.

Embora os exemplos apresentados tenham detalhado apenas a customização da *melhor*, outras estratégias armazenadas no processamento da *Histrategy* também devem ser customizadas em uma dada iteração. Isso permite que todas as estratégias possam evoluir juntas, independentemente se elas foram selecionadas como a *melhor* em uma dada iteração. Um exemplo detalhado desse processo é apresentado na Seção 4.2.

Passo 5 - Análise do critério de parada

Sempre que um trecho de código é avaliado pelo desenvolvedor, podendo gerar a customização estratégias, a *Histrategy* precisa decidir se o processo de customização foi suficiente para indicar uma estratégia de detecção alinhada com percepção desenvolvedor, ou se tal processo deve ser executado novamente a partir do **passo 1**. Essa decisão depende que um critério de parada seja satisfeito, considerando dois fatores principais. Primeiramente, deve-se verificar quão convergente são as classificações dos desenvolvedores das estratégias que foram customizadas. Nesse caso, tal convergência deve ser considerada através da efetividade da estratégia ao classificar os trechos de código avaliados pelo desenvolvedor. Um outro fator analisado está relacionado ao esforço requerido dos desenvolvedores para customizar as estratégias. Durante o processo de customização a *Histrategy* frequentemente solicita que o desenvolvedor indique se um trecho de código contém (ou não) uma instância de smell. Assim, se a técnica realiza um grande número de solicitações a customização pode se tornar impraticável ou consumir muito tempo [53]. Portanto, é importante balancear a *efetividade* e o *esforço* ao estabelecer a parada no processo de customização.

4.2 Exemplo de Execução

Para exemplificar o funcionamento da *Histrategy*, a Figura 4.5 ilustra como estratégias para a detecção de *Long Method* podem ser customizadas para um dado desenvolvedor. Do lado esquerdo da Figura 4.5 estão apresentadas as entradas necessárias para a execução da *Histrategy*. Nesse ponto, duas estratégias são definidas: **S1** ($LOC > 100$) e **S2** ($LOC > 90\ and\ CC > 15$). Além disso, cinco métodos, nomeados de *M1-M5* são indicados como entrada. Para cada método, são indicados os valores para as métricas LOC e CC, que serão utilizadas no exemplo. Assim, o processo de customização realiza uma sequência de cinco iterações apresentadas no centro da Figura 4.5 e descritas a seguir:

(**Iter. 1**) *Histrategy* seleciona a *melhor* estratégia entre as estratégias iniciais (**S1** e **S2**). Como o desenvolvedor ainda não proveu nenhuma avaliação no início do processo, a seleção da *melhor* estratégia é realizada aleatoriamente e, nessa iteração, **S1** é selecionada. Em seguida *Histrategy* seleciona um método com valores para as métricas próximos aos limiares da estratégia selecionada. Como **S1** é definida como $LOC > 100$, o método *M4* (com

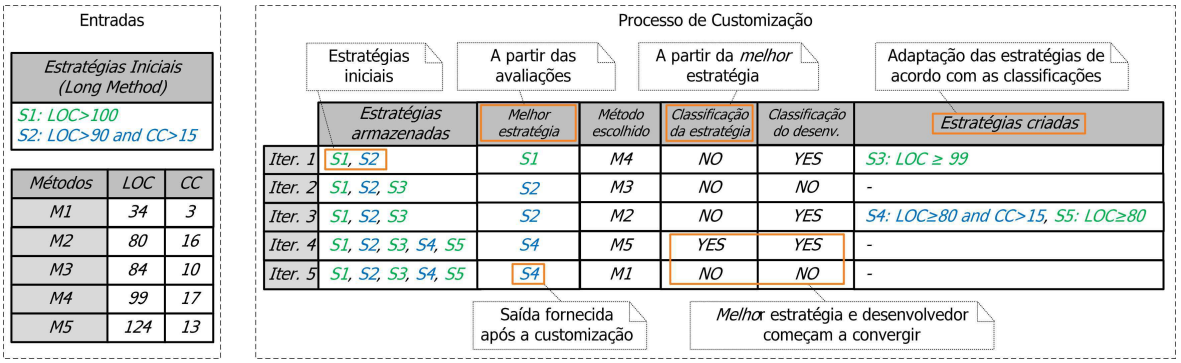


Figura 4.5: Exemplo de customização para *Long Method*

$LOC = 99$) é selecionado e apresentado para o desenvolvedor. O desenvolvedor classifica *M4* como *Long Method*, enquanto **S1** o classifica como não-*smell*. Devido a divergência de classificações, a estratégia **S1** é customizada para seguir a classificação do desenvolvedor, originando a estratégia **S3** ($LOC \geq 99$). Nesse caso, nota-se que além de alterar o limiar, a nova estratégia criada modificou o operador $>$ para $\geq$ de modo a adaptar a estratégia à avaliação do desenvolvedor. Após, *Histrategy* verifica se as estratégias derivadas de outras heurísticas (**S2**) também classificam o método *M4* de forma diferente do desenvolvedor. Uma vez que a classificação do desenvolvedor não diverge da classificação de **S2**, nenhuma estratégia é derivada da mesma.

(**Iter. 2**) *Histrategy* seleciona a *melhor* estratégia a partir das três estratégias disponíveis. Como apenas **S2** e **S3** classificam o método apresentado na iteração anterior conforme o desenvolvedor classificou, estas duas estratégias são analisadas para a escolha da *melhor* estratégia da iteração. Nesse exemplo, **S2** é escolhida aleatoriamente. A partir de **S2**, *Histrategy* recupera o método *M3*, que apresenta valores de métricas próximos dos limiares de **S2**. Nesse ponto, a classificação do desenvolvedor e da *melhor* estratégia convergem indicando que **M3** não apresenta uma instância de *Long Method*. Uma vez que a melhor estratégia derivada de outra heurística (**S3**) também classifica o método como o desenvolvedor o fez, nenhuma nova estratégia é criada nessa iteração.

(**Iter. 3**) Novamente, a *Histrategy* precisa selecionar a *melhor* estratégia da iteração. Nesse ponto, apenas **S2** e **S3** são capazes de classificar os métodos selecionados em iterações anteriores de acordo com o desenvolvedor. Assim, **S2** selecionado aleatoriamente. Em seguida o método *M2* é selecionado para ser apresentado ao desenvolve-

dor, que apresenta uma classificação diferente de **S2**. Nesse ponto é criada a estratégia **S4** ($LOC \geq 80\ and\ CC > 15$) a partir da adaptação dos limiares de **S2**. Além disso, uma vez que a melhor estratégia derivada de outra heurística (**S3)** também diverge da classificação do desenvolvedor outra estratégia deve ser criada. Nesse caso, **S5** é definida a partir da adaptação dos limiares de **S3** para acompanhar a classificação do desenvolvedor.

Nas **Iterações 4 e 5**, **S4** foi capaz de classificar os métodos *M5* e *M1* em acordo com o desenvolvedor. Assim, não foi necessário customizar nenhuma estratégia. Por fim, a estratégia **S4** pode ser apresentada ao desenvolvedor como resultado do processo de customização. Afinal, **S4** é capaz de classificar todos os métodos apresentados no exemplo de forma convergente com as classificações do desenvolvedor.

Capítulo 5

Avaliando a Customização Guiada na Detecção de *Bad Smells*

No Capítulo 4 foi apresentada uma técnica de customização guiada de estratégias para a detecção de *bad smells*. Essa técnica, definida através da *Histrategy*, visa diminuir o esforço necessário para que os desenvolvedores possam obter estratégias de detecção eficazes. Portanto, faz-se necessária uma avaliação para verificar se, de fato, a *Histrategy* é capaz de melhorar a detecção de *bad smells*. Nesse capítulo é apresentado um estudo que visa avaliar a eficácia e eficiência da *Histrategy* na detecção de *bad smells*.

O restante deste capítulo está estruturado como segue. Na Seção 5.1 é apresentado o projeto do estudo, indicando questões de pesquisa, seleção dos indivíduos, procedimento e métodos de análise dos dados. Na Seção 5.2 são apresentados os resultados obtidos com a realização do estudo. As principais conclusões do estudo são discutidas na Seção 5.3. Por fim, na Seção 5.4 são apresentadas as ameaças à validade do estudo.

5.1 Projeto do Estudo

O estudo apresentado nesse capítulo foi projetado para avaliar a *customização guiada* proposta pela *Histrategy* na produção de estratégias capazes de detectar *bad smells* com acurácia para desenvolvedores com percepções diferentes. Nesse sentido, tal estudo se concentra em duas frentes principais:

A primeira frente visa analisar a performance da detecção ao avaliar quão efetivas são as

estratégias customizadas na identificação correta de um *bad smell*. Ainda nesse contexto, é analisado se os procedimentos introduzidos pela customização guiada apresentam um papel importante na detecção proposta pela *Histrategy*.

Por fim, além de avaliar a performance da *Histrategy* o estudo compara sua efetividade e eficiência com abordagens de *customização não guiada*. Tal comparação permite verificar se, de fato, a customização guiada pode melhorar a detecção para os desenvolvedores que possuem diferentes percepções sobre *bad smells*.

Assim, o estudo tenta responder às seguintes questões de pesquisa:

- **Q1**: *Quão efetivas são as estratégias customizadas pela Histrategy na detecção de bad smells sensíveis ao desenvolvedor?*
- **Q2**: *A customização a partir de diferentes heurísticas pode melhorar a detecção de um mesmo tipo de bad smell?*
- **Q3**: *A Histrategy é capaz de detectar bad smells com mais eficácia e eficiência do que as abordagens de customização não guiadas?*

Para responder essas questões foi seguida uma abordagem experimental baseada no uso de algoritmos de aprendizagem de máquina a partir de trechos de código que que incluem instâncias de *bad smell*. Essa abordagem permite produzir e avaliar a performance de estratégias de detecção customizadas a partir de um conjunto de exemplos manualmente validados por um único desenvolvedor. Uma visão geral dessa abordagem é apresentada na Figura 5.1.

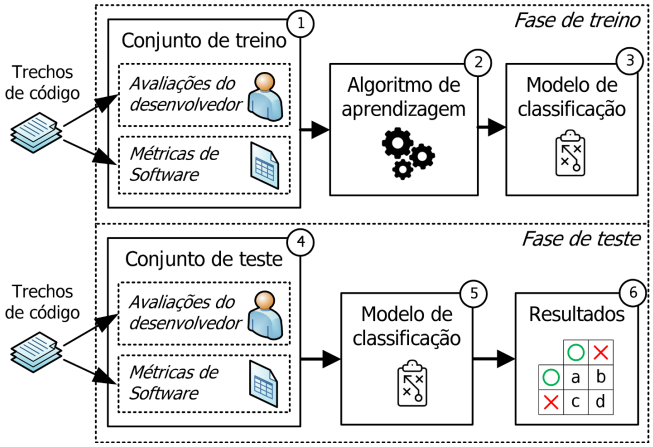


Figura 5.1: Fases de treino e teste dos algoritmos de aprendizagem

A abordagem é baseada em duas fases principais: treino e teste. Na primeira fase, o **conjunto de treino (1)** é composto por uma coleção de avaliações realizadas pelos desenvolvedores, indicando a presença (ou ausência) de um *smell* em trechos de código, juntamente com métricas de software extraídas destes trechos. Em seguida, um **algoritmo de aprendizagem (2)** é treinado para identificar quais métricas podem determinar as avaliações do **smell** analisado. Baseado nessa identificação, o algoritmo produz um **modelo (3)** capaz de classificar qualquer trecho de código como *smell* ou não-*smell*. No estudo apresentado, tal modelo define uma estratégia de detecção composta por métricas e limiares.

Já na fase de teste, o modelo de classificação é testado a fim de verificar se o mesmo é capaz de classificar novas instâncias de *bad smell* como o desenvolvedor faz. Para isto, um **conjunto de teste (4)** é produzido com avaliações e métricas de software relacionadas a outros trechos de código. Em seguida, o **modelo de classificação (5)** tenta a predizer as avaliações do desenvolvedor a partir das métricas de cada trecho de código. Finalmente, as classificações realizadas pelo modelo são comparadas com as do desenvolvedor a fim de obter os **resultados (6)** para avaliar a performance do modelo de classificação.

5.1.1 *Bad Smells* e Trechos de Código

O estudo apresentado contempla a detecção de 8 diferentes tipos de *bad smell*: *God Class*, *Data Class*, *Feature Envy*, *Long Method*, *Long Parameter List*, *Switch Statements*, *Middle Man* e *Message Chains*. Tais *smells* foram escolhidos devido a dois motivos principais. O primeiro está relacionado à reutilização das avaliações obtidas no estudo apresentado no Capítulo 3. A escassez de bases de dados com *smells* anotados por diferentes desenvolvedores motivou a adoção dos dados obtidos no estudo anterior. O segundo motivo considerou o conjunto de estratégias de detecção propostas na literatura para identificar os *smells* avaliados no estudo anterior. Uma vez que a customização guiada implementada pela *Histrategy* utiliza diferentes estratégias de detecção como entrada para detectar um mesmo tipo de *smell*, foram consideradas todas as anomalias em que a literatura apresentou duas ou mais estratégias de detecção. Essa restrição limitou o estudo aos 8 tipos de *smell* considerados.

A partir da escolha dos *smells* selecionados, os trechos de código avaliados foram extraídos de 2 projetos Java que foram utilizados no estudo anterior: GanttProject (2.0.10) e Apache Xerces (2.11.0). Esses projetos foram investigados em outros estudos para a detec-

ção de *smells* [65; 47; 7] que reportaram uma variedade de códigos suspeitos relacionados às anomalias investigadas.

5.1.2 Construção dos Oráculos

A base de dados utilizada para a avaliação apresentada nesse estudo foi criada a partir das avaliações dos desenvolvedores descritas no Capítulo 3. Essa base de dados contou, para cada *smell* analisado, com avaliações independentes de 12 desenvolvedores sobre 15 trechos de código. Essas avaliações individuais sobre um mesmo tipo de *smell* serviram como base para a customização e avaliação das estratégias customizadas nesse estudo.

Uma vez que os desenvolvedores apresentaram percepções diferentes sobre um mesmo *smell* (como descrito na Seção 3.2.1), suas avaliações apresentaram um número diferente de anomalias. Por exemplo, em alguns casos, desenvolvedores indicaram que em 13 das 15 avaliações realizadas um determinado *smell* estava presente. Em outros cenários, desenvolvedores indicaram que apenas 1 *smell* se manifestava nos trechos analisados. Nesse contexto, a utilização de bases desbalanceadas contando, em sua grande maioria, com avaliações com (ou sem) *smell*, poderia prejudicar a customização realizada pelos algoritmos de aprendizagem [7]. Assim, as estratégias customizadas não seriam capazes de oferecer uma detecção acurada.

Visando reduzir os efeitos negativos relacionados à aprendizagem com bases de dados desbalanceadas, as avaliações produzidas pelos desenvolvedores foram verificadas para garantir que os casos em que certo balanceamento estava presente. Nesse sentido, apenas as bases individuais (com 15 avaliações) que contemplavam pelo menos 33% de avaliações com *smell* e 33% de avaliações sem *smell* foram mantidas. Esse procedimento tem sido adotado em outras pesquisas envolvendo a utilização de algoritmos de aprendizagem para a detecção de *bad smells* [87; 7]. Após a remoção das bases desbalanceadas foram consideradas avaliações de 10 desenvolvedores diferentes para cada tipo de smell considerado no estudo. Como cada desenvolvedor avaliou 15 trechos de código, o estudo contou com um total de 1.200 avaliações.

Além de coletar as avaliações dos desenvolvedores, foi extraída uma grande quantidade de métricas de software a partir dos trechos de código analisados. No total, mais de 70 métricas foram extraídas a partir dos trechos de código avaliados pelos desenvolvedores, incluindo

métricas relacionadas ao volume, complexidade e características do paradigma de orientação a objetos. Estas métricas, que foram utilizadas no processo de aprendizagem dos algoritmos avaliados, foram coletadas utilizando diversos extratores e abordagens disponíveis na literatura [73; 17; 77; 51].

Finalmente, visando permitir o processamento realizado pelas técnicas de aprendizagem, foram criados um conjunto de oráculos que combinaram as avaliações dos desenvolvedores com as métricas extraídas dos trechos de código. Cada oráculo contempla as 15 avaliações realizadas por um único desenvolvedor sobre um único *bad smell*, juntamente com as métricas relacionadas aos trechos de código avaliados. No total, foram produzidos 80 oráculos com as avaliações dos desenvolvedores que avaliaram os casos relacionados aos 8 tipos de *smells* analisados no estudo. Nesse sentido, a Figura 5.2 ilustra a estrutura de cada oráculo produzido.

Figura 5.2: Estrutura dos oráculos considerados no estudo

No esquema ilustrado na Figura 5.2 a primeira coluna identifica o trecho de código analisado e a segunda coluna apresenta a respectiva avaliação realizada pelo desenvolvedor. As colunas seguintes descrevem os valores das métricas de software extraídas a partir do trecho de código indicado na primeira coluna. Essas métricas são analisadas por algoritmos de aprendizagem que tentam identificar as características do código que levam o desenvolvedor a avaliar o código como uma instância de *smell*.

5.1.3 Estratégias de Detecção

A customização guiada implementada pela *Histrategy* requer um conjunto de estratégias como entrada visando a detecção de *smells* sensíveis ao desenvolvedor. Nesse sentido, foi considerada uma variedade de estratégias de detecção adotadas em trabalhos anteriores [51; 20; 57; 41; 8; 29; 25; 26; 12; 7; 73; 17; 35] visam detectar os *smells* analisados no estudo.

Tais estratégias são descritas na Tabela 5.1.

Tabela 5.1: Estratégias de detecção propostas em trabalhos anteriores

Bad Smell	Estratégia de Detecção	Referência
Feature Envy	H1: ATFD $\geq$ 9	[7]
	H2: ATFD $>$ 4 $\wedge$ LAA $<$ 0,458 $\wedge$ NOA $\leq$ 16	[7]
	H3: ATFD $\geq$ 5 $\wedge$ LAA $<$ 0,3125	[7]
	H4: FDP $\leq$ 3 $\wedge$ NMO $\leq$ 1	[7]
	H5: ATFD $>$ 5 $\wedge$ LAA $<$ 0,33333 $\wedge$ FDP $\leq$ 5	[51]
Data Class	H1: AMW $\leq$ 1 $\wedge$ NOPVA $\geq$ 3	[7]
	H2: NOAM $>$ 2 $\wedge$ WMCNAMM $\leq$ 21 $\wedge$ NIM $\leq$ 30	[7]
	H3: WOC $<$ 0,35 $\wedge$ NOAM $\geq$ 4 $\wedge$ RFC $\leq$ 41	[7]
	H4: CFNAMM $\leq$ 0 $\wedge$ NOAM $\geq$ 3	[7]
	H5: WOC $<$ 0,33 $\wedge$ WMC $<$ 47 $\wedge$ NOAP+NOAM $>$ 5	[51]
God Class	H1: LOC $>$ 150	[26]
	H2: LCOM $>$ 78 $\wedge$ LOC $>$ 500	[12]
	H3: WMC $\geq$ 47,0 $\wedge$ ATFD $>$ 5 $\wedge$ TCC $\leq$ 0,33	[51]
	H4: WMCNAMM $\geq$ 48	[7]
Long Method	H1: MLOC $\geq$ 80	[73; 17]
	H2: MLOC $\geq$ 80 $\wedge$ CC $\geq$ 8	[7]
Long Parameter List	H1: PARAMS $>$ 8	[57]
	H2: PARAMS $>$ 7 $\wedge$ CC $>$ 5	[29]
Switch Statements	H1: BRANCHES $>$ 3	[25]
	H2: BRANCHES $>$ 3 $\wedge$ CC $>$ 10	[20]
Middle Man	H1: NFM $>$ 2 $\wedge$ FMR $>$ 0,5	[41]
	H2: STATEMENTS $<$ 10	[8]
Message Chains	H1: CHAINS $>$ 3	[12]
	H2: CHAINS $>$ 3 $\wedge$ CC $>$ 10	[35]

As 24 estratégias apresentadas na Tabela 5.1 são capazes de detectar os oito tipos de smells analisados no estudo: *Feature Envy*, *Data Class*, *God Class*, *Long Method*, *Long Parameter List*, *Switch Statements*, *Middle Man* e *Message Chains*. Para cada tipo de *smell* foram selecionadas de duas a cinco diferentes estratégias que representam diferentes heurísticas para detectar um mesmo tipo de *smell*. A principal diferença entre essas estratégias se concentra nas métricas e limiares que as compõem.

A seleção das estratégias consideradas no estudo foi realizada com base em um levan-

tamento das estratégias de detecção apresentadas na literatura e em ferramentas existentes para a identificação de *smells*. Uma vez que a customização guiada implementada pela *Histrategy* utiliza diferentes estratégias de detecção como entrada para detectar um mesmo tipo de *smell*, foram consideradas todas as anomalias em que a literatura apresentou duas ou mais estratégias de detecção. Assim como informado na Seção 5.1.1, essa restrição limitou o estudo aos oito tipos de *smell* considerados.

5.1.4 Experimentação

A partir dos oráculos criados para cada tipo de *bad smell*, foram executados quatro diferentes experimentos visando responder as questões de pesquisa definidas para esse estudo, como segue:

Experimento A. Nesse experimento, *Histrategy* foi utilizada para produzir estratégias de detecção individuais a partir de cada um dos 80 oráculos definido no estudo, juntamente as estratégias que definem as diferentes heurísticas para detectar um dado *smell*. Em seguida, foi avaliada a performance de cada uma das estratégias produzidas, verificando como as mesmas classificam os trechos de código previamente classificados pelos desenvolvedores. Como resultado dessa comparação, a questão **Q1** foi respondida a partir da efetividade aferida para cada estratégia customizada, utilizando a métrica *f-measure* [9]. Essa métrica oferece uma avaliação dos aspectos quantitativos e qualitativos da detecção e vem sido bastante utilizada em diversos trabalhos que lidam com a detecção de *bad smells* [65; 47; 55; 7].

Experimento B. Nesse experimento, o procedimento realizado no *Experimento A* foi repetido alterando as estratégias oferecidas como entrada para a *Histrategy*. Nesse caso, a *Histrategy* foi executada considerando, individualmente, cada uma das estratégias que definem uma heurística particular para detectar um determinado tipo de *bad smell*. Ao final, realizou-se uma comparação da efetividade de cada execução da *Histrategy*, considerando cada uma das heurísticas de detecção. Essa comparação permitiu responder a questão **Q2**.

Experimento C. Para responder parcialmente a questão **Q3**, esse experimento permitiu a comparação da efetividade da *Histrategy* com as técnicas de customização não-guiadas baseadas em seis algoritmos de aprendizagem de máquina, nomeados *J48*, *JRip*, *Random Forest*, *SMO*, *Naive Bayes* e *SVM*, que foram investigados na detecção de *smells* em um

trabalho recente [7]. Nesse caso, a efetividade foi avaliada através da performance de cada modelo de classificação produzido a partir de cada oráculo definido. Assim, a métrica *f-measure* foi utilizada para verificar a efetividade a partir da aplicação de um procedimento de *cross*-validação com cinco partes. Esse procedimento visou limitar problemas de sobre-ajuste (*overfitting*) no experimento.

Experimento D. Finalmente, completando a resposta para a questão **Q3**, verificou-se a eficiência da *Histrategy* em comparação com as mesmas técnicas de customização não-guiadas investigadas no *Experimento C*. Nesse ponto, a eficiência foi definida através da relação da *efetividade* (com a métrica *f-measure*) com o *esforço*, que foi medido a partir do número de exemplos considerados na aprendizagem que permitiu a customização. Mais uma vez, as análises foram realizadas individualmente para cada oráculo definido no estudo. Entretanto, nesse experimento, o aprendizado foi realizado a partir da variação do número de exemplos de 2 a 13, garantindo que ambos os conjuntos de treino e teste, fossem compostos com trechos classificados como *smell* e não-*smell* pelo desenvolvedor. Esse procedimento permitiu verificar as curvas de aprendizado de cada abordagem avaliada.

5.2 Resultados

Nessa seção são apresentados os principais resultados do estudo após a realização dos experimentos descritos na Seção 5.1.4. Tais resultados foram agrupados em seções que definem os principais objetivos de cada experimento. Para favorecer a replicação do experimento e a realização de novas pesquisas relacionadas, os resultados e os materiais utilizados nesse estudo foram disponibilizados [38].

5.2.1 Avaliando a Efetividade da *Histrategy*

Na Tabela 5.2 são apresentados os principais resultados relacionados à efetividade da *Histrategy*. Nela são apresentados valores de *f-measure* obtidos pelas estratégias customizadas a partir de cada oráculo definido no estudo. A primeira coluna da tabela distingue os 10 desenvolvedores que avaliaram trechos de código verificando a presença de cada um dos *smells* identificados nas demais colunas. Finalmente, no rodapé da Tabela 5.2, são reportados os valores mínimo, máximo e médios obtidos na detecção de cada tipo de *bad smell*. A seguir,

os resultados para cada tipo de *smell* são detalhados.

Tabela 5.2: Efetividade da *Histrategy* na detecção de *bad smells* para os desenvolvedores

Desenvolvedor	God Class	Data Class	Feature Envy	Long Method	Long Param. List	Switch Statements	Middle Man	Message Chains
#1	1,000	0,900	0,824	0,870	0,696	0,909	0,667	0,750
#2	0,714	0,941	0,909	1,000	0,800	0,842	0,727	0,714
#3	0,667	0,933	0,800	0,762	0,727	0,923	0,875	0,857
#4	0,923	0,947	0,737	0,875	0,952	0,588	0,706	0,800
#5	1,000	0,818	0,750	0,833	0,857	0,824	0,625	0,750
#6	0,750	0,947	0,842	0,875	0,588	0,750	0,625	0,500
#7	0,800	0,947	0,727	0,889	0,750	0,909	0,700	1,000
#8	0,667	0,909	0,833	0,833	0,833	0,909	0,769	0,696
#9	0,933	0,941	0,933	0,933	0,750	0,706	0,571	0,750
#10	0,714	0,833	0,727	0,800	0,952	0,842	0,706	0,900
MIN	0,667	0,818	0,727	0,762	0,588	0,588	0,571	0,500
MAX	1,000	0,947	0,933	1,000	0,952	0,923	0,875	1,000
MÉD	0,817	0,912	0,808	0,867	0,791	0,820	0,697	0,772

God Class. Observou-se que a *Histrategy* obteve uma alta efetividade na detecção de instâncias de *God Class*, com uma média de 0,817 na detecção para todos os desenvolvedores. Tal resultado contou com uma performance máxima na detecção para os desenvolvedores #1 e #5, quando o *f-measure* foi igual a 1,000. A maioria dos casos restantes apresentou *f-measure* maior que 0,7, com exceção da detecção para os desenvolvedores #3 e #8, onde a *Histrategy* alcançou uma efetividade de 0,667.

Data Class. A detecção de *Data Class* apresentou valores acima de 0,818 para todos os 10 desenvolvedores que participaram do estudo. Embora *Histrategy* não tenha alcançado a máxima efetividade para nenhum desenvolvedor, a média obtida para eles alcançar o valor mais alto do experimento considerando todos os tipos de *smell*, com um valor de 0,912. Nesse cenário a mais alta performance foi obtida para os desenvolvedores #4, #6 e #7, que alcançaram uma efetividade de 0,947.

Feature Envy. A efetividade obtida na detecção de *Feature Envy* apresentou valores maiores que 0,700 para todos os 10 desenvolvedores. O menor valor (0,727) foi obtido na detecção para os desenvolvedores #7 e #10. Por outro lado, *Histrategy* alcançou uma efetivi-

dade maior que 0,900 para dois desenvolvedores. O desenvolvedor #2 obteve uma detecção com *f-measure* igual a 0,909, enquanto o desenvolvedor #9 alcançou uma efetividade de 0,933, que foi o valor mais alto de *f-measure* relacionado à detecção de *Feature Envy*.

Long Method. Para esse *smell*, *Histrategy*, atingiu uma alta efetividade para todos os 10 desenvolvedores que analisaram trechos de código na identificação de instâncias de *Long Method*. Exceto para o desenvolvedor #3, que obteve *f-measure* de 0,762, os demais desenvolvedores alcançaram uma efetividade maior ou igual a 0,800. Os valores mais altos foram obtidos pelo desenvolvedor #9, com *f-measure* igual a 0,933, e pelo desenvolvedor #2, que a *Histrategy* alcançou uma efetividade perfeita (1,000). Tais resultados levaram a detecção de *Long Method* a atingir uma média de efetividade igual a 0,867.

Long Parameter List. A detecção de *Long Parameter List* apresentou resultados que variaram de 0,588 a 0,952. Exceto na detecção para o desenvolvedor #6, todos os demais desenvolvedores atingiram valores de *f-measure* superiores a 0,600. Ao final, os desenvolvedores apresentaram, em média, uma efetividade de 0,791.

Switch Statements. Os resultados relacionados à detecção de *Switch Statements* foram similares aos obtidos para *Long Parameter List*. De fato, a menor efetividade aferida para ambos os *smells* foi igual a 0,588, alcançada para o desenvolvedor #6 na detecção de *Long Parameter List*. O valor mais elevado para esse *smell* foi obtido para o desenvolvedor o #3, que apresentou *f-measure* de 0,923. Tais resultados levaram a detecção de *Switch Statements* a alcançar, em média, uma efetividade de 0,820.

Middle Man. A detecção do *smell Middle Man* apresentou os valores mais baixos do experimento. De fato, o desenvolvedor #9 alcançou uma efetividade de apenas 0,571. Esse foi o menor valor alcançado no experimento, considerando todos os desenvolvedores que avaliaram este tipo de *bad smell*. Além disso, a maior efetividade verificada para esse *smell* foi obtida pelo desenvolvedor #3, que obteve um valor de *f-measure* igual a *0,875*. Esse valor máximo foi o único, considerando todos os outros *smells*, que não foi superior a 0,900. Por fim, a efetividade na detecção de *Middle Man* alcançou a menor média entre todos os *smells*, com um valor de 0,697.

Message Chains. Finalmente, os resultados relacionados a *Message Chains* apresentaram a maior diferença entre os valores mínimo e máximo, considerando todos os *smells* analisados. Enquanto o desenvolvedor #6 alcançou um valor de *f-measure* igual a 0,500, o

desenvolvedor #7 conseguiu uma efetividade perfeita (1,000). A detecção dos demais desenvolvedores apresentou valores de *f-measure* que variaram de 0,696 a 0,900. Embora a efetividade para o desenvolvedor #6 tenha apresentado o menor valor do experimento considerando todos os *smells*, a média obtida para *Message Chains* não foi a menor. Nesse caso a média foi igual a 0,772, que ainda foi superior à média obtida na detecção do *smell Middle Man*.

5.2.2 Avaliando o Uso de Diversas Heurísticas para a Detecção

Após executar o **Experimento B** foram coletadas as médias de *f-measure* considerando, individualmente, cada estratégia de detecção que operacionalizava uma determina heurística na detecção de *bad smell*. Os resultados obtidos estão ilustrados na Figura 5.3. Os resultados estão agrupados para cada tipo de *bad smell* apresentado no *eixo x*. Cada tipo de *smell* está associado com um conjunto de barras horizontais coloridas indicando a efetividade (em termos de *f-measure*) ao executar a *Histrategy* considerando como entrada cada heurística apresentada na Tabela 5.1. Além disso, visando facilitar a análise dos resultados, foi inserida, para cada *smell*, uma linha vertical (em azul) indicando a efetividade obtida pela *Histrategy* ao considerar todas as heurísticas como entrada do processo de customização. Os valores representados por essas linhas se referem às médias apresentadas na Tabela 5.2. A legenda no topo da Figura 5.3 permite identificar a efetividade associada à cada heurística/estratégia. A seguir são descritos os principais resultados do experimento.

Os resultados relacionados à detecção de *Data Class* apresentaram a efetividade obtida para cinco diferentes heurísticas. A customização a partir da heurística **H1** alcançou, em média, uma efetividade de 0,851, conforme ilustrado pela barra laranja associada a esse tipo de *smell*. Em seguida, a heurística **H2** permitiu a produção de estratégias que foram capazes de alcançar uma efetividade de apenas 0,787. A customização considerando apenas a heurística **H3** apresentou o valor mais alto de efetividade entre as cinco heurísticas avaliadas para esse *smell*, com *f-measure* igual a 0,893. Após, a heurística **H4**, alcançou um valor próximo, com *f-measure* igual a 0,890. Finalmente, a customização a partir da heurística **H5**, apresentou a mais baixa efetividade na detecção de instâncias de *Data Class*, com um valor de apenas 0,714. Esses resultados revelaram que as estratégias customizadas a partir de uma heurística individual não foram capazes de atingir uma efetividade superior àquelas

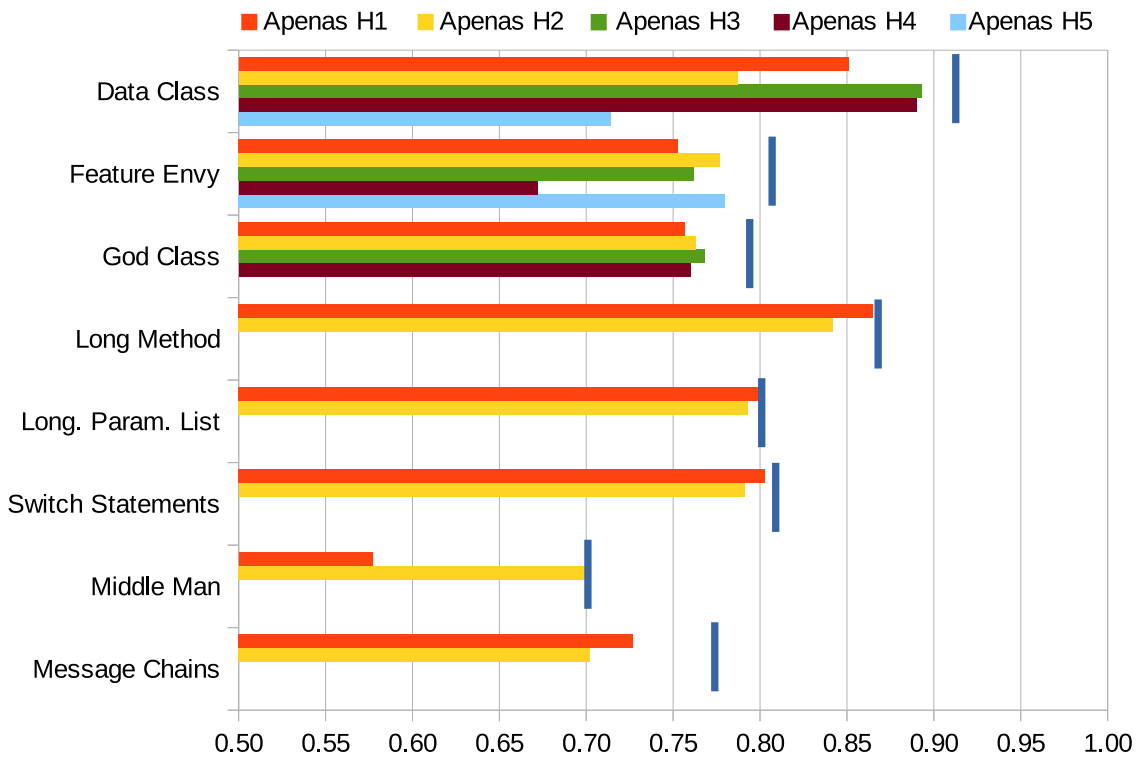


Figura 5.3: Efetividade obtida considerando heurísticas individuais

obtidas pela *Histrategy* ao considerar todas as heurísticas como entrada. Afinal, a acurácia obtida pela *Histrategy* ao considerar todas as cinco heurísticas foi igual a 0,913.

A detecção de *Feature Envy* também contou com cinco diferentes heurísticas utilizadas no experimento. Nesse caso, a customização a partir de uma heurística individual não foi capaz de alcançar uma efetividade maior que 0,8. Os valores mais elevados foram obtidos pelas heurísticas **H5** e **H2**, que alcançaram *f-measure* igual a 0,780 e 0,777, respectivamente. Em seguida, a heurística **H3** alcançou uma efetividade de 0,762, seguida pela heurística **H1**, com *f-measure* igual a 0,753. O valor mais baixo foi aferido na customização da heurística **H4**, que apresentou uma efetividade de 0,672. Diferentemente da customização a partir de heurísticas individuais, *Histrategy* foi capaz de detectar instâncias de *Feature Envy* com uma efetividade de 0,807, quando considerou todas as heurísticas em seu processo de customização.

Os resultados relacionados à customização para a detecção de *God Class* considerou quatro heurísticas diferentes. Nesse caso, a customização para cada uma dessas heurísticas apresentou uma efetividade similar, com valores que variaram de 0,757 (com a heurística

H1) a 0,768 (com a heurística **H3**). Para esse *smell*, *Histrategy* alcançou uma efetividade de 0,796, quando considerou as quatro heurísticas para detectar *God Class*.

Para cada um dos demais *smells* analisados apenas duas heurísticas foram avaliadas individualmente. Para *Long Method*, *Long Parameter List* e *Switch Statements*, suas heurísticas **H1** apresentaram efetividade que alcançaram os valores 0,865, 0,801 e 0,803, respectivamente. Com exceção do *smell Long Parameter List*, esses valores foram ligeiramente inferiores que os obtidos pela *Histrategy* ao considerar todas as heurísticas, que alcançou uma efetividade de 0,867 para *Long Method* e 0,810 para *Switch Statements*. Já na detecção de *Long Parameter List*, *Histrategy* apresentou a mesma efetividade obtida pela customização da heurística **H1**, com *f-measure* igual a 0,801.

Similar à detecção de *Long Parameter List*, a customização a partir de uma heurística individual para detectar instâncias de *Middle Man* também apresentou uma maior efetividade igual à obtida pela *Histrategy*. Nesse caso, o valor foi obtido através da customização da heurística **H2**, que alcançou um valor igual a 0,701. Para esse *smell* a customização a partir da heurística **H1** apresentou a pior efetividade verificada em todo o experimento, com *f-measure* igual a 0,577.

Finalmente, a detecção de *Message Chains* obteve uma efetividade de 0,727 a partir da heurística **H1** e 0,702, a partir de **H2**. Tais valores não foram superiores ao obtido pela *Histrategy* (0,774) quando considerou ambas as heurísticas na customização.

5.2.3 Avaliando a Efetividade das Técnicas de Customização Guiada e não-Guiada

Após executar o **Experimento C** foram coletados os resultados relacionados à efetividade das técnicas de customização guiada e não-guiada. Tais resultados indicam a efetividade (em termos de *f-measure*) que cada técnica alcançou na detecção de *smells* para os desenvolvedores que participaram do estudo. A Figura 5.4 reúne um conjunto de figuras onde os resultados para cada tipo de *smell* são ilustrados. Em cada figura, o *eixo y* indica a média dos valores alcançados por cada técnica na detecção do tipo de *smell* analisado. Para cada barra apresentada é indicado o valor exato alcançado por cada técnica identificada pela legenda no topo de cada figura.

(a) God Class (b) Data Class (c) Feature Envy (d) Long Method

(e) Long Parameter List (f) Switch Statements (g) Middle Man (h) Message Chains

Figura 5.4: Efetividade das abordagens de detecção

Os resultados apresentados na Figura 5.4a indicam que a *Histrategy* alcançou uma efetividade média de 0,627 na detecção de *God Class*. Esta média foi a maior que as atingidas pelas demais técnicas avaliadas. Entre as técnicas *não-guiadas*, *Naive Bayes* alcançou a maior média, com valor igual a 0,546. *SMO* alcançou uma média igual a 0,511, que foi ligeiramente inferior à alcançada pelo algoritmo *Naive Bayes*. Por outro lado, o algoritmo *SVM* apresentou a pior média, com valor igual a 0,158, entre todos os casos relacionados à detecção de *God Class*.

Nos casos relacionados à detecção de *Data Class*, *Histrategy* alcançou uma média igual a 0,795, conforme ilustrado na Figura 5.4b. Esta média representa o maior valor obtido por todas as técnicas na detecção de todos os tipos de *smell* analisados no experimento. Considerando as técnicas *não-guiadas*, *Random Forest* alcançou uma média superior às demais técnicas. Nota-se que a média obtida pelo algoritmo *SVM* foi apenas ligeiramente inferior ao obtido pelo *Random Forest*. Enquanto *Random Forest* alcançou uma média igual a 0,664, a média obtida pelo *SVM* foi igual a 0,641. Ainda considerando a detecção de *Data Class*, *Naive Bayes* apresentou a pior média, com valor igual a 0,166.

A Figura 5.4c ilustra os resultados relacionados à detecção de *Feature Envy*. Para esse tipo de *smell*, *Histrategy* alcançou a maior efetividade com média igual a 0,629. Em seguida, o algoritmo *SMO* apresentou a segunda maior efetividade, com média igual a 0,615. Os piores resultados foram obtidos pelos algoritmos *Random Forest*, com média igual a 0,490, e *JRip*, com média de 0,484.

Considerando a detecção de *Long Method*, os resultados apresentados na Figura 5.4b indicam que a *Histrategy* obteve o valor médio mais elevado (0,761) quando comparado com os valores obtidos pelas demais técnicas. Considerando as técnicas *não-guiadas*, o valor mais alto foi obtido pelo algoritmo *SVM*, com efetividade média de 0,730, seguido por *Naive Bayes* com efetividade de 0,702. O pior resultado foi obtido pelo algoritmo *Random Forest*, com efetividade média de apenas 0,511.

Os resultados relacionados à *Long Parameter List*, ilustrados na Figura 5.4e, também apresentaram uma superioridade da *Histrategy* quando comparada as técnicas *não-guiadas*. Para esse tipo de *smell*, a abordagem *guiada* foi capaz de alcançar uma efetividade média de 0,750, enquanto o resultado mais elevado das técnicas *não-guiadas* foi obtido pelo algoritmo *SVM*, com efetividade média de 0,655. O algoritmo *Random Forest* apresentou a menor média, com valor igual a 0,352.

Os resultados ilustrados na Figura 5.4f, indicam que a *Histrategy* foi capaz de alcançar uma efetividade média de 0,730. Tal resultado foi maior que os obtidos pelas técnicas *não-guiadas*. Com exceção da média aferida para *SMO* (0,572), as técnicas *não-guiadas* apresentaram resultados similares, que variaram de 0,616, com *J48*, a 0,663, com *SVM*.

Para a detecção de *Middle Man*, *Histrategy* apresentou a maior diferença entre a efetividade média aferida e os valores obtidos pelas técnicas *não-guiadas*. Enquanto a média obtida pela *Histrategy* foi igual a 0,607, o valor mais elevado obtido pelas técnicas *não-guiadas* foi obtido pelo algoritmo *J48*, com média igual a 0,404. Nesse caso, o pior resultado foi obtido pelo algoritmo *SVM*, que alcançou uma efetividade média de apenas 0,098. Esse valor foi a pior efetividade média aferida para todas as técnicas avaliadas no experimento, considerando todos os tipos de *smell* analisados.

Finalmente, os resultados relacionados à detecção de *Message Chains* indicaram que a *Histrategy* foi capaz de alcançar a maior efetividade média entre todas as técnicas avaliadas, com valor igual a 0,700. Considerando as técnicas *não-guiadas* o valor mais elevado foi

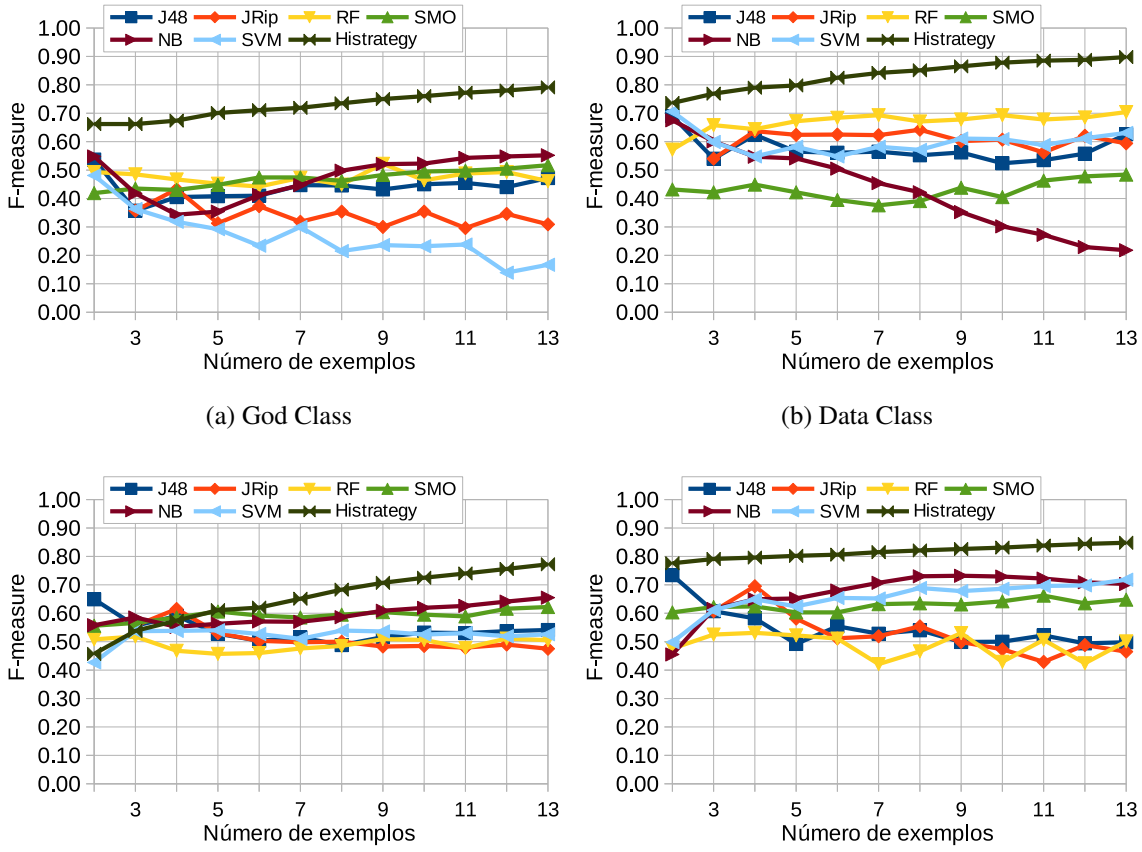
obtido pelo algoritmo *JRip*, que apresentou uma efetividade média de 0,660. Para esse tipo de *smell*, o pior resultado foi atribuído ao algoritmo *Naive Bayes*, que obteve uma média de 0,538.

5.2.4 Avaliando a eficiência das técnicas de customização guiada e não-guiada

A execução do **Experimento D** permitiu avaliar a eficiência das técnicas de customização. Nesse estudo, a eficiência foi aferida em termos de: (i) efetividade, a partir da métrica *f-measure*, e (ii) esforço de treinamento, considerando o número de exemplos necessários para alcançar uma determinada efetividade na detecção. A Figura 5.5, concentra os resultados obtidos para cada tipo de *smell* analisado. Os gráficos que compõem tais figuras descrevem as *curvas de aprendizado* que indicam a eficiência alcançada por cada técnica avaliada. Em particular, cada curva de aprendizado descreve a efetividade média (*eixo y*) obtida por cada técnica a medida que o número de exemplos considerado varia conforme indicado no *eixo x*. A legenda no topo de cada gráfico indica as cores usadas para identificar a curva de aprendizado de cada técnica avaliada.

Considerando a detecção de *God Class*, *Histrategy* foi capaz de alcançar uma eficiência superior às apresentadas pelas técnicas *não-guiadas*, considerando a aprendizagem de 2 a 13 exemplos, como ilustrado na Figura 5.5a. De fato, enquanto *Histrategy* foi capaz de apresentar uma efetividade de 0,66 ao utilizar apenas dois exemplos, as outras técnicas não foram capazes de obter uma efetividade maior que 0,55. Observa-se também que, diferentemente das técnicas *não-guiadas*, *Histrategy* obteve um incremento na efetividade a medida que o número maior de exemplos foi considerado, atingindo uma efetividade maior que 0,79 ao considerar 13 exemplos. As demais técnicas não apresentaram uma efetividade superior a 0,57 em nenhum ponto de suas curvas de aprendizado.

Similarmente à detecção de *God Class*, *Histrategy* apresentou uma eficiência superior às técnicas *não-guiadas* em todas as análises relacionadas à *Data Class*, conforme apresentado na Figura 5.5b. Nesse cenário, observou-se que a *Histrategy* alcançou uma efetividade média igual a 0,74 ao considerar apenas dois exemplos. Essa efetividade foi a mais elevada considerando apenas dois exemplos. Além disso, a efetividade da *Histrategy* apresentou um



(a) God Class

(b) Data Class

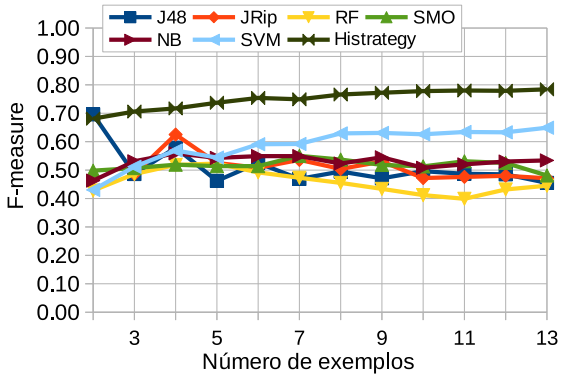
(c) Feature Envy

(d) Long Method

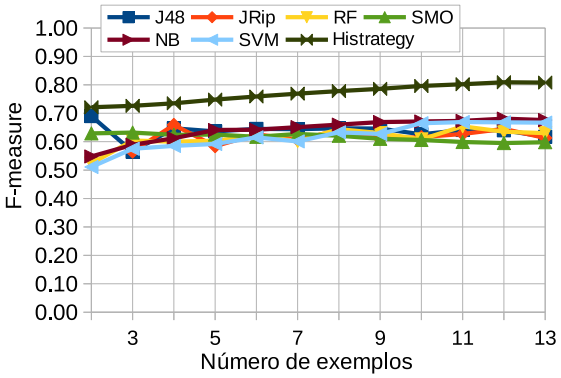
Figura 5.5: Curvas de aprendizado das abordagens de detecção

incremento a medida que o número de exemplos foi aumentando, alcançando uma efetividade igual a 0,9 ao considerar 13 exemplos no aprendizado. O pior resultado foi obtido pelo algoritmo *Naive Bayes* que diminuiu a efetividade registrada a medida que mais exemplos foram considerados no treinamento, levando a técnica a apresentar uma efetividade de apenas 0,22, quando considerou 13 exemplos.

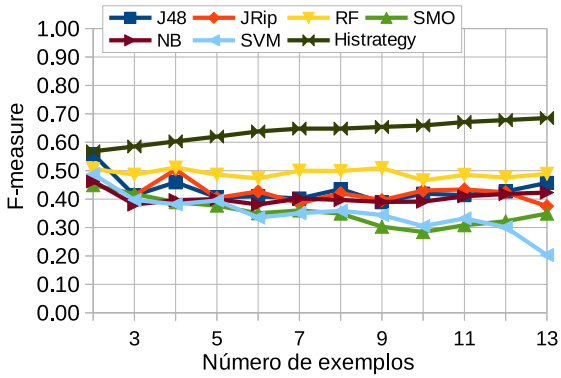
Diferentemente dos outros tipos de *smell*, os resultados relacionado à detecção de *Feature Envy* indicaram que a *Histrategy* não apresentou a maior efetividade ao considerar poucos exemplos em sua curva de aprendizado. Nesse caso, os maiores valores foram atribuídos ao algoritmo *J48* (com dois exemplos de treino), *Naive Bayes* (com três exemplos) e *JRip* (com quatro exemplos). Não obstante, ao considerar cinco exemplos, *Histrategy* apresentou uma ligeira superioridade às demais técnicas e aumentou essa diferença a medida que mais exemplos foram considerados no aprendizado. Ao final, *Histrategy* alcançou uma efetividade de 0,772, enquanto as técnicas *não-guiadas* apresentaram a maior efetividade com o algoritmo



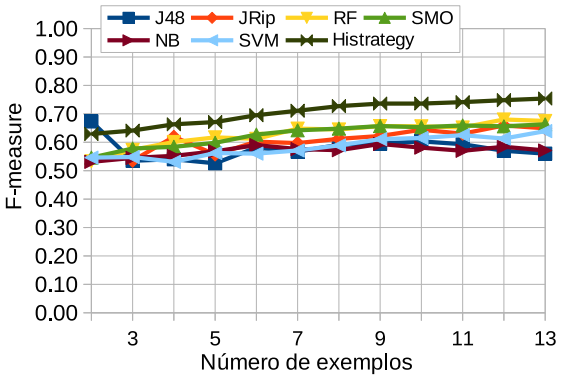
(e) Long Parameter List



(f) Switch Statements



(g) Middle Man



(h) Message Chains

Figura 5.5: Curvas de aprendizado das abordagens de detecção (cont.)

Naive Bayes, que alcançou uma média de *f-measures* igual a 0,655.

No caso da detecção de *Long Method*, *Histrategy* apresentou uma eficiência maior que as demais técnicas em todos os cenários analisados, como ilustrado na Figura 5.5d. Para esse *smell*, *Histrategy* foi capaz de alcançar uma efetividade de 0,85 ao considerar 13 exemplos. As técnicas *não-guiadas* apenas apresentaram uma efetividade inferior a 0,72, com *SVM*, e 0,70 com *Naive Bayes*.

As curvas de aprendizado relacionadas à detecção de *Long Parameter List* e *Switch Statements* apresentaram resultados similares, como ilustrado nas Figuras 5.5e e 5.5f, respectivamente. Para estes *smells*, *Histrategy* apresentou uma efetividade superior às demais técnicas ao variar o número de exemplos considerado. A única exceção ocorreu na detecção de *Long Parameter List* com dois exemplos, quando *Histrategy* e *J48* obtiveram a mesma efetividade igual a 0,70. No final, com 13 exemplos, *Histrategy* alcançou uma efetividade próxima de 0,8 ao detectar esses tipos de *smell*. Nesse ponto, a maior efetividade das demais técnicas foi

alcançada pelo algoritmo *SVM* (0,65) ao detectar instâncias de *Long Parameter List*, e *Naive Bayes* (0,78) ao detectar casos de *Switch Statements*.

Similar aos resultados anteriores, a detecção de *Middle Man* também apresentou uma superioridade da *Histrategy*, conforme ilustrado na Figura 5.5g. Para esse *smell*, a técnica apresentou uma efetividade que variou de 0,57 a 0,68 enquanto o número de exemplos variou de 2 a 13. Ao final a curva de aprendizado do algoritmo *Random Forest* apresentou a maior efetividade entre as técnicas *não-guiadas* com valor médio igual a 0,49.

Finalmente, a detecção de *Message Chains* ilustrada na Figura 5.5h apresentou uma ligeira superioridade da *Histrategy* quando considerou uma vasta maioria dos exemplos avaliados. A única exceção ocorreu quando as técnicas consideraram apenas dois exemplos, onde o algoritmo *J48* apresentou uma efetividade igual a 0,67, enquanto a *Histrategy* alcançou uma efetividade de 0,63. Não obstante, ao considerar 3 ou mais exemplos, *Histrategy* apresentou uma efetividade superior a todas as técnicas avaliadas, obtendo uma efetividade de 0,75 quando considerou 13 exemplos no aprendizado. Nesse ponto, a maior efetividade das demais técnicas foi alcançada pelo algoritmo *Random Forest*, que apresentou uma média de 0,67.

5.3 Discussões

Nessa seção, os resultados apresentados anteriormente são discutidos visando responder às questões de pesquisa do estudo. Além disso, são reportados algumas conclusões relacionadas a trabalhos existentes.

5.3.1 Q1: Quão efetivas são as estratégias customizadas pela *Histrategy* na detecção de *bad smells* sensíveis ao desenvolvedor?

Os resultados apresentados indicam que as estratégias produzidas pela *Histrategy* apresentam uma efetividade média igual a 0,810 ao detectar todos os *smells* analisados no estudo. Nesse cenário, as maiores médias foram verificadas na detecção de *God Class* (0,817), *Switch Statements* (0,820), *Long Method* (0,867) and *Data Class* (0,912). Além disso, os resultados contaram ainda com uma perfeita efetividade (1,000) obtida para desenvolvedores que

detectaram instâncias de *God Class*, *Long Method* and *Message Chains*. A menor efetividade média foi observada na detecção de *Middle Man*, onde a *Histrategy* ainda apresentou uma efetividade próxima a 0,7. Tais resultados sugerem que as estratégias produzidas pela *Histrategy* foram capazes de detectar *bad smells* com alta efetividade para a maioria dos desenvolvedores.

Um outro ponto interessante observado nos resultados se dá com o fato de que diferentemente de estudos anteriores [55; 7], onde centenas de exemplos tiveram de ser manualmente validados pelos desenvolvedores, o estudo apresentado obteve uma alta efetividade com a *Histrategy* ao considerar poucos exemplos. Portanto, os resultados sugerem que o uso da customização *guiada* permitiu reduzir o esforço envolvido para realizar a customização das estratégias de detecção.

5.3.2 Q2: A customização a partir de diferentes heurísticas pode melhorar a detecção de um mesmo tipo de *bad smell*?

Os resultados apresentados na Figura 5.3 indicaram que a customização das estratégias derivadas a partir de uma única heurística não foi melhor do que a customização a partir de um conjunto com diferentes heurísticas. De fato, para 22 das 24 heurísticas consideradas no estudo, a customização a partir de uma única heurística apresentou uma efetividade inferior quando comparada à customização considerando todas as heurísticas para detectar um mesmo tipo de *bad smell*. Por exemplo, ao comparar os resultados da detecção de *Data Class* considerando todas e somente a heurística **H5**, verifica-se uma diferença de quase 20% na efetividade aferida. Tal resultado sugere que a customização para diferentes desenvolvedores deve considerar diferentes heurísticas, permitindo uma detecção adequada à percepção de cada desenvolvedor.

Embora os resultados relacionados a essa questão tenham sido obtidos a partir de diferentes execuções da *Histrategy*, suas conclusões também pode ter impacto sobre ferramentas de detecção existentes que adotam uma única heurística para detectar um dado *smell*. Afinal, quando apenas uma heurística de detecção é considerada como entrada da *Histrategy*, o processo de customização é restringido a ajustes de limiares da estratégia oferecida como entrada. Esse comportamento é similar ao oferecido por ferramentas de detecção, tais como

PMD [73], *Checkstyle* [17] e *inFusion* [39], que resumem a customização através da indicação manual dos limiares adotados pela heurística considerada internamente para a detecção. Portanto, o uso dessas ferramentas podem não apresentar um resultado efetivo quando utilizado por desenvolvedores com percepções diferentes sobre um determinado tipo de *smell*.

5.3.3 Q3: A *Histrategy* é capaz de detectar *bad smells* com mais eficácia e eficiência do que as abordagens de customização não guiadas?

De acordo com os resultados ilustrados na Figura 5.4, foi observado que a *Histrategy* atingiu valores de efetividade média maiores que as técnicas *não-guiadas* em todos os casos analisados. Percebeu-se também que a consistência dos resultados obtidos pela *Histrategy* não foi observada com os resultados obtidos pelas técnicas de customização *não-guiadas*. Por exemplo, enquanto o algoritmo *SVM* alcançou a maior efetividade média entre as técnicas *não-guiadas* ao detectar *Long method*, *Long Parameter List* e *Switch Statements*, o mesmo algoritmo apresentou o pior resultado entre todas as técnicas, na detecção de instâncias de *God Class* e *Middle Man*. Similarmente, o algoritmo *Naive Bayes* apresentou a melhor efetividade média entre as técnicas *não-guiadas* ao detectar instâncias de *God Class*, mas também apresentou a pior performance ao detectar casos de *Data Class*. Portanto, diferentemente das técnicas *não-guiadas*, foi observada uma influência consistente da customização guiada sobre a melhora da efetividade na detecção de *smells* sensíveis ao desenvolvedor.

Ao analisar a eficiência das técnicas de customização *guiada* e *não-guiada*, ilustrados na Figura 5.5, também foi observada uma superioridade da *Histrategy* contra as demais técnicas. Afinal, os resultados indicaram que *Histrategy* alcançou uma eficiência superior às outras técnicas na vasta maioria dos casos analisados para os oito tipos de *smell* investigados. Observou-se ainda que, enquanto a customização guiada aumentava a efetividade a medida que o número de exemplos considerados no aprendizado crescia, as curvas de aprendizado das técnicas *não-guiadas* não apresentaram o mesmo comportamento. De fato as curvas referentes aos algoritmos *SVM* ao detectar *God Class* e *Naive Bayes* ao detectar *Data Class* diminuíram consistentemente a efetividade aferida a medida que mais exemplos foram considerados no aprendizado. Tais resultados sugerem que a *Histrategy* é capaz detectar *smells* sensíveis ao desenvolvedor mais efetivamente e com um número menor de exemplos

do que as técnicas *não-guiadas*. Em média, a *Histrategy* foi capaz de apresentar uma acurácia superior a 70% após considerar menos de 6 exemplos durante o aprendizado. Essas conclusões sugerem que a customização guiada possa ser uma importante alternativa para acomodar as diferentes percepções dos desenvolvedores, produzindo estratégias customizadas com eficiência.

5.4 Ameaças à Validade

Nesta seção, são apresentadas as ameaças à validade de acordo com os critérios de validade definidos em [86].

5.4.1 Validade de Construto

Assim como na investigação apresentada no Capítulo 3, esse estudo contou com um conjunto de avaliações realizadas individualmente pelos desenvolvedores. Em cada avaliação, os desenvolvedores tiveram que reportar uma entre duas opções (**SIM** e **NÃO**) indicando a presença de um *smell* no trecho de código analisado. A limitação das avaliações com apenas estas duas opções pode ser encarada como uma ameaça, uma vez que as mesmas não permitem que os desenvolvedores informem um grau de confiança em suas respostas. Entretanto, tal procedimento foi adotado para garantir que os desenvolvedores fossem capazes de decidir sobre a existência de *bad smell* em um dado trecho de código e permitir a avaliação da eficiência das técnicas de customização analisadas no estudo. Ademais, conforme informado na Seção 3.4, os casos em que os desenvolvedores reportaram dificuldades nas avaliações foram removidos do estudo.

5.4.2 Validade Interna

A execução das técnicas *não-guiadas* adotou as configurações definidas por Fontana *et al.* [7]. Embora o uso de outras configurações pudessem ter influenciado algumas dessas técnicas a incrementar sua eficácia/eficiência, essas configurações foram adotadas porque os autores do trabalho apresentado em [7] indicaram que as técnicas investigadas foram capazes de apresentar uma eficácia mais elevada a partir dessas configurações.

5.4.3 Validade Externa

No estudo apresentado foram utilizados trechos de código extraídos de apenas dois projetos Java. Embora esses projetos possuam diferentes tamanhos e domínios, e ainda tenham sido amplamente utilizados em trabalhos existentes relacionados a *bad smells* [45; 47; 65; 55], as conclusões apresentadas podem não ser contempladas em outros projetos, uma vez que eles podem apresentar características distintas.

Considerando os participantes selecionados, mesmo sabendo que os 62 avaliadores possuem diferentes atuações e experiência, as resultados apresentadas também não podem ser estendidas para outros desenvolvedores, uma vez que eles podem ter diferentes percepções sobre a presença de um mesmo tipo de *smell* [56; 57; 76; 75].

Finalmente, o conjunto de heurísticas considerado para executar os experimentos foi extraído de trabalhos existentes. Nesse sentido, embora algumas dessas heurísticas tenham sido re-utilizadas por diversas técnicas propostas na literatura, a seleção das mesmas pode ter descartado outras heurísticas que poderiam ser mais adequadas para detectar *smells* para os desenvolvedores. Assim, a *Histrategy* poderia apresentar efetividade e eficiência superiores.

5.5 Conclusões do Estudo

No Capítulo 4, foi apresentada uma técnica que utiliza uma customização guiada para produzir estratégias de detecção de *bad smells* de acordo com a percepção dos desenvolvedores. Essa técnica, implementada como *Histrategy*, foi avaliada através de um estudo que investigou sua efetividade e eficiência na detecção de *bad smells*, comparando-a com abordagens não guiadas.

Para a avaliação da *Histrategy* foi realizado um estudo envolvendo 62 desenvolvedores que avaliaram a presença de 8 tipos de *bad smell* em trechos de código de 2 projetos de código aberto. A partir de um conjunto contendo 1.200 avaliações, verificou-se que a *Histrategy* foi capaz de conseguir uma alta eficiência e eficiência ao detectar *smells* individualmente para cada desenvolvedor. Em média, *Histrategy* precisou de apenas 7 exemplos para produzir estratégias eficazes para os desenvolvedores.

Em seguida, o desempenho da *Histrategy* foi comparado com abordagens de customização não guiadas, implementadas com base em seis algoritmos de aprendizagem de máquina

investigados em um recente estudo [7]: *J48*, *JRip*, *Random Forest*, *SMO*, *Naive Bayes* e *SVM*. Os resultados indicaram que a *Histrategy* foi capaz de alcançar uma maior efetividade e eficiência do que as técnicas não guiadas para a grande maioria dos casos analisados. Esses resultados indicam que a customização guiada pode aumentar a eficiência na detecção de *bad smells*.

Capítulo 6

Trabalhos Relacionados

Neste capítulo são apresentados os principais trabalhos relacionadas aos objetivos dessa tese. Assim, na Seção 6.1 são apresentados os trabalhos que investigaram a concordância entre os desenvolvedores na detecção de *bad smells*, bem como os fatores que influenciam em suas avaliações. Por fim, na Seção 6.2 são apresentados trabalhos que utilizam aprendizagem de máquina para promover a customização da detecção de *bad smells*.

6.1 Concordância dos Desenvolvedores na Detecção de *Bad Smells*

Poucos estudos investigaram a similaridade entre as avaliações dos desenvolvedores ao detectar *bad smells* e quais fatores podem influenciar em tais avaliações. Em [56], Mäntylä apresentou um estudo que investigou a concordância entre os desenvolvedores em três tipos de *bad smell*: *Long Method*, *Long Parameter List* e *Feature Envy*). Os desenvolvedores realizaram suas avaliações sobre 10 métodos previamente selecionados que foram extraídos de uma aplicação desenvolvida para o estudo. Os resultados indicaram uma baixa concordância relacionada às avaliações de *Feature Envy* e uma concordância considerável para as avaliações de *Long Method* e *Long Parameter List*. De acordo com tal estudo, a razão para a concordância aferida nas avaliações de *Long Method* e *Long Parameter List* acontece porque tais *smells* são fáceis de entender e detectar, fazendo com que os desenvolvedores os avaliem de forma similar. Adicionalmente, o autor também investigou se métricas de software e as-

pectos demográficos puderam ajudar a predizer a concordância nas avaliações. Os resultados mostraram que o número de linhas de código (`MLOC`) e o número de parâmetros em um método (NPARAM) puderam ser utilizadas como preditores das avaliações relacionadas a *Long Method* e *Long Parameter List*, respectivamente. Por outro lado, os autores não puderam identificar uma influência relevante dos aspectos demográficos sobre a concordância aferida.

Embora os resultados do estudo [56] tenham reportado conclusões relacionadas à concordância entre as avaliações dos desenvolvedores ao detectar *bad smells*, o estudo realizou suas análises sobre apenas três tipos de *bad smell*. Além disso, as avaliações utilizadas no estudo foram realizadas apenas por estudantes de mestrado de um curso específico. Finalmente, todas as avaliações foram realizadas sobre apenas 10 métodos extraídos de um projeto pequeno criado exclusivamente para o experimento. Algumas dessas limitações foram identificadas pelo autor do trabalho.

Em [57], Mäntylä e Lassenius estenderam o estudo apresentado em [56] envolvendo 12 desenvolvedores que avaliaram trechos de código contendo 23 tipos de *bad smell*. Nesse novo estudo, o autor apresentou a definição e um exemplo de cada tipo de *bad smell* para os desenvolvedores, que em seguida avaliaram trechos de código relacionados aos *smells* apresentados. Os desenvolvedores julgaram a presença ou ausência de cada *smell* em trechos de código extraídos de diferentes módulos desenvolvidos por uma única empresa. Os resultados indicaram uma concordância perfeita nas avaliações de cinco desenvolvedores em 1 dos 46 casos analisados no estudo. Este caso aconteceu nas avaliações relacionadas ao *smell Long Method*. Além de *Long Method*, os autores indicaram que as avaliações relacionadas a *Refesuded Bequest* também apresentaram alta concordância. Em contrapartida, as avaliações relacionadas a *Switch Statement*, *Inappropriate Intimacy* e *Message Chains* indicaram uma alta discordância.

Apesar dos resultados reportados em [57] tenham apresentado importantes conclusões acerca da concordância dos desenvolvedores, os autores afirmaram que os dados utilizado em seus experimentos não foram suficientes para fornecer uma maior confiança em suas análises. Em particular, o reduzido número de trechos de código analisados e o baixo número de desenvolvedores envolvidos no experimento criaram dificuldades na obtenção de significância estatística relacionada à concordância dos desenvolvedores. De fato, os valores de concordância foram aferidos considerando, no máximo, seis desenvolvedores que avali-

aram um único trecho de código para cada tipo de *bad smell*. Ademais, suspeita-se que a apresentação dos exemplos de *bad smells* para os avaliadores, possa ter induzido eles a seguir uma heurística pré-definida, criando uma ameaça à validade das conclusões do estudo, como discutido na seção anterior.

Schumacher *et al.* [76] apresentaram um estudo que analisou o entendimento dos desenvolvedores sobre o *smell God Class*. O estudo investigou as dificuldades envolvidas na detecção dessa anomalia incluindo uma investigação sobre a concordância entre os desenvolvedores ao avaliar classes de dois projetos desenvolvidos por eles. Cada projeto foi avaliado por dois desenvolvedores. Os desenvolvedores reportaram se classes dos projetos continham o *bad smell* investigado e responderam um conjunto de questões abertas envolvendo responsabilidades e funções das classes analisadas. Os resultados do estudo apresentaram uma baixa concordância entre os desenvolvedores, similar às conclusões evidenciadas nessa tese. Os autores atribuíram esses resultados devido à complexidade em analisar *God Classes*. Além disso, os autores perceberam que o tamanho e o escopo do experimento apresentou uma ameaça à validade do estudo. Finalmente, o fato dos desenvolvedores de um mesmo projeto discordarem ao avaliar a presença de *smells* em códigos produzidos por eles reforça a ideia que o entendimento dos desenvolvedores sobre tais anomalias é algo individual e depende do ponto de vista de cada indivíduo.

Em [75] os autores estenderam o estudo apresentado em [76] ao executar um experimento controlado com 11 estudantes de graduação. Esses estudantes avaliaram um conjunto de classes extraídas de seis pequenos programas e reportaram a presença do *smell God Class*. Os estudantes indicaram a presença da anomalia respondendo uma de três opções: *Sim*, *Não* e *Talvez*. Conforme observado em [76], os resultados desse novo estudo também apresentaram uma baixa concordância a partir das avaliações consideradas. No final, os autores compararam as avaliações dos estudantes com um oráculo produzido por dois pesquisadores experientes. Novamente uma alta discordância foi observada. De acordo com os autores, a discordância verificada é influenciada devido ao diferente entendimento dos desenvolvedores sobre o conceito de *God Class*, e pelos níveis de exigência em suas avaliações. De fato, mesmo os pesquisadores que produziram o oráculo utilizado no estudo apresentaram divergências ao avaliar as classes analisadas. Em resumo, os estudos apresentados em [76; 75] reforça as conclusões relacionadas à baixa concordância dos desenvolvedores ao avaliar

God Class, conforme apresentado no do Capítulo 3.

6.2 Customização da Detecção por Aprendizagem de Máquina

Estudos anteriores [65; 28; 70] investigaram abordagens para detectar *bad smells*. Algumas das abordagens mais recentes são baseadas no uso de técnicas de aprendizagem de máquina [45; 47; 55; 4; 7]. Assim como apresentado na Seção 5.1 tais abordagens são capazes de customizar a detecção de *bad smells* a partir de um conjunto de exemplos utilizados como treinamento. Embora esses estudos tenham avaliado a eficiência das técnicas baseadas em algoritmos de aprendizagem de máquina para a detecção de *smells*, eles apresentam duas limitações relacionadas a como as abordagens propostas podem ajudar os desenvolvedores a detectar *smells* eficientemente.

Primeiro, os estudos não avaliaram a eficiência das técnicas na detecção de *smells* de acordo com a percepção de cada desenvolvedor. Alguns desses estudos envolveram diferentes desenvolvedores ao experimentar as abordagens propostas. Entretanto, a efetividade de tais abordagens não foi avaliada individualmente para cada desenvolvedor. Nesses casos a efetividade foi computada a partir de um conjunto restrito de exemplos validados por desenvolvedores que foram conduzidos a compartilhar uma mesma percepção sobre alguns *bad smells*. Ao considerar que os desenvolvedores frequentemente têm diferentes percepções sobre a presença de *bad smells* em um programa, como apresentado na Seção 3.2, não se tornava possível saber se as abordagens propostas são eficientes para detectar *smells* individualmente para os desenvolvedores com percepções diferentes. Nesse sentido, o estudo apresentado no Capítulo 5 indicou que muitas dessas abordagens não apresentaram a mesma eficácia.

A segunda limitação está relacionada ao elevado número de exemplos requerido pelas abordagens para customizar a detecção. Dessa forma, a construção de um conjunto de treino contendo um elevado número de exemplos pode requerer muito tempo e esforço dos desenvolvedores, questionando, assim, a aplicabilidade de tais abordagens em um cenário real. A seguir, são apresentados e descritos os trabalhos relacionados que investigaram o uso de algoritmos de aprendizagem de máquina para customizar a detecção de *bad smells*.

Khomh *et al.* [45] propuseram o uso de algoritmos de redes *bayesianas* (BBN[1]) para detectar instâncias do *smell God Class* em dois projetos de código aberto. Os autores envolveram quatro estudantes de graduação para validar manualmente um conjunto de classes, reportando se as mesmas continham uma instância de *God Class* ou não. A partir de um conjunto contendo 15 instâncias do *bad smell*, os autores realizaram um procedimento de *cross*-validação com 3 partes visando calibrar a BBN e avaliar sua efetividade na detecção de *God Class*. Ao final, os autores reportaram que a BBN foi capaz de detectar todos os *smells*, resultando em um *recall* de 100%. Entretanto, o algoritmo classificou erroneamente algumas classes que não continham o *smell* investigado, alcançando uma *precisão* de 68%.

Em um estudo seguinte [47], os mesmos autores estenderam o trabalho anterior [45] aplicando as redes *bayesianas* para detectar três tipos de *bad smell*. Nesse trabalho os autores seguiram o mesmo procedimento para produzir o conjunto de treinamento que foi utilizado para calibrar as BBNs. Novamente as BBNs produzidas foram capazes de alcançar um *recall* de 100%. Por outro lado, elas apresentaram uma *precisão* pior ao detectar os *smells* investigados. Nesse estudo a *precisão* obtida foi de aproximadamente 33%.

O trabalho apresentado em [55] avaliou a eficiência de uma abordagem baseada em uma Máquina de Vetores de Suporte (*SVM*[2]) para detectar *smells*. A abordagem proposta foi avaliada a partir de um conjunto de treino contendo 250 exemplos manualmente validados por diferentes desenvolvedores. Os oráculos utilizados no experimento não consideraram a customização para cada desenvolvedor que participou da construção do oráculo. De acordo com os resultados apresentados, a abordagem baseada em *SVM* foi capaz de alcançar, em média, um *recall* de 70% e uma *precisão* igual a 74%.

Em [4] os autores propuseram o uso de algoritmos baseados em *Árvores de Decisão* para detectar *smells*. Similar a trabalhos anteriores, os autores utilizaram um conjunto de treino contendo um grande número de exemplos e validados por diferentes desenvolvedores. Entretanto as análises não individualizaram os resultados. Por fim o trabalho reportou que o algoritmo foi capaz de alcançar, em média, um *recall* de 71% e uma *precisão* de 78%.

Nesse contexto, Fontana *et al.* [7] apresentaram um grande estudo que comparou e experimentou diferentes configurações de seis algoritmos de aprendizagem de máquina para

[1] do Inglês *Bayesian Belief Network*

[2] do Inglês *Support Vector Machine*

a detecção de *bad smells*. Como treino, os autores consideraram um conjunto composto de mais de 1900 exemplos de trechos de código manualmente validados por diferentes desenvolvedores. Novamente, as análises não foram individualizada para cada participante. Mesmo assim, os autores reportaram que todas as técnicas apresentaram uma elevada eficácia, com destaque para dois algoritmos baseados em *Árvores de Decisão* (*J48* e *Random Forest* [64]). Por fim, os autores informaram que as técnicas investigadas precisavam de centenas de exemplos de treino para alcançar uma efetividade de pelo menos 95%.

Capítulo 7

Considerações Finais

Este trabalho apresentou uma técnica que utiliza uma customização guiada para detectar *bad smells* que são sensíveis à percepção do desenvolvedor. Essa técnica, definida como *Histrategy*, visa melhorar a efetividade e a eficiência na detecção de *smells* obtida a partir do estado da arte das abordagens de customização existentes. Nesse contexto, dois importantes estudos foram apresentados ressaltando a relevância do problema atacado pela *Histrategy* e avaliando sua performance na detecção de *bad smells*.

O primeiro estudo, apresentado no Capítulo 3, investigou a concordância dos desenvolvedores ao detectar *smells* e analisou possíveis fatores que poderiam influenciar em tal concordância. Dessa forma, foi realizado um experimento envolvendo 75 desenvolvedores que avaliaram 15 tipos de *bad smell* em trechos de código de 5 projetos de código aberto. No total mais de 2.700 avaliações resultaram em uma grande porção de dados quantitativos e qualitativos.

A partir dos dados coletados, verificou-se uma grande discordância entre as avaliações dos desenvolvedores sobre todos os tipos de *smell* investigados no estudo. Essa discordância foi verificada, inclusive, para anomalias reportadas como "simples" em trabalhos anteriores, tais como *Long Method* e *Long Parameter List*. Esses resultados apresentaram evidências que, em geral, os desenvolvedores detectam *bad smells* de diferentes formas, e seus julgamentos pessoais devem ser considerados em técnicas que propõem uma detecção automática para eles. Dessa forma, a performance de tais técnicas não deveriam ser avaliadas a partir da comparação dos resultados com um oráculo definido por "especialistas", como observado em trabalhos anteriores [45; 65; 55; 70;

72].

Já a investigação dos fatores que poderiam ter influenciado a concordância indicou que, mesmo quando as avaliações de grupos de desenvolvedores com características comuns, relacionadas à atuação e experiência, a concordância aferida não foi consistentemente mais elevada. Entretanto, percebeu-se que os desenvolvedores que seguiram a mesma heurística de detecção apresentaram uma concordância consistente, quando analisados separadamente. Esses resultados revelaram que a heurística seguida pelos desenvolvedores durante suas avaliações possuem um papel importante para determinar como eles detectam determinadas anomalias. Além disso, o alto número de heurísticas reportadas sugere que esse fator deve ser considerado na proposição de técnicas efetivas para a detecção automática.

As conclusões obtidas na investigação da concordância dos desenvolvedores levaram à criação da *Histrategy* (Capítulo 4), que foi avaliada em um segundo estudo, apresentado no Capítulo 5. Nesse novo estudo, as avaliações desenvolvedores que participaram do primeiro estudo foram utilizadas, individualmente, para verificar a performance da customização guiada na produção de estratégias relacionadas a oito tipos de *bad smell*. Os resultados indicaram que a *Histrategy* foi capaz de alcançar uma alta eficácia e eficiência ao detectar *smells* para cada desenvolvedor.

Em seguida, o estudo comparou a efetividade e eficiência da *Histrategy* com abordagens de customização não-guiadas, implementadas por seis técnicas de aprendizagem de máquina investigadas em um estudo recente [7]. Os resultados indicaram que a *Histrategy* foi capaz de apresentar uma performance superior às técnicas não-guiadas em uma grande maioria dos casos analisados, sugerindo que a customização guiada pode melhorar a eficácia e eficiência na detecção de *bad smells*.

7.1 Revisão das Contribuições

Os esforços dedicados a esse trabalho resultaram nas seguintes contribuições:

- Apresentação de um estudo (Capítulo 3) que investigou a concordância entre desenvolvedores ao detectar instâncias de 15 tipos de *bad smell* diferentes. Esse estudo também analisou possíveis fatores que poderiam influenciar na concordância dos desenvolvedores ao detectarem um mesmo tipo de *bad smell*. Os resultados desse estudo incre-

mentam o conhecimento acerca da percepção dos desenvolvedores sobre *bad smells*, favorecendo a proposição de novas técnicas de detecção mais efetivas.

- Produção e publicação [38] de uma base de dados contendo 2.700 avaliações, onde 75 desenvolvedores reportaram a presença de 15 tipos de *bad smell* em projetos de código aberto. A ausência de bases de dados desse tipo, individualizadas por desenvolvedores, faz com que essa base possa apoiar diversas pesquisas que investigam a percepção dos desenvolvedores sobre *bad smells*. Até a produção desse documento, o autor desconhece outras bases com tamanho e conteúdo similares.

- Criação de uma técnica para a customização guiada de estratégias para a detecção e *bad smells* (Capítulo 4). Essa técnica, definida como *Histrategy*, visa incrementar a efetividade e eficiência da detecção e utilizou como base as conclusões obtidas no estudo sobre a concordância dos desenvolvedores [37].

- Disponibilização de uma versão inicial da ferramenta que implementa a customização guiada a partir de um componente (*plugin*) para o ambiente de desenvolvimento integrado Eclipse [36; 3]. Essa ferramenta permite a utilização da técnica proposta em cenários reais.

- Apresentação de um estudo que avaliou o desempenho de técnicas de customização não-guiadas na produção de estratégias para desenvolvedores com percepções diferentes [35].

7.2 Implicações no uso da *Histrategy*

Os resultados apresentados nesse documento indicam que a customização guiada implementada pela *Histrategy* pode ajudar desenvolvedores a detectar *bad smells* com eficácia. Além disso, o baixo número de exemplos necessários para a customização das estratégias revelam que *Histrategy* foi mais eficiente que as técnicas de aprendizagem propostas em trabalhos anteriores. Não obstante, a utilização da técnica proposta em cenários reais de desenvolvimento requer alguns cuidados para que a detecção de *bad smells* possa, de fato, contribuir para o desenvolvimento sem prejudicar a produtividade dos desenvolvedores.

Conforme descrito no Capítulo 4, a customização promovida pela *Histrategy* requer um conjunto de trechos de código com a indicação da presença ou ausência de um determinado tipo de *bad smell*. Esse conjunto habilita o aprendizado necessário para customização da detecção a partir de exemplos que demonstrem a percepção do desenvolvedor. Embora o esforço para a produção desse conjunto de exemplos tenha sido reduzido quando comparado ao esforço requerido por outras abordagens, a realização dessa tarefa ainda demanda tempo e esforço do desenvolvedor. Entretanto, acredita-se que após a criação de um dado conjunto de exemplos a re-utilização da *Histrategy* em outros cenários similares possa fazer uso dos exemplos já anotados anteriormente pelo desenvolvedor. Nesse caso, o esforço inicial poderia ser eliminado ou reduzido.

Um outro aspecto importante pode ser observado na utilização da *Histrategy* em cenários onde um mesmo software é construído por desenvolvedores diferentes. Uma vez que os desenvolvedores de uma equipe podem ter percepções distintas, a customização e a consequente detecção dos trechos de código que devem ser refatorados poderia causar divergência de opiniões sobre o que seria implementado. Nesses casos, a *Histrategy* não deveria ser utilizada por todos os desenvolvedores, mas sim por um pequeno grupo de indivíduos responsáveis por manter a qualidade do código ao longo do desenvolvimento. Esse grupo, que poderia ser composto pelo analista de qualidade, pelo gerente do projeto ou por um líder da equipe, deveria fornecer os exemplos durante o processo de customização a fim de produzir as estratégias de detecção adequadas para o projeto. Em seguida, tais estratégias poderiam ser aplicadas sistematicamente por todos os desenvolvedores da equipe para verificar se o código que eles estão construindo apresentam alguma anomalia para o projeto.

7.3 Limitações

O trabalho descrito nesse documento possui algumas limitações conhecidas. Por exemplo, a efetividade das estratégias produzidas pela *Histrategy* depende do conjunto de heurísticas apresentados como entrada. Nesse caso, pode ocorrer que um determinado desenvolvedor utilize uma heurística não considerada durante o processo de customização. Dessa forma, a estratégia produzida poderá não ser efetiva para o desenvolvedor.

Além de considerar as heurísticas definidas como entrada, a produção de estratégias pro-

movida pela *Histrategy* é guiada a partir de um conjunto de exemplos de classificação fornecidos pelo desenvolvedor durante as iterações do processo de customização. Assim, o bom funcionamento da *Histrategy* depende que o desenvolvedor seja capaz de discernir se um dado trecho de código contém ou não o *smell* avaliado. Nesse sentido, o provimento de classificações que não reflitam a confiança do desenvolvedor no julgamento dos trechos de código avaliados, poderá prejudicar a efetividade da estratégia produzida.

Uma outra limitação consiste na seleção dos trechos de código apresentados ao desenvolvedor durante o processo de customização. A atual seleção considera a proximidade dos valores de métricas de tais trechos com os limiares definidos pela *melhor* estratégia da iteração. Esse procedimento considera que as estratégias definidas como entrada do processo apresentem limiares próximos dos adotados pelo desenvolvedor, requerendo, apenas um ajuste fino dos limiares adotados. Entretanto, pode ser que a heurística adotada pelo desenvolvedor em suas classificações considere um limiar muito distante dos definidos nas estratégias iniciais. Dessa forma, a definição de um limiar adequado para o desenvolvedor poderá requerer de uma grande quantidade de iterações até que um trecho de código com métricas próximas ao limiar adotado por ele seja atingido.

Por fim, é sabido que a customização promovida pela *Histrategy* considera as diferentes heurísticas adotadas pelo desenvolvedor durante a detecção de *bad smells*. Embora alguns estudos já tenham sugerido que o domínio da aplicação [30], a arquitetura do software desenvolvido [29] e até a utilização de padrões de projeto e outros aspectos contextuais [41] podem determinar a indicação ou não de *bad smells*, esses aspectos não foram considerados. Assim, as estratégias customizadas podem não apresentar um desempenho satisfatório em diferentes contextos de desenvolvimento.

7.4 Trabalhos Futuros

A complementação do trabalho apresentado nesse documento deve incluir novos esforços que permitam uma investigação mais profunda do tema abordado. Entre outras coisas espera-se repetir o estudo sobre a concordância dos desenvolvedores em um cenário mais controlado com indivíduos de uma mesma empresa avaliando trechos de código desenvolvido por eles. Esse estudo poderá dar mais detalhes de como a divergência de opiniões ocorre em cenários

mais restritos que, por vezes, seguem padrões de codificação pré-estabelecidos.

Como mencionado anteriormente, alguns estudos já sugeriram que o domínio da aplicação [30], a arquitetura do software desenvolvido [29] e até a utilização de padrões de projeto e outros aspectos contextuais [41] podem determinar a indicação ou não de *bad smells*. Entretanto, as evidências apresentadas ainda não realizaram um estudo dedicado à investigação da influência desses fatores de acordo com a diferente percepção dos desenvolvedores sobre *bad smells*. Assim, pretende-se, como trabalho futuro, estender a base de *smells* anotados de modo a permitir a investigação desses e outros fatores que possam influenciar a detecção de *smells* por desenvolvedores.

Outro trabalho futuro consiste na investigação de outros fatores para melhorar a customização. Afinal, acredita-se que outros fatores, relacionados ao sistema avaliado, possa influenciar na detecção de *smells*. Assim, pretende-se estender a *Histrategy* de modo a contemplar uma customização mais robusta, indicando novas heurísticas e novas métricas. Além disso, espera-se realizar experimentos que possam adaptar a utilização de alguns algoritmos de aprendizagem de máquina ao processo de customização guiada.

Por fim, pretende-se investigar o formato da disponibilização da *Histrategy*, a partir de estudos de usabilidade que favoreçam a adoção de uma ferramenta de detecção no processo de desenvolvimento.

Bibliografia

[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 181–190. IEEE, 2011.

[2] Alain Abran and Hong Nguyenkim. Measurement of the maintenance process from a demand-based perspective. *Journal of Software Maintenance: Research and Practice*, 5(2):63–90, 1993.

[3] Henrique Ferreira Alves. Uma Ferramenta para a Detecção de *Code Smells* Utilizando Feedback do Desenvolvedor, 2015. Trabalho de Conclusão de Curso de Bacharelado em Ciência da Computação. Universidade Federal de Alagoas - Campus Arapiraca.

[4] Lucas Amorim, Evandro Costa, Nuno Antunes, Baldoino Fonseca, and Marcio Ribeiro. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In *Proceedings of the 2015 IEEE 26th International Symposium on Software Reliability Engineering*, ISSRE '15, pages 261–269, Washington, DC, USA, 2015. IEEE Computer Society.

[5] Apache. The Apache Xerces Project. http://xerces.apache.org/. Abril, 2017.

[6] Apache Software Foundation. ASF - JIRA [CSV-76]. https://issues.apache.org/jira/browse/CSV-76, 2012. Abril, 2017.

[7] Francesca Arcelli Fontana, Mika Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191, June 2015.

[8] Pratibha Atri. *Detect malodorous software pattern and refactor them*. PhD thesis, San Diego State University, 2013.

[9] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[10] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[11] Jagdish Bansiya. *A Hierarchical Model for Quality Assessment of Object-oriented Designs*. PhD thesis, University of Alabama in Huntsville, 1997. AAI9834310.

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

[13] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87. ACM, 2000.

[14] Isela Macia Bertran, Alessandro Garcia, Christina Chavez, and Arndt von Staa. Enhancing the detection of code anomalies with architecture-sensitive strategies. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, pages 177–186, 2013.

[15] James M. Bieman and Byung kyoo Kang. Cohesion and reuse in an object-oriented system. In *ACM SIGSOFT Software Engineering Notes*, volume 20, pages 259–262. ACM, 1995.

[16] Jim Bird. Building Real Software: Rule of 30. When is a method, class or subsystem too big? http://swreflections.blogspot.com.br/2012/12/rule-of-30-when-is-method-class-or.html, 2012. Abril, 2017.

[17] Checkstyle. Checkstyle. http://checkstyle.sourceforge.net/. Abril, 2017.

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

[19] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, Jun 1994.

[20] Yania Crespo, Carlos López, Raul Marticorena, and Esperanza Manso. Language independent metrics support towards refactoring inference. In *9th ECOOP Workshop on QAOOSE*, volume 5, pages 18–29. Citeseer, 2005.

[21] R. Geoff Dromey. A model for software product quality. *IEEE Transactions on Software Engineering*, 21(2):146–162, February 1995.

[22] R. Geoff Dromey. Cornering the chimera. *IEEE Software*, 13(1):33–43, January 1996.

[23] Patrick Dubroy. Are short methods actually worse? http://dubroy.com/blog/method-length-are-short-methods-actually-worse/, 2009. Abril, 2017.

[24] Eclipse. Eclipse. http://www.eclipse.org/. Abril, 2017.

[25] Amin Milani Fard and Ali Mesbah. Jsnose: Detecting javascript code smells. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 116–125. IEEE, 2013.

[26] Manuele Ferreira, Eiji Barbosa, Isela Macia, Roberta Arcoverde, and Alessandro Garcia. Detecting architecturally-relevant code anomalies: a case study of effectiveness and effort. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1158–1163. ACM, 2014.

[27] Joseph L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

[28] F. A. Fontana, P. Braione, and M. Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2), 2012.

[29] Francesca Arcelli Fontana, Vincenzo Ferme, Alessandro Marino, Bartosz Walter, and Pawel Martenka. Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains. *2013 IEEE International Conference on Software Maintenance*, pages 260–269, September 2013.

[30] Francesca Arcelli Fontana, Vincenzo Ferme, and Stefano Spinelli. Investigating the impact of code smells debt on quality code evaluation. *2012 3rd International Workshop on Managing Technical Debt (MTD)*, pages 15–22, June 2012.

[31] Francesca Arcelli Fontana, Elia Mariani, Andrea Mornioli, Raul Sormani, and Alberto Tonello. An Experience Report on Using Code Smells Detection Tools. *2011 IEEE 4th International Conference on Software Testing, Verification and Validation Workshops*, pages 450–457, March 2011.

[32] Martin Fowler. Refactoring: Improving the design of existing code. In *Extreme Programming and Agile Methods - XP/Agile Universe 2002, Second XP Universe and First Agile Universe Conference Chicago, IL, USA, August 4-7, 2002, Proceedings*, page 256, 2002.

[33] GanttProject. GanttProject. http://www.ganttproject.biz/. Abril, 2017.

[34] Rohit Gopalan. *Automatic detection of code smells in Java source code*. PhD thesis, Dissertation for Honour Degree, The University of Western Australia, 2012.

[35] Mário Hozano, Nuno Antunes, Baldoino Fonseca, and Evandro Costa. Evaluating the accuracy of machine learning algorithms on detecting code smells for different developers. In *ICEIS 2017 - Proceedings of the 19th International Conference on Enterprise Information Systems, Porto, Portugal, April 26-29*, 2017 (aceito para publicação).

[36] Mario Hozano, Henrique Ferreira, Italo Silva, Baldoino Fonseca, and Evandro Costa. Using developers' feedback to improve code smell detection. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1661–1663. ACM, 2015.

[37] Mário Hozano, Alessandro Garcia, Nuno Antunes, Baldoino Fonseca, and Evandro Costa. Smells are sensitive to developers! on the efficiency of (un)guided customized detection. In *ICPC 2017 - Proceedings of the 25th IEEE International Conference on Program Comprehension, Buenos Aires, Argentina, May 22-23*, 2017 (aceito para publicação).

[38] Mário Hozano. Support Material - Are you smelling it? Investigating how similar developers detect code smells. http://bit.ly/2fFFAcz. Junho, 2017.

[39] inFusion. inFusion. https://www.intooitus.com/products/infusion. Abril, 2017.

[40] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[41] Sebastian Jancke. *Smell Detection in Context*. PhD thesis, University of Bonn, 2010.

[42] JDeodorant. JDeodorant. http://www.jdeodorant.com/. Abril, 2017.

[43] JEdit. jEdit - Programmer's Text Editor. http://www.jedit.org/. Abril, 2017.

[44] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *ICSM 2001, Proceedings of the International Conference on Software Maintenance*, pages 736–743, Florence, Italy, November 6–10, 2001.

[45] F Khomh, S Vaucher, Y. Guéhéneuc, and H. Sahraoui. A bayesian approach for the detection of code and design smells. In *Quality Software, 2009. QSIC'09. 9th International Conference on*, pages 305–314. IEEE, 2009.

[46] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, August 2011.

[47] Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. BDTEX: A GQM-based Bayesian Approach for the Detection of Antipatterns. *J. Syst. Softw.*, 84(4):559–572, April 2011.

[48] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 50. ACM, 2012.

[49] Jochen Kreimer. Adaptive Detection of Design Flaws. *Electronic Notes in Theoretical Computer Science*, 141(4):117–136, December 2005.

[50] J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[51] Michele Lanza, Radu Marinescu, and Stéphane Ducasse. *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[52] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *J. Syst. Softw.*, 23(2):111–122, November 1993.

[53] Hui Liu, Limei Yang, Zhendong Niu, Zhyi Ma, and Weizhong Shao. Facilitating software refactoring with appropriate resolution order of bad smells. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 265–268, New York, NY, USA, 2009. ACM.

[54] Isela Macia, Roberta Arcoverde, Alessandro Garcia, Christina Chavez, and Arndt von Staa. On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms. *2012 16th European Conference on Software Maintenance and Reengineering*, pages 277–286, March 2012.

[55] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabane, Yann-Gael Gueheneuc, and Esma Aimeur. SMURF: A SVM-based Incremental Anti-pattern Detection Approach. *2012 19th Working Conference on Reverse Engineering*, pages 466–475, October 2012.

[56] M. Mäntylä. An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In *Empirical Software Engineering, International Symposium on*, pages 17–18, November 2005.

[57] M. Mäntylä and C. Lassenius. *Subjective evaluation of software evolvability using code smells: An empirical study*, volume 11. Springer, May 2006.

[58] M. Mäntylä, J. Vanhanen, and C. Lassenius. Bad smells - humans as code critics. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 399–408, Sept 2004.

[59] Radu Marinescu. Detecting Design Flaws via Metrics in Object-Oriented Systems A Metrics-Based Approach for Problem Detection. *International Conference and Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 173–182, 2001.

[60] Radu Marinescu. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 350–359, Washington, DC, USA, 2004. IEEE Computer Society.

[61] Radu Marinescu. Measurement and quality in object-oriented design. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 701–704, Sept 2005.

[62] Jim A. McCall, Paul K. Richards, and Gene F. Walters. Factors in software quality. volume i. concepts and definitions of software quality. Technical report, DTIC Document, 1977.

[63] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[64] Tom M. Mitchell. *Machine learning*. McGraw-Hill series in computer science. McGraw-Hill, Boston (Mass.), Burr Ridge (Ill.), Dubuque (Iowa), 1997.

[65] N. Moha, Y.-G. Gueheneuc, L. Duchien, and a. F. Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, January 2010.

[66] M.J. Munro. Product Metrics for Automatic Identification of "Bad Smell"Design Problems in Java Source-Code. *11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 15–15, 2005.

[67] Willian Oizumi, Alessandro Garcia, Leonardo da Silva Sousa, Bruno Cafeo, and Yixue Zhao. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 440–451, New York, NY, USA, 2016. ACM.

[68] William F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

[69] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 268–278, November 2013.

[70] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Mining Version Histories for Detecting Code Smells. *IEEE Transactions on Software Engineering*, 5589(c):1–1, 2014.

[71] Fabio Palomba, GABRIELE BAVOTA, ROCCO OLIVETO, and ANDREA DE LUCIA. Anti-pattern detection: Methods, challenges, and open issues. *ADVANCES IN COMPUTERS, VOL 95*, 95:201–238, 2014.

[72] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 101–110, September 2014.

[73] PMD. PMD. http://pmd.sourceforge.net/. Abril, 2017.

[74] Donald Bradley Roberts and Ralph Johnson. *Practical analysis for refactoring*. University of Illinois at Urbana-Champaign, 1999.

[75] José A. M. Santos, Manoel G. de Mendonça, and Carlos V. A. Silva. An exploratory study to investigate the impact of conceptualization in god class detection. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, EASE '13, pages 48–59, New York, NY, USA, 2013. ACM.

[76] Jan Schumacher, Nico Zazworka, Forrest Shull, Carolyn Seaman, and Michele Shaw. Building empirical support for automated code smell detection. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '10*, pages 8:1–8:10, 2010.

[77] SciTools. Understand. http://scitools.com/. Abril, 2017.

[78] C. B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, Jul 1999.

[79] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334, 2014.

[80] Stack Exchange. What is the ideal length of a method? http://programmers.stackexchange.com/questions/133404/what-is-the-ideal-length-of-a-method, 2012. Abril, 2017.

[81] Stack Overflow. How many lines of code should a function/procedure/method have? http://stackoverflow.com/questions/611304/how-many-lines-of-code-should-a-function-procedure-method-have, 2009. Abril, 2017.

[82] Stack Overflow. How many lines of code is too many? http://stackoverflow.com/questions/20981/how-many-lines-of-code-is-too-many, 2010. Abril, 2017.

[83] Tigris.org. ArgoUML. http://argouml.tigris.org/. Junho, 2017.

[84] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. Detecting defects in object-oriented designs: Using reading techniques to increase software quality. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 47–56, New York, NY, USA, 1999. ACM.

[85] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 97–106. IEEE, 2002.

[86] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[87] Gueheneuc Y., H. Sahraoui, and F. Zaidi. Fingerprinting design patterns. In *11th Working Conference on Reverse Engineering*, pages 172–181, Nov 2004.