Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

A influência da integração contínua na identificação e tratamento de bad smells

Seany Caroliny Oliveira Silva

JUIZ DE FORA
FEVEREIRO, 2022

A influência da integração contínua na identificação e tratamento de bad smells

Seany Caroliny Oliveira Silva

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Gleiph Ghiotto Lima de Menezes

JUIZ DE FORA

FEVEREIRO, 2022

A influência da integração contínua na identificação e tratamento de bad smells

Seany Caroliny Oliveira Silva

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Gleiph Ghiotto Lima de Menezes
Doutor em Computação

Andre Luiz de Oliveira
Doutor em Ciência da Computação

Igor de Oliveira Knop
D. Sc. Modelagem Computacional

JUIZ DE FORA
10 DE FEVEREIRO, 2022

Aos meus amigos e irmãos.

Aos pais, pelo apoio e sustento.

Resumo

Bad smells são considerados sintomas de decisões de *design* equivocadas tomadas ao longo do desenvolvimento de um *software*. A Integração Contínua (IC) é uma técnica que monitora de modo contínuo os artefatos de um projeto de *software* para identificar inconsistências como um código-fonte que não compila, não passa pelos testes ou possuem algum indicativo da presença de *bad smells*. Entretanto, nada se sabe da influência da adoção de IC no ciclo de vida dos *bad smells*. Este trabalho tem o objetivo de averiguar se a IC influencia na identificação, tratamento e no tempo de vida dos *bad smells*. Os experimentos foram realizados em oito sistemas de código aberto divididos em dois grupos de quatro projetos cada: os que adotam IC e os que não adotam. Para isso, foi realizada uma análise da frequência em que os *bad smells* são identificados, se eles são tratados e quanto tempo persistem no histórico do *software* até serem tratados. Os resultados mostraram que a IC teve impacto positivo em todos esses três critérios: os *bad smells* são menos frequentes em projetos que utilizam IC; o volume de tratamento é maior no conjunto de projetos que utilizam IC; e o ciclo de vida dos *bad smells* é menor no conjunto de projetos que utilizam IC.

Palavras-chave: *bad smell*; integração contínua; mineração de repositórios de software.

Abstract

Bad smells are considered symptoms of wrong design decisions made during software development. Continuous Integration (CI) is a technique that continuously monitors the artifacts of a software project to identify inconsistencies such as source code that does not compile, does not pass the tests, or has some indication of the presence of bad smells. However, there are no studies about the influence of CI adoption on the life cycle of bad smells. This work investigates whether CI influences the identification, treatment, and life span of bad smells. We performed the experiments in eight open-source systems divided into two groups of four projects each: those that adopt CI and those that do not. To answer these questions, we analyzed the frequency in which the bad smells are identified, the frequency they are treated, and how long they persist in the software history until being treated. The results show that CI has positively impacted all of these criteria: bad smells are less frequent in projects that use CI; the volume of treatments is higher in the set of projects that use CI; and the life cycle of bad smells is shorter in the projects that use CI.

Keywords: *bad smell; continuous integration; mining software repositories.*

Agradecimentos

A minha mãe e todos os meus parentes, pelo encorajamento e apoio.

Ao professor Gleiph pela orientação, amizade e principalmente, pela paciência, sem a qual este trabalho não se realizaria.

Aos meus amigos e professores pelos seus ensinamentos que me ajudaram durante esses anos, que contribuíram de algum modo para o nosso enriquecimento pessoal e profissional.

"Lembra que o sono é sagrado e alimenta de horizontes o tempo acordado de viver".

Beto Guedes (Amor de Índio)

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

DCC	Departamento de Ciência da Computução
UFJF	Universidade Federal de Juiz de Fora
MRS	Mineração de Repositórios de *software*
SCV	Sistema de Controle de Versão
IC	Integração Contínua
SIC	Servidores de Integração Contínua

1 Introdução

1.1 Apresentação do tema

Ao longo da evolução do *software*, o código vai se tornando mais complexo e sujeito à modificação por diversos motivos. Lehman e Ramil (2000) constataram que as mudanças são imprescindíveis para manter o sucesso de um sistema de *software*. Entretanto, mudanças constantes podem inserir inconsistências no *design* do *software*, causando a degradação de sua estrutura (PARNAS, 1994). Os *bad smells* são considerados sintomas de escolhas ruins de *design* e implementação (FOWLER, 1999), que aumentam a tendência a mudanças e falhas no *software* (KHOMH; PENTA; GUEHENEUC, 2009), contribuindo para o envelhecimento de código.

De acordo com Fowler (2010), a Integração Contínua (IC) é uma prática de desenvolvimento de *software* onde os membros de uma equipe integram seu trabalho frequentemente (i.e., ao menos uma vez no dia), levando a várias integrações diárias. Cada integração é verificada por um processo automatizado de construção, composto por tarefas como compilação, testes e execução de *scripts* para identificar erros o mais rápido possível. Deste modo, a IC apresenta um rápido *feedback* que é a chave para resolução de problemas. Por exemplo, com a execução de *scripts* é possível indicar a presença de *bad smells* como *Long Method* se um método possui mais de 100 linhas. Além disso, IC possui um conjunto de 10 práticas que, segundo Fowler (2010), possibilitam extrair melhores resultados e diminuir os riscos, como realizar a identificação e tratamento de inconsistências de *software* como os *bad smells*.

1.2 Descrição do problema

Na literatura, foram identificadas diversas pesquisas sobre detecção de *bad smells*. Dentre elas, estudos que propuseram abordagens de identificação de *bad smells* a partir da investigação das informações do código-fonte por meio da Mineração de Repositórios de

software (MRS) (*e,g.,* (PETERS; ZAIDMAN, 2012; PALOMBA et al., 2014; FU; SHEN, 2015; TUFANO et al., 2015)). Pesquisas com MRS são focadas na exploração dos dados históricos relacionados com repositórios de *software* (KAGDI; COLLARD; MALETIC, 2007), o que é eficaz para fazer o acompanhamento da evolução dos *bad smells* ao longo do projeto. Entretanto, apesar da existência de diversos estudos sobre *bad smells* na literatura, nenhum trabalho relaciona a identificação de *bad smells* com a utilização de IC.

1.3 Motivação e objetivo

De acordo com Tufano et al. (2015), a introdução de *bad smells* pode ser evitada através de uma verificação contínua da qualidade em cada versão do projeto. Então, este trabalho visa verificar se a IC auxilia na identificação e tratamento de *bad smells* por meio de um estudo exploratório de repositórios de controle de versão disponíveis em plataformas, como o *GitHub*[1]. Como a IC faz o monitoramento contínuo do código, este estudo visa descobrir se a utilização de IC em projetos de *software* tem influência na ocorrência de *bad smells* tanto na sua identificação quanto no tratamento. O objetivo é verificar se a IC tem influência no ciclo de vida dos *bad smells*. Logo, para atingir esse objetivo, é proposta uma investigação em dois grupos de projetos, projetos que utilizam IC e que não utilizam, para identificar a frequência na qual os *bad smells* são identificados, se eles são tratados e quanto tempo persistem no histórico do projeto de *software* até serem tratados.

1.4 Organização

Este trabalho está organizado em quatro capítulos, além deste. No Capítulo 2 é apresentada a fundamentação teórica necessária à compreensão das contribuições deste trabalho de conclusão de curso. O Capítulo 3 descreve a abordagem proposta e mostra como foi realizada a implementação para estudar o ciclo de vida dos *bad smells*. O Capítulo 4 descreve a avaliação aplicada em oito projetos de *software* mostrando como foram realizadas as análises, apresentando as discussões dos resultados obtidos e as ameaças à

[1] https://github.com/

validade. Por fim, o Capítulo 5 contém as conclusões, contribuições e algumas sugestões de trabalhos futuros.

2 Fundamentação teórica

Neste capítulo são apresentados os conceitos utilizados para descrever o estudo sobre a influência da Integração Contínua no ciclo de vida dos *bad smells*. A Seção 2.1 descreve a mineração de repositórios de *software* com o enfoque aos Sistemas de Controle de Versão (SCV) e às técnicas de Mineração de Repositórios. A Seção 2.2 apresenta as métricas de *software* utilizadas para identificação de *bad smells*. A Seção 2.3 apresenta os principais tipos de *bad smells* encontrados na literatura e detalha como são realizadas a identificação e o tratamento de cinco dos tipos de *bad smells*. A Seção 2.4 aborda o tema integração contínua e um conjunto de boas práticas que devem ser seguidas quando esta técnica é utilizada. Finalmente, a Seção 2.5 apresenta os trabalhos relacionados aos temas: *bad smells* e integração contínua.

2.1 Mineração de repositórios de *software*

O termo Mineração de Repositórios de *software* (MRS) é utilizado para descrever uma extensa categoria de investigações sobre os dados presentes em repositórios de *software*. Os dados extraídos de repositórios através de técnicas de MRS proporcionam uma visão sobre a evolução do sistema de *software*. Por exemplo, os repositórios possuem informações como: quais foram as mudanças realizadas pelos desenvolvedores, quem as realizou, o motivo da mudança e quando ela foi realizada (KAGDI; COLLARD; MALETIC, 2007).

Na literatura, há vários estudos que aplicam a MRS. Por exemplo, Hassan (2008) apresenta um breve histórico da área de MRS, discutindo as contribuições e resultados da utilização de técnicas MRS para auxiliar nas pesquisas de *software* em áreas como reutilização de *software* (MANDELIN et al., 2005; XIE; PEI, 2006) e busca por melhoria da experiência do usuário (MICHAIL; XIE, 2005; MOCKUS; ZHANG; LI, 2005). O autor conclui que os repositórios podem ser minerados para extrair informações úteis e padrões de projetos de *software*.

Para uma melhor compreensão do funcionamento da MRS, nesta seção é apre-

sentado como os SCV são organizados e manuseados. Além disso, são descritas algumas aplicações e técnicas de MRS. Esta seção está organizada com as seguintes subseções: a Subseção 2.1.1 define o sistema de controle de versão e a Subseção 2.2.2 mostra algumas técnicas de mineração de repositório.

2.1.1 Sistema de controle de versão

O Sistema de Controle de Versão (SCV) é capaz de gerenciar os artefatos (*e.g.*, código-fonte e documentação) utilizados durante o desenvolvimento de sistemas de *software*. Sua adoção possibilita que os desenvolvedores versionem os artefatos e tenham acesso à todas as versões do projeto em que trabalham (ZOLKIFLI; NGAH; DERAMAN, 2018). Além disso, a utilização de um SCV auxilia o gerenciamento dos artefatos de maneira automatizada evitando tarefas manuais que podem ser complexas e propensas a erros (BERCZUK, 2003).

Os SCV podem ser classificados em centralizados ou distribuídos. Os SCV centralizados, como ilustrado na Figura 2.1, contêm um repositório central responsável por armazenar todas as versões dos artefatos. Os repositórios podem possuir diversas cópias de trabalho, onde o desenvolvedor pode realizar alterações e sincronizá-las com repositório central. Deste modo, o ciclo de trabalho de um desenvolvedor é iniciado com o comando de *checkout* para criar uma cópia de trabalho local. A partir disso, o desenvolvedor pode executar o comando de *commit* para enviar as alterações realizadas localmente para o repositório, gerando uma nova versão. Alternativamente, o desenvolvedor pode executar o comando *update* para trazer as contribuições dos outros desenvolvedores para a cópia de trabalho local. Alguns exemplos de SCV centralizados são: CVS (BAR; FOGEL, 2003) e *Subversion* (FITZPATRICK; PILATO; COLLINS-SUSSMAN, 2004).

Por outro lado, nos SCV distribuídos, como mostrado na Figura 2.2, o desenvolvedor pode gerenciar as alterações em múltiplos repositórios que podem ser locais ou remotos. Deste modo, o histórico do projeto é mantido nas cópias de trabalho e o desenvolvedor utiliza o comando *clone* para fazer a cópia do repositório remoto para sua estação de trabalho, no qual ele pode executar as alterações desejadas, salvar os arquivos e criar versões no repositório local, utilizando o comando de *commit*. Posteriormente,

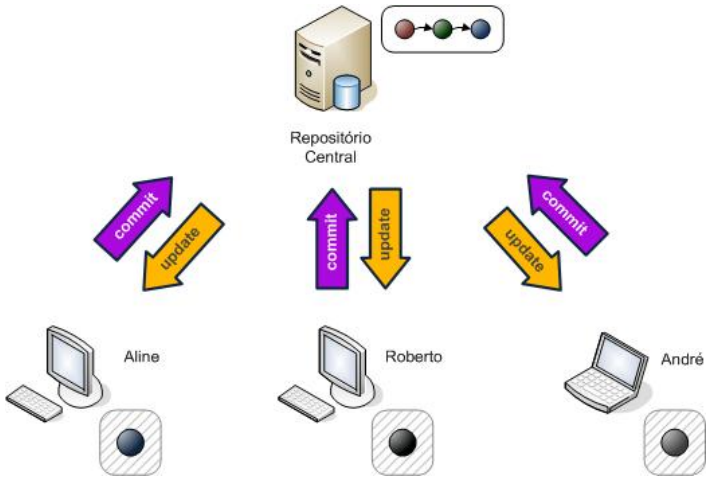

Figura 2.1: Representação do SCV centralizado. Fonte: (DIAS, 2016)

o desenvolvedor pode sincronizar as suas modificações com outros repositórios remotos utilizando os comandos *pull*, que atualiza o repositório local com a versão mais recente do repositório remoto, e *push*, que envia as alterações realizadas localmente para o repositório remoto. Alguns exemplos deste SCV são: Git (CHACON, 2009) e Mercurial (O'SULLIVAN, 2009).

Todo comando *commit* gera uma nova versão que possui informações como: a data de realização do *commit*, o *hash* que é o identificador do *commit*, o autor que o realizou e a mensagem do *commit*. Essas versões são baseadas em *commits* anteriores e criadas em sequência ao longo do tempo, e podem ser organizadas em ordem cronológica. Seguindo a ordenação de tempo, é possível percorrer os *hashes fbd0f95d5*, *f177152c7*, ..., como mostra a Figura 2.3 que é a representação visual reversa do histórico das versões presentes no repositório do projeto *Voldemort*[2]. Na figura é possível observar as primeiras 8 versões em que é a primeira versão com *hash fbd0f95d5* foi executada pelo autor Jay Kreps no dia 02/01/2009 às 21:52 com a mensagem *"Initial import"*.

Após compreender sobre os *commits*, é preciso entender o que são ramos (do inglês *branches*), eles são similares a um ramificação de uma árvore, onde o tronco seria a base do código no repositório representando uma linha independente de desenvolvimento (WOMAKERSCODE, 2021a). O ramo no Git é um ponteiro móvel para um dos *commits*. Então, quando é realizado o *commit*, é obtido o ramo *master* que aponta para o seu último

[2]https://github.com/voldemort/voldemort

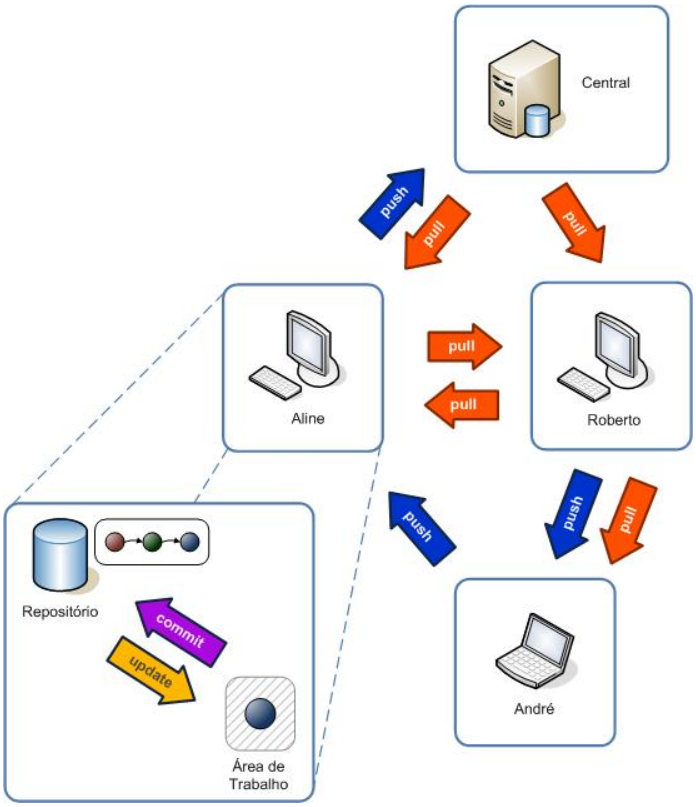

Figura 2.2: Representação do SCV distribuído. Fonte: (DIAS, 2016)

Gráfico	Descrição	Data	Commit	Autor
	Update license and notice files.	13 jan 2009 20:19	e0cff171a	Jay Kreps <jay.kreps@gmail.com>
	Add license file.	13 jan 2009 19:50	573221382	Jay Kreps <jay.kreps@gmail.com>
	Some fixes for read-only store. Some	13 jan 2009 19:48	9e864bc2c	Jay Kreps <jay.kreps@gmail.com>
	read_only store modified.	13 jan 2009 17:36	7d77574c1	Bhupesh Bansal <bbansal.usc@gmail.com>
	Add script to generate partition ids.	12 jan 2009 14:16	b373eaab5	Jay Kreps <jay.kreps@gmail.com>
	Fix serialization bug with writing bool	6 jan 2009 20:53	46a5b4d7e	Jay Kreps <jay.kreps@gmail.com>
	Add package.html files for more pack	6 jan 2009 3:17	f177152c7	Jay Kreps <jay.kreps@gmail.com>
	Initial import	2 jan 2009 21:52	fbd0f95d6	Jay Kreps <jay.kreps@gmail.com>

Figura 2.3: Representação do armazenamento de versões no projeto *Voldemort*.

commit. Cada vez que for executado um *commit*, o ponteiro do ramo principal avançará automaticamente. Vale pontuar que por padrão o ramo principal é denominado *master*. Para exemplificar os ramos, na Figura 2.4 é ilustrado um diagrama que mostra um repositório com duas linhas isoladas de desenvolvimento: *branch 1* e *branch 2*. Ao desenvolver as tarefas em ramificações distintas, há a possibilidade de não apenas trabalhar em ambas

em paralelo, mas também manter a ramificação *master* livre de códigos duvidosos.

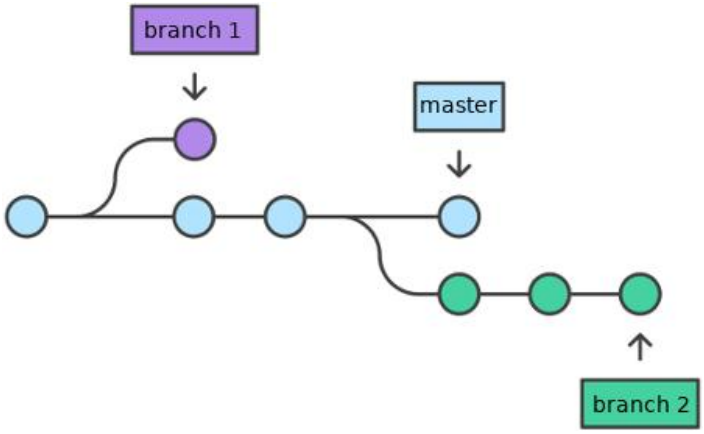

Figura 2.4: Ilustração de *branches*. Fonte: (WOMAKERSCODE, 2021b)

2.1.2 Técnicas de mineração de repositórios

A mineração de repositórios possui diversas aplicações. Por exemplo, Mahmood et al. (2021) propuseram uma abordagem de previsão de falhas para identificar bons preditores para *software* de alta qualidade. Esta proposta aplica a técnica de mineração de regra de associação e analisa estatisticamente um conjunto de quinze projetos de diferentes domínios para indicar se uma entidade de *software* (*i.e,* classe) pode ser classificada como limpa ou com erro. A análise foi realizada a partir da seleção de métricas de *software* que avaliam o projeto de um programa orientado a objetos, como o número de subclasses imediatas de uma classe, o número de classes que são acopladas a uma determinada classe, entre outras. Além disso, é utilizado um banco de dados de *bugs* de projetos do *GitHub* para extrair os conjuntos de dados de métricas. Para identificar as métricas de *software* que podem ser preditores adequados de falhas de *software* e encontrar as melhores regras de associação para previsão de falhas é utilizado o algoritmo *Apriori* que produz regras de associação, encontrando um padrão repetido nos dados. A partir disso, é realizada a análise comparativa e de previsão utilizando projetos industriais que mostra que os resultados são promissores quanto ao uso de regras de associação para predição de falhas durante desenvolvimento inicial de *software*. Por exemplo, com os resultados

desse trabalho, um gerente de qualidade de *software* pode alocar recursos para qualquer módulo específico conforme as regras de associação para economizar tempo e dinheiro. Neste estudo, é aplicada a técnica de mineração de regra de associação O uso de regras selecionadas através da técnica de mineração de regra de associação para classificar classes ajudará a melhorar a qualidade do *software* e reduzir os custos do projeto. A abordagem proposta produz as principais regras de associação para qualquer tipo de campo. Pelo fato de que o preditor selecionado a partir deste método mostra promessa no uso da Curva de Característica de Operação do Receptor (ROC) para prever as classes dos projetos de *software* selecionados, a abordagem proposta é útil para a evolução de *software*. Por exemplo, um gerente de qualidade de *software* pode alocar recursos para qualquer módulo específico de acordo com as melhores regras de associação para economizar tempo e custo. Além disso, os resultados mostraram que são promissores quanto à utilização de regras de associação para previsão de falhas durante desenvolvimento inicial de *software*.

Outro estudo que utiliza MRS foi conduzido por Robles et al. (2006). Este estudo investiga a evolução da distribuição Linux chamada Debian a partir das mudanças relacionadas com as centenas de milhões de linhas de código desenvolvidas ao longo de sete anos. A análise é feita com base no número de pacotes e no número de linhas de código coletados através da extração do arquivo Sources.gz, que contém informações como o nome, a versão e as dependências de construção de cada pacote de versões estáveis do Debian. Concluiu-se que as versões estáveis do Debian dobraram o tamanho do sistema em um curto espaço de tempo e isso pode indicar problemas relevantes para o gerenciamento da evolução do sistema, algo que provavelmente influenciou no atraso dos lançamentos de novas versões estáveis.

No estudo da "Lei de Linus", Raymond e Young (2001) afirmaram que o número de colaboradores é diretamente proporcional à probabilidade de identificar e corrigir um problema em um sistema. Os autores realizaram esta análise em três projetos de *software*: o *kernel* do Linux, a linguagem de programação PHP e o analisador de protocolo de rede *Wireshark* que são de códigos abertos amplamente utilizados em diversos domínios e possuem tamanhos de comunidade de desenvolvedores variados. Através desses projetos foram extraídos os dados de segurança no controle de versão, o que possibilitou a coleta

das métricas de atividade do desenvolvedor, como o número total de arquivos vulneráveis. Desse modo, os autores concluíram que os arquivos alterados por muitos grupos de desenvolvedores eram mais propensos a serem vulneráveis que os alterados por um único grupo (MENEELY; WILLIAMS, 2010).

2.2 Métricas de *software*

Métricas de *software* são utilizadas para quantificar atributos (propriedades ou características) em processos, produtos e projetos de *software* (DASKALANTONAKIS, 1992). As métricas de *software* são importantes por diversos motivos. Por exemplo, Lanza e Marinescu (2006) propuseram uma estratégia de detecção falhas em projetos com regras baseadas em métricas.A seguir são descritas algumas métricas utilizadas na identificação de *bad smell*:

- *Weighted Method per Class* (WMC) é a soma da complexidade ciclomática de McCabe de todos os métodos em uma classe. O valor desse cálculo varia entre o conjunto de números naturais. Por exemplo, considere uma classe A_1 com os métodos M_1, ... M_n que possuem as complexidades respectivamente iguais a c_1, ... c_n. Então, a métrica WMC é extraída conforme a Equação 2.1.

$$WMC = \sum_{i=1}^{n} c_i \tag{2.1}$$

  Por exemplo, considerando o código da classe Funcionário, apresentado na Figura 2.5, é possível concluir que o valor na métrica WMC é igual a quatro. O valor quatro é obtido pois todos os quatro métodos da classe *Funcionário* possuem complexidade igual a um, resultado em um WMC (Funcionário) = 4.

- *Tight Class Cohesion* (TCC) mede a coesão entre os métodos públicos de uma classe. Representa o número relativo de pares de métodos diretamente conectados a atributos na classe. Ou seja, se os métodos acessam um atributo em comum A diretamente. Em outras palavras, se A aparece no corpo dos métodos. O valor dessa

```
1  public class Funcionario {
2      private String nome;
3      private float salario;
4
5      public String getNome() {
6          return nome;
7      }
8
9      public void setNome(String nome) {
10         this.nome =  nome;
11     }
12
13     public float getSalario() {
14         return salario;
15     }
16
17     public void setSalario(float salario) {
18         this.salario = salario;
19     }
20 }
```

Figura 2.5: Exemplo com a métrica WMC(Funcionario) = 4.

métrica varia de 0 a 1 e pode ser obtido através da Equação 2.2.

$$TCC(C) = \frac{NDC(C)}{NP(C)} \tag{2.2}$$

Onde,

- $NP(C)$ representa o número máximo possível de conexões diretas ou indiretas na classe. Se houver um ou mais atributos comuns entre os dois métodos, eles serão conectados diretamente. Se dois métodos estiverem vinculados por outros métodos diretamente conectados, eles serão indiretamente conectados (BIEMAN; KANG, 1995).

Seja N o número de métodos em uma classe.

$$NP(C) = \frac{N * (N-1)}{2} \tag{2.3}$$

- $NDC(C)$ representa o número de conexões diretas.

Por exemplo, o código da Figura 2.6 mostra que classe Produto possui 3 métodos e tem 3 conexões diretas, pois o atributo *custo* aparece no corpo dos métodos *aumentarCusto* e *diminuirCusto* e é transitivamente invocado pelo método *colocaDescon-*

toNoCustoAumentado. Logo, o cálculo do TCC para a classe Produto resulta em 1 como mostra o resultado da equação 2.4.

$$TCC(Produto) = \frac{3}{3 * (3-1)/2} = \frac{3}{3} = 1. \tag{2.4}$$

```
1  public class Produto {
2      private float custo;
3
4      public void aumentarCusto() {
5          custo += 1;
6      }
7
8      public void diminuirCusto() {
9          custo -= 1;
10     }
11
12     public void colocaDescontoNoCustoAumentado() {
13         custo = aumentarCusto() - custo*0.1;
14     }
15 }
```

Figura 2.6: Exemplo com a métrica TCC(Produto) = 1.

- *Access to Foreign Data* (ATFD) conta o número de classes externas cuja uma determinada classe acessa atributos, diretamente ou via métodos de *getters* e *setters*. O valor do cálculo varia entre o conjunto de números naturais. Por exemplo, o código da Figura 2.7 mostra as classes *Pessoa* e *ValidaPessoa*, onde a classe *ValidaPessoa* acessa os atributos da classe *Pessoa* através de métodos *getters*. Logo, o cálculo do ATFD para a classe *ValidaPessoa* resulta em 1.

```
1  public class Pessoa {
2      private String nome;
3      private String cpf;
4
5      public Pessoa() {
6          nome = "";
7          cpf = "";
8      }
9
10     public String getNome() {
11         return nome;
12     }
13
14     public void setNome(String nome) {
15         this.nome =  nome;
16     }
17
18      public String getCpf() {
19         return cpf;
20     }
21
22     public void setCpf(String cpf) {
23         this.cpf =  cpf;
24     }
25 }
26
27 public class ValidaPessoa {
28     public ValidaPessoa(Pessoa pessoa) {
29          if (pessoa.getNome().equals("")) {
30             throw new IllegalArgumentException("Nome invalido");
31         }
32         if (pessoa.getCpf().equals("")) {
33             throw new IllegalArgumentException("CPF invalido");
34         }
35     }
36 }
```

Figura 2.7: Exemplo com a métrica ATFD(Pessoa) = 1.

- *Weight Of Class* (WOC) calcula o número de métodos públicos "funcionais" dividido pelo número total de métodos públicos. Para os métodos públicos "funcionais"são desconsiderados os construtores, *getters* ou *setters* em relação ao total de métodos que a classe possui. O valor do cálculo varia de 0 a 1. Por exemplo, o código da Figura 2.8 mostra a classe Produto que possui 2 métodos públicos "funcionais" *pegarDesconto* e *aumentarDescontoDado* e um total de 7 métodos públicos. Então, o WOC (Produto) = 2/7 = 0,29.

```
1  public class Produto {
2      private float preco;
3      private float desconto;
4
5      public Produto() {
6          preco = 150.0;
7          desconto = 0.2;
8      }
9
10     public float getPreco() {
11         return preco;
12     }
13
14     public void setPreco(float preco){
15         this.preco = preco;
16     }
17
18     public float getDesconto() {
19         return desconto;
20     }
21
22     public void setDesconto(float desconto) {
23         this.desconto = desconto;
24     }
25
26     public float pegardesconto() {
27         return preco * desconto;
28     }
29
30     public float aumentarDescontoDado() {
31         return 0.1 + pegardesconto();
32     }
33 }
```

Figura 2.8: Exemplo com a métrica WOC(Produto) = 0,29.

- *Number Of Public Attributes* (NOPA) calcula o número de atributos públicos de uma classe. Por exemplo, o código da Figura 2.9 mostra a classe Produto que possui os públicos atributos *preço* e *desconto*, ou seja, o NOPA (Produto) = 2.

```
1  public class Produto {
2      public float preco;
3      public float desconto;
4  }
```

Figura 2.9: Exemplo com a métrica NOPA(Produto) = 2.

- *Number Of Access Methods* (NOAM) conta o número de métodos *getters* e *setters* não herdados que uma classe possui. Por exemplo, o código da Figura 2.10 mostra a classe Produto que têm os atributos privados *preço* e *desconto* que possuem seus métodos *getter* e *setter*. Ou seja, o NOAM (Produto) = 4.

```
1  public class Produto {
2      private float preco;
3      private float desconto;
4
5      public float getPreco() {
6          return preco;
7      }
8
9      public void setPreco(float preco){
10         this.preco = preco;
11     }
12
13     public float getDesconto() {
14         return desconto;
15     }
16
17     public void setDesconto(float desconto) {
18         this.desconto = desconto;
19     }
20 }
```

Figura 2.10: Exemplo com a métrica NOAM(Produto) = 4.

- *Number Line of Code* (NLOC) representa o número de linhas do código excluindo linhas em branco e comentários. Por exemplo, o código da Figura 2.11 mostra o método *dividirValores* que possui 6 linhas, ou seja, o NLOC (dividirValores) = 6.

```
1  public float dividirValores(float valor1, float valor2) {
2          if (valor2 == 0) {
3              throw new IllegalArgumentException("Divisao por zero");
4          }
5          else {
6              return valor1/valor2;
7          }
8  }
```

Figura 2.11: Exemplo com a métrica NLOC(dividirValores) = 6.

- *Number Of Parameter* (NOP) conta o número de parâmetros de um método. Por exemplo, o código da Figura 2.12 mostra que o método *localizar* possui 7 parâmetros, ou seja, o NOP (localizar) = 7.

```
1  public void localizar(String pais, String estado, String cidade,
2      String logradouro, String rua, String bairro, String numero) {
3  }
```

Figura 2.12: Exemplo com a métrica NOP(localizar) = 7.

As métricas listadas são usadas como estratégias para identificar quatro dos cinco *bad smells* que serão o foco deste trabalho. Por exemplo, as métricas WOC, NOPA,

NOAM e WMC são utilizadas para detectar o *bad smell Data Class*. Maiores detalhes de como essas métricas podem ser utilizadas para identificar *bad smells* são apresentados na Seção 2.3.

2.3 Bad smell

Segundo Fowler (1999) os *bad smells* são sintomas de escolhas ruins de *design* e implementação que precisam ser removidos do código-fonte por refatoração. Ou seja, a presença de um *bad smell* representa um sintoma de algo errado no projeto ou no código do sistema. Em geral, a ocorrência de *bad smell* mostra que o código deve passar por uma refatoração ou o projeto geral deve ser reexaminado.

Essa seção está organizada com as seguintes subseções: Subseção 2.3.1 define de modo geral os *bad smells* apresentados por Fowler (1999) e da Subseção 2.3.2 até a Subseção 2.3.6 são apresentados como os cinco tipos de *bas smells* tratados nessa abordagem podem ser identificados e tratados.

2.3.1 Visão geral

Nesta subseção é apresentada uma visão geral dos 22 *bad smells* catalogados por Fowler (1999). Para cada *bad smell* são apresentados o nome e a definição como segue:

- *God Class* ou *Large Class*: Ocorre quando uma classe possui muitas responsabilidades. Essas classes geralmente têm muitos atributos e/ou métodos.

- *Data Class*: É identificado quando uma classe contém apenas atributos e seus métodos de acesso (*getters* e *setters*). Nestas classes as suas funcionalidades são tratadas em outras classes, quando poderia ser feito na mesma classe. Ou seja, essas classes são simplesmente contêineres de dados utilizados por outras classes.

- *Duplicated Code*: Como o próprio nome diz, esse *bad smell* ocorre quando o código-fonte possui trechos duplicados. Segundo Fowler (1999), o código duplicado é o pior *bad smell*. Deste modo, o desenvolvedor deve remover o código duplicado sempre que localizar, porque pode tornar o código menos intuitivo e óbvio. Se a mesma estrutura

de código estiver presente em mais de um lugar, certamente o programa será melhor se encontrar uma maneira de unificá-los. Por exemplo, utilizando métodos.

- *Long Method*: É caracterizado pela presença de um método muito longo, resultando em um código difícil de compreender, modificar ou ampliar. Fowler (1999) recomenda fortemente métodos curtos. Quanto mais longo é o método, mais difícil de compreender e dar manutenção.

- *Long Parameter List*: É identificado em métodos que possuem uma lista de parâmetros muito longa. Este tipo de método é difícil de compreender, pois ele possivelmente está fazendo mais do que deveria, gerando alguns problemas de manutenção. Um exemplo de efeito colateral pode ocorrer quando for necessário modificar ou inserir uma nova funcionalidade no método, que pode acarretar na necessidade de adicionar um novo parâmetro, que pode ser opcional.

- *Feature Envy*: Esse *bad smell* ocorre quando um método acessa mais os dados de outra classe do que daquela que ele pertence. Por exemplo, um método *voa* de uma classe *Peixe* que usa apenas atributos da classe *Ave*.

- *Parallel Inheritance Hierarchies*: Esse *bad smell* ocorre quando a criação uma subclasse de uma Classe *A* implica na necessidade de criar uma subclasse de uma Classe *B*.

- *Primitive Obsession*: Ocorre quando os tipos primitivos, como inteiros, números de ponto flutuante e *strings*, são usados no lugar de classes mais apropriadas para o domínio da aplicação. Ou seja, esta má prática ocorre quando tipos primitivos são utilizados para representar uma classe em um domínio.

- *Divergent Change*: Ocorre quando é necessário fazer muitas mudanças em uma classe para introduzir uma nova mudança/recurso. Ou seja, ao realizar alterações na classe, é preciso modificar muitos métodos não relacionados. Por exemplo, ao inserir uma nova categoria de produto, é preciso alterar os métodos de localização, exibição e pedido de produtos.

- *Shotgun Surgery*: Uma classe é afetada por este *bad smell* quando uma mudança nesta classe desencadeia muitas pequenas mudanças em diversas outras classes.

- *Alternative Classes with Different Interfaces*: Expressa o caso em que duas classes realizam a mesma funcionalidade. Contudo, têm nomes distintos para seus métodos.

- *Temporary Field*: Ocorre quando uma classe tem um atributo usado apenas em algumas circunstâncias. Esse código é difícil de compreender, porque normalmente espera-se que um objeto use todos os seus atributos.

- *Data Clumps*: Ocorre quando segmentos de código diferentes contêm o mesmo grupo de variáveis (por exemplo, parâmetros para conexão com o banco de dados). Idealmente, esses aglomerados de dados deveriam ser modelados em uma ou mais classes.

- *Switch Statements*: Ocorre quando a utilização de subclasses e/ou polimorfismo é substituída pelo uso de instruções *switch* como meio de obter comportamento ou dados distintos.

- *Speculative Generality*: Ocorre quando existe uma classe, método, atributo ou parâmetro não utilizado. Por exemplo, quando um código é criado apenas para antecipar o suporte à recursos futuros que podem não ser implementados. O resultado geralmente é um código mais difícil de entender e manter.

- *Message Chains*: Ocorre quando um objeto pede dados a outro objeto, que pede outro objeto, e assim por diante. Esse *bad smell* pode surgir como uma longa linha de métodos *getters*. Por exemplo, suponha que a Classe A que precisa de dados da Classe E. Para recuperar esses dados, o objeto A primeiro precisa acessar o objeto B, depois objeto C, em seguida o objeto D, para assim conseguir acesso ao objeto E. Portanto, tem-se algo como *a.getB().getC().getD().getE().getDado()*.

- *Middle Main*: Uma das principais características dos objetos é o encapsulamento, que geralmente vem com uma delegação. O *Middle Main* ocorre quando uma classe está delegando a maior parte de suas tarefas a outras classes.

- *Lazy Class*: A *Lazy Class* é uma classe que não está fazendo o suficiente. Portanto, deveria ser excluída. Por exemplo, uma classe preparada para apoiar uma funcionalidade futura que nunca será implementada.

- *Refused Bequest*: Ocorre quando uma subclasse não suporta todos os métodos ou dados que herda, isso mostra que a hierarquia provavelmente está errada.

- *Inappropriate Intimacy*: Ocorre quando as classes se tornam muito íntimas e acessam as partes privadas umas das outras. Por exemplo, quando uma classe utiliza os atributos e métodos internos de outra classe.

- *Incomplete Library Class*: Ocorre quando as bibliotecas param de atender às necessidades dos usuários. Uma solução para o problema é alterar a biblioteca, o que geralmente é impossível, pois as bibliotecas são somente de leitura.

- *Comments*: Não são necessariamente *bad smell*. Contudo, costumam ser mal utilizados para compensar códigos mal estruturados.

Nesta subseção foi apresentada uma visão geral dos *bad smells*. O restante dessa seção apresenta detalhadamente como identificar e tratar os *bad smells Long Method*, *God Class*, *Data Class*, *Duplicated Code* e *Long Parameter List*.

2.3.2 Identificação e tratamento de *God Class*

A identificação do *God Class* pode ser realizada de diversas formas. Por exemplo, Lanza e Marinescu (2006) propuseram detectá-lo através do cálculo de três métricas: WMC, TCC e ATFD. De acordo com essa abordagem, uma classe é uma *God Class* quando a Equação 2.5 retorna um valor verdadeiro.

$$WMC \geq 47 \wedge ATFD > 5 \wedge TCC < \frac{1}{3} \tag{2.5}$$

Algumas ferramentas implementam os algoritmos para identificar o *bad smell God Class*. Por exemplo, o *PMD*[3], o *inFusion* (FONTANA et al., 2013) e o *iPlasma* (MARINESCU et al., 2005) adotam a regra da Equação 2.5. Por outro lado, o *JDeodorant*

[3] https://pmd.github.io/

(TSANTALIS; CHAIKALIS; CHATZIGEORGIOU, 2008) identifica *God Class* como uma classe que pode ser dividida em outras classes. Ou seja, quando é identificada a oportunidade de aplicar a refatoração *Extract Class*. Essa análise é realizada através da aplicação de um algoritmo de *clustering* para detectar grupos coesos de atributos ou métodos que podem ser extraídos como classes separadas.

Uma vez identificado, o *bad smell* pode ser tratado. Por exemplo, o código da Figura 2.13 possui *God Class*, pois mostra uma classe que faz a função de duas classes: *Cliente* e *Endereço*. O tratamento consiste na divisão dessa classe em duas novas classes: *Cliente*, que permanece com os atributos e métodos relacionados ao cliente; e *Endereço*, que recebe os atributos e métodos relacionados ao endereço. A Figura 2.14 mostra como a nova classe foi criada.

```
1  @Getter
2  @Setter
3  @AllArgsConstructor
4  public class Cliente {
5      private Long id;
6      private String nomeCompleto;
7      private String cpf;
8      private String telefone;
9      private String bairro;
10     private String numero;
11     private String cidade;
12     private String estado;
13     private String cep;
14     private String pais;
15     //....
16 }
```

Figura 2.13: Código sem o tratamento do *God Class*. Fonte: (SINGH, 2020)

2.3.3 Identificação e Tratamento do *Data Class*

A detecção do *Data Class* pode ser realizada de diversas maneiras. Por exemplo, Lanza e Marinescu (2006) propuseram a identificação de *Data Class* via o cálculo das seguintes métricas: WOC, NOPA, NOAM e WMC. Tais métricas são utilizadas na identificação conforme descrito na Equação 2.6.

```
1  @Getter
2  @Setter
3  @AllArgsConstructor
4  public class Cliente {
5      private Long id;
6      private String nomeCompleto;
7      private String cpf;
8      private String telefone;
9      private Endereco endereco;
10     //....
11 }
12
13 @Getter
14 @Setter
15 @AllArgsConstructor
16 public class Endereco {
17     private Long id;
18     private String bairro;
19     private String numero;
20     private String cidade;
21     private String estado;
22     private String cep;
23     private String pais;
24     //....
25 }
```

Figura 2.14: Código com o tratamento do *God Class*. Fonte: (SINGH, 2020)

$$(WOC < \frac{1}{3}) \wedge (((NOPA + NOAM > 4) \wedge (WMC < 47)) \\ \vee((NOPA + NOAM > 2) \wedge (WMC < 31))) \tag{2.6}$$

Por exemplo, a classe *DataClass* da Figura 2.15 é suspeita de ser uma *Data Class* por ser observado que as métricas WOC, NOPA, NOAM e WMC possuem os valores 0, 3, 1 e 1 respectivamente. O valor de WOC (DataClass) é igual a 0, pois a classe não possui nenhum método público "funcional"; o valor de NOPA (DataClass) é igual a 3, pois a classe possui 3 atributos públicos *name*, *bar* e *na*; o NOAM (DataClass) é igual a 1, pois a classe contém apenas um método *setter*; e o WMC (DataClass) = 1 pois a classe possui apenas um método que tem complexidade igual a 1. Para comprovar o resultado, a seguir é feito o passo a passo da resolução da expressão da Equação 2.6 utilizando os valores citados anteriormente.

Substituindo WOC por 0, NOPA por 3, NOAM por 1 e WMC por 1 na Equação 2.6:

$$(0 < \frac{1}{3}) \wedge (((4 > 4) \wedge (1 < 47)) \vee ((4 > 2) \wedge (1 < 31))) \tag{2.7}$$

Resultando:

$$V \wedge ((F \wedge V) \vee (V \wedge V)) \tag{2.8}$$

$$V \wedge (F \vee V) \tag{2.9}$$

$$V \wedge V \tag{2.10}$$

$$V \tag{2.11}$$

Após resolver a expressão da Equação 2.6, a expressão deu verdadeiro o que indica que a classe *DataClass* é realmente suspeita de ser um *Data Class*.

Algumas ferramentas que identificam esse *bad smell* são: *iPlasma* (MARINESCU et al., 2005), *inFusion* (FONTANA et al., 2013), *JSpIRIT* (VIDAL et al., 2015) e *PMD*. Estas quatro ferramentas utilizam a regra da Equação 2.6 para realizar suas identificações.

```
1  public class DataClass {
2
3      // classe expoe atributos publicos
4      public String name = "";
5      public int bar = 0;
6      public int na = 0;
7
8      private int bee = 0;
9
10     // e privados por meio de getters
11     public void setBee(int n) {
12         bee = n;
13     }
14 }
```

Figura 2.15: Código sem o tratamento do *Data Class*. Fonte: (PMD, 2021a)

Na Figura 2.16 mostra um tratamento para o código da Figura 2.15 que utiliza a refatoração *Encapsulate Field* (FOWLER, 1999) para ocultar o acesso direto aos atributos *name*, *bar* e *na*, e exigir que o acesso seja somente por *getters* e *setters*.

```
1  public class DataClass {
2      private String name = "";
3      private int bar = 0;
4      private int na = 0;
5      private int bee = 0;
6
7      public String getName() {
8          return name;
9      }
10     public void setName(String name) {
11         this.name = name;
12     }
13     public int getBar() {
14         return bar;
15     }
16     public void setBar(int bar) {
17         this.bar = bar;
18     }
19     public int getNa() {
20         return na;
21     }
22     public void setNa(int na) {
23         this.na = na;
24     }
25     public int getBee() {
26         return bee;
27     }
28     public void setBee(int bee) {
29         this.bee = bee;
30     }
31 }
```

Figura 2.16: Código com o tratamento do *Data Class*.

2.3.4 Identificação e Tratamento do *Long Method*

A identificação do *Long Method* é normalmente baseada no número de linhas do método. Por exemplo, ferramentas como *Checkstyle*[4] e *PMD* detectam este *bad smell* considerando a métrica NLOC do método. No *PMD* o limite é de 100 linhas, enquanto no *Checkstyle*[4] o limite é 150 linhas. Entretanto, o *JDeodorant* (TSANTALIS; CHAIKALIS; CHATZIGEORGIOU, 2008) usa técnicas de fatiamento para determinar se uma classe é elegível para uma refatoração de *Extract Method*.

Uma das maneiras de tratar este *bad smell* é removendo qualquer código duplicado. Por exemplo, utilizando o *PMD*, o código da Figura 2.17 possui *Long Method*, pois o método *doSomething* possui 100 linhas. Então, um dos tratamentos para esse problema foi inserir uma estrutura de repetição para manter o comportamento de imprimir as 100

[4]https://checkstyle.sourceforge.io/

vezes da frase *"Hello World!"* como mostra a Figura 2.18.

```
1 public void doSomething() {
2     System.out.println("Hello World!");
3     System.out.println("Hello World!");
4     // 98 copias de println omitidas por questoes de brevidade.
5 }
```

Figura 2.17: Código sem o tratamento do *Long Method*. Fonte: (PMD, 2021b)

```
1 public void doSomething() {
2     for(int i = 0; i < 100; i++) {
3         System.out.println("Hello World!");
4     }
5 }
```

Figura 2.18: Código com o tratamento do *Long Method*.

2.3.5 Identificação e Tratamento do *Long Parameter List*

A detecção do *Long Parameter List* é realizada a partir da contagem da quantidade de parâmetros do método (SINGH; BINDAL; KUMAR, 2020). O ponto crítico na identificação deste *bad smell* é a configuração do valor limite. Por exemplo, no *PMD*, o valor padrão do limite é 10, enquanto em *Checkstyle*[4] é 7. Ou seja, não há uma estratégia única para definir a quantidade limite de parâmetros.

Uma das formas de tratamento desse *bad smell* é substituir um grupo de variáveis passados como parâmetros por objetos. Por exemplo, o código da Figura 2.19 mostra um método com 6 parâmetros, onde 3 deles podem ser agrupados no objeto data de nascimento e 2 deles como objeto de medidas. Então, o tratamento utilizado nesse caso foi o uso da refatoração *Introduce Parameter Object* (FOWLER, 1999) que agrupa vários parâmetros em novos objetos como é mostrado na Figura 2.20.

```
1 public void addPessoa( // muitos argumentos susceptiveis
2 //de serem confundidos
3     int anoAniversario, int mesAniversario, int diaAniversario,
4         int altura, int peso, int snn) {
5     ....
6 }
```

Figura 2.19: Código sem o tratamento do *Long Parameter List*. Fonte: (PMD, 2021c)

```
1 public void addPessoa( // abordagem preferida
2     Date aniversario, MedidasCorporais medidas, int snn) {
3     ....
4 }
```

Figura 2.20: Código com o tratamento do *Long Parameter List*. Fonte: (PMD, 2021c)

2.3.6 Identificação e Tratamento do *Duplicated Code*

A identificação do *Duplicated Code* ocorre quando a mesma estrutura de código é duplicada em mais de um local dentro de um sistema de *software* (FOWLER, 1999). Por exemplo, esse *bad smell* acontece quando um mesmo código é localizado em dois ou mais métodos de uma mesma classe.

A estratégia de detecção pode variar de acordo com abordagem ou ferramenta utilizada. Por exemplo, o *Checkstyle*[4] identifica esse *bad smell* quando o programa apresenta pelo menos 12 linhas de código duplicado consecutivo em mais de um local. Vale ressaltar que o código duplicado pode abranger vários métodos ou classes. Por outro lado, o *PMD* analisa o código duplicado pelo número mínimo de *tokens* informado para determinar a ocorrência deste *bad smell*. Vale ressaltar que esse número de *token* é configurável pela ferramenta.

Um exemplo de *Duplicated Code* ocorre quando o mesmo fragmento de código está presenta em subclasses de mesmo nível (FOWLER, 1999). Considerando que os fragmentos tratam um campo comum em todas as subclasses, o tratamento recomendado é o uso da refatoração *Pull Up Field* (FOWLER, 1999) que remove o atributo das subclasses e movê-lo para a superclasse. Para exemplificar, na Figura 2.21 mostra o código duplicado dos atributos *nome* e *salario* no relacionamento de herança. Deste modo, para tratá-lo, os dois atributos duplicados são removidos das subclasses e movido para superclasse *Funcionario* como mostra a Figura 2.22.

2.3.7 Ferramentas de detecção de *bad smells*

Para melhorar a qualidade do código no processo de desenvolvimento de *software*, diversas ferramentas foram construídas. Entre essas ferramentas, existem as que podem detectar *bad smells* como os catalogados por Fowler (1999). Por muitos motivos, ferramentas

```
1  public class Funcionario {
2
3  }
4
5  public class Vendedor extends Funcionario {
6      private String nome;
7      private Double salario;
8  }
9
10 public class Engenheiro extends Funcionario {
11     private String nome;
12     private Double salario;
13 }
```

Figura 2.21: Código sem o tratamento do *Duplicated Code*.

```
1  public class Funcionario {
2      private String nome;
3      private Double salario;
4  }
5
6  public class Vendedor extends Funcionario {
7
8  }
9
10 public class Engenheiro extends Funcionario {
11
12 }
```

Figura 2.22: Código com o tratamento do *Duplicated Code*.

encontram resultados diferentes ao analisar o mesmo sistema. Conforme discutido anteriormente, as ferramentas podem utilizar formas diferentes para identificar os *bad smells* em suas implementações.

A Tabela 2.1 apresenta a relação dos cinco tipos de *bad smells* citados anteriormente com as ferramentas que identificam em programas escritos em Java. Na coluna ferramentas da tabela são apresentadas os nomes das 9 ferramentas encontradas, sendo elas: *PMD, JDeodorant, JSpirit, Checkstyle, inFusion, iPlasma, Stench Blossom, True-Refactor e inCode*. Nas demais colunas são apresentados cada um dos cinco tipos de *bad smell*: *God Class, Data Class, Long Method, Long Parameter List* e *Duplicated Code*. Vale pontuar que as células marcadas com X representam que um determinado tipo de *bad smell* é detectado pela determinada ferramenta. Por exemplo, o *PMD* identifica o *Duplicated Code*.

O *PMD* é uma ferramenta que verifica o código-fonte e procura problemas potenciais ou possíveis *bugs*, como código morto, variáveis ou parâmetros locais não utilizados

Ferramentas	Tipos de *bad smell*				
	God Class	*Data Class*	*Long Method*	*Long Parameter List*	*Duplicated Code*
JDeodorant	X		X		X
PMD	X	X	X	X	X
Checkstyle	X		X	X	X
JSpIRIT	X	X	X		
inFusion	X	X	X		X
iPlasma	X	X	X		X
Stench Blossom	X		X		
TrueRefactor			X		
InCode		X			X

Tabela 2.1: Relação entre os *bad smells* e as ferramentas que os detectam.

e código duplicado. Além disso, permite que o usuário defina os valores de limite para as métricas exploradas. A realização da detecção de más práticas através de métricas é independente de IDE. Porém, pode ser utilizada com outras ferramentas como o *Eclipse Plugin*, está disponível para *download*, que é ativamente desenvolvida e mantida.

A ferramenta *JDeodorant* auxilia os usuários a determinar a refatoração apropriada para os *bad smells* encontrados na aplicação, identificando possíveis transformações de refatoração que resolvem os problemas identificados. Além disso, classifica os *bad smells* de acordo com seu impacto de *design*, apresentando-os ao desenvolvedor e aplicando automaticamente a refatoração de *bad smell* escolhida pelo desenvolvedor. Entretanto, é um *Eclipse Plugin* que está disponível para *download*, porém é exclusiva da IDE.

O *Checkstyle* é uma ferramenta de desenvolvimento para ajudar os desenvolvedores a escrever código Java que siga um padrão de codificação, além de ser altamente configurável e possibilitar suporte para quase todos os padrões de codificação. Entretanto, é um *Eclipse Plugin* que está disponível para *download*, mas é limitada pela IDE.

O *inFusion* é a evolução do *iPlasma* que é uma plataforma integrada para avaliação da qualidade de sistemas orientados a objetos que inclui suporte para todas as fases de análise necessárias, desde a extração do modelo até a análise baseada em métricas de alto nível. Entretanto, não oferece uma maneira acessível e fácil de exportar seus resultados, e não foi encontrada disponível para *download*.

O *Decor* é uma ferramenta que não oferece uma maneira acessível e fácil de

exportar seus resultados e não foi encontrada disponível para *download*.

O *Stench Blossom* é um *plugin* para o ambiente Eclipse que fornece ao desenvolvedor três visualizações diferentes, que progressivamente oferece mais informações sobre os *bad smells* que está sendo visualizado. Entretanto, não oferece uma maneira acessível e fácil de exportar seus resultados e não foi encontrada disponível para *download*.

A ferramenta *InCode* é um *plugin* Eclipse semelhante ao *inFusion* que fornece detecção de problemas de *design* conforme o código é escrito. Por fim, a *TrueRefactor* é uma ferramenta de refatoração automatizada que melhora significativamente a compreensão dos sistemas legados. Ambas ferramentas não foram encontrados disponíveis para *download*.

Após investigar essas 9 ferramentas, foi concluído que apenas as ferramentas *PMD, Jdeodorant, JSpirit* e *Checkstyle* teriam disponibilidade para *download*. Dentre as disponíveis, a *PMD* apresentou mais pontos positivos com respeito à usabilidade, facilidade de exportação dos resultados, não é necessário compilar o código-fonte para usar e é independente de IDE.

2.4 Integração contínua

A Integração Contínua (IC) é uma prática utilizada no desenvolvimento de *software* onde os membros de uma equipe integram continuamente seus trabalhos. A cada integração é realizada uma verificação através de um processo de construção que contém tarefas como compilação, testes e execução de *scripts*. Um dos principais objetivos da IC é detectar erros o mais rápido possível para que os desenvolvedores possam trabalhar na sua solução (FOWLER, 1999).

A IC pode ser utilizada de maneira manual ou automática. Na forma manual, a tarefa é serializada e, dificilmente, pode ser escalável conforme a equipe aumenta (DUVALL; MATYAS; GLOVER, 2007). Por outro lado, na forma automatizada, as integrações podem se tornar mais facilmente escaláveis, em função desse procedimento ser apoiado pelos Servidores de Integração Contínua (SIC). Esses servidores são responsáveis por monitorar os artefatos quando são encaminhados para repositórios de *software*, essa análise é realizada através da execução do *script* de *build* que compila o código-fonte, in-

tegra as bases de dados, executa inspeções, executa testes e implanta *software* (DUVALL; MATYAS; GLOVER, 2007).

Diversos SIC estão disponíveis para que equipes de desenvolvimento consigam aplicar IC de maneira automatizada. Alguns dos servidores de IC são o Jenkins[5], Travis CI[6] e o GitlabCI[7]. O Jenkins[5] é uma ferramenta de código aberto que requer uma configuração elaborada. Porém, fornece uma variedade de opções de personalização e de configurações completas. O Travis CI[6] é uma ferramenta comercial de CI que oferece menos opções de personalização e possui um arquivo de configuração. Enquanto, o GitlabCI[7] é uma ferramenta desenvolvida pelo GitLab que usa uma configuração de arquivo relativamente leve, ela só faz sentido para projetos que utilizam outras ferramentas do GitLab.

Na Figura 2.23 é possível visualizar a lista do histórico de *build* do Travis CI. Em especial, os dois últimos lançamentos terminaram com erros na implantação causados pela falha no momento de realizar a execução do *openssl* para criptografar a chave.

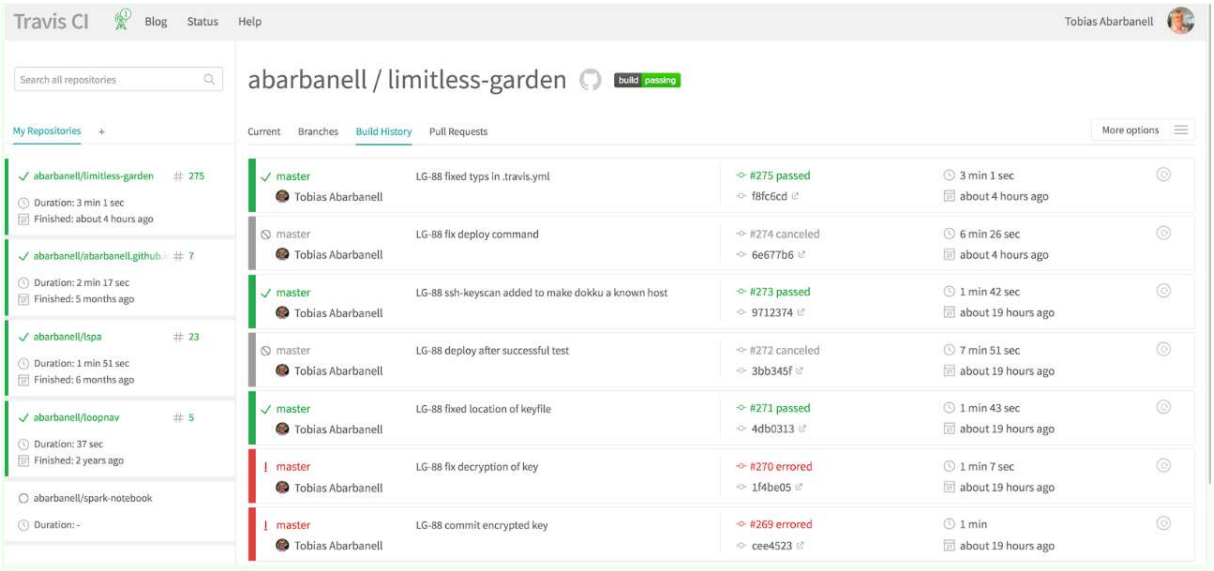

Figura 2.23: Implantação do Travis CI para o projeto Dokku. Fonte: (ABARBANELL, 2017)

Vale ressaltar que Fowler (1999) fornece um conjunto de 10 boas práticas que ajudam a IC a alcançar melhores resultados e diminuir as falhas. As práticas são listadas

[5]https://www.jenkins.io/
[6]https://travis-ci.org/
[7]https://docs.gitlab.com/ee/ci/

a seguir:

1. **O projeto de *software* deve ser mantido em um repositório de referência única.** Devido ao grande esforço de manter o controle de todos os arquivos, principalmente quando há diversas pessoas envolvidas.

2. **Automatizar a construção**. Esse processo proporciona a possibilidade de economia do tempo via a automatização.

3. **Tornar a construção autotestável**. Uma boa forma de identificar erros de forma rápida e eficiente é adicionando testes automatizados ao processo de construção.

4. **Cada desenvolvedor integra suas mudanças pelo menos uma vez ao dia**. A IC tem como uma de suas vantagens a comunicação entre os integrantes da equipe, compartilhando as mudanças com outros membros da equipe e recebendo *feedback*.

5. **Cada *commit* no repositório principal deve ser construído em uma máquina de integração**. Para que o repositório permaneça em uma condição saudável, deve ser garantido que as construções frequentes aconteçam em uma máquina de integração e apenas se essa construção de integração for bem-sucedida, o *commit* deve ser considerado finalizado.

6. **Manter o tempo de execução da *build* rápido**. Na IC, o *feedback* rápido é a meta para que o responsável pela mudança seja notificado e providencie as alterações necessárias.

7. **O teste deve ser realizado em uma cópia do ambiente de produção**. O objetivo dessa prática é identificar e tratar, em condições controladas, qualquer problema que o sistema possa ter na produção.

8. **O último executável do projeto deve ser armazenado em uma área de fácil acesso para que qualquer um consiga usar**. Esse método é útil para os desenvolvedores checarem se o *software* está correto, a partir de demonstrações, visualização das últimas mudanças realizadas ou até realizando testes exploratórios.

9. **Todos os desenvolvedores devem ter fácil acesso ao estado do projeto e as suas mudanças**, possibilitando o *feedback* rápido, um dos principais propósitos da IC.

10. **Automatizar a implantação**. A IC possibilita ter diversos ambientes, seja para produção, desenvolvimento, testes e outros. Portanto, é importante ter *scripts* que permitem implantar aplicações facilmente em qualquer ambiente.

Com a utilização das práticas citadas anteriormente, a IC se torna mais eficaz. Dessa forma, a IC pode ser utilizada como um aliado na identificação e tratamento de *bad smells*, além de outras tarefas do ciclo de desenvolvimento de um *software*. Por exemplo, uma possível solução é utilizar o servidor de IC para capturar *bad smells* através da execução de *scripts* que utilizam a ferramenta *PMD*. Caso um *bad smells* seja identificado, o desenvolvedor receberá uma notificação e poderá iniciar o tratamento.

2.5 Trabalhos relacionados

Para encontrar trabalhos relacionados com *bad smells* e IC, foi realizada uma busca utilizando alguns passos do Mapeamento Sistemático da Literatura (BARRETO; MURTA; ROCHA, 2012).

De acordo com Petersen et al. (2008), o processo de Mapeamento Sistemático da Literatura é composto por etapas como: a definição das questões de pesquisa, que define o escopo da pesquisa; a realização da coleta dos estudos primários através de uma estratégia e fontes de busca, que deve ser capaz de apresentar todos os artigos que estejam disponíveis no escopo; a seleção dos trabalhos relevantes através de critérios de inclusão e exclusão dos artigos; a extração de dados e os resultados. Desse modo, o mapeamento sistemático oferece uma visão geral mais ampla de determinada área, além de proporcionar o conhecimento sobre as frequências de publicações ao longo do tempo, quantidade e os tipos de pesquisa dentro dela, possibilitando indicar tendências.

Para coletar estudos relacionados ao tema deste trabalho foi criada uma *string* de busca que relaciona os termos "identificação de *bad smells*"e "Integração Contínua". A elaboração da *string* de busca foi baseada em palavras-chave relacionadas ao tema alvo,

visando eliminar o máximo possível de resultados não desejados. A *string* é composta pelos parâmetros: população, que está relacionado à identificação de *bad smells*; intervenção, as técnicas de IC; e saída, as técnicas, abordagens, métodos, metodologias, ferramentas ou processos de identificação de *bad smells* que utilizam IC. Após alguns testes para calibrar as expressões dos parâmetros, foi obtida uma *string* da seguinte forma:

População: (("bad smell" OR "code smell" OR "bad code smells") AND ("identification" OR "detection"))

Intervenção: "Continuous Integration"

Saída: (characterization OR approach OR method OR methodology OR procedure OR definition OR mechanism OR experience OR findings OR research OR study OR technique OR knowledge OR tool OR support)

(("bad smell" OR "code smell" OR "bad code smells") AND ("identification" OR "detection")) AND "Continuous Integration" AND (characterization OR approach OR method OR methodology OR procedure OR definition OR mechanism OR experience OR findings OR research OR study OR technique OR knowledge OR tool OR support) AND (LIMIT-TO (SUBJAREA , "COMP") OR LIMIT-TO (SUBJAREA , "ENGI"))

Deste modo, a *string* de busca mais satisfatória foi definida utilizando a filtros para restringir a área de computação e engenharia. Vale ressaltar que a sintaxe da *string* está conforme a base de dados bibliográfica *Scopus*[8]. Essa biblioteca foi escolhida devido ao fácil acesso à busca de referências, bem como à facilidade de busca de texto completo dos artigos. Além de ser considerada significativa, pois oferece publicações relevantes que podem contribuir para os resultados da pesquisa e ser considerada uma das bibliotecas mais relevantes para a Engenharia de *Software* de acordo com o Keele et al. (2007). Finalmente, a busca na base *Scopus* quando utilizada de forma isolada, retorna cerca de 75% dos resultados em comparação as bases *Compendex* e *IeeeXplore* que retornam cerca de 65% e 42% dos resultados, respectivamente. Tendo assim, o maior percentual de cobertura de publicações dentre essas bibliotecas digitais (BARRETO; MURTA; ROCHA, 2012).

Na primeira etapa, a *string* de busca foi aplicada na pesquisa avançada do *Scopus*,

[8]https://www.scopus.com/home.uri

retornando um total de 57 artigos. Na segunda etapa, foi feita a leitura do resumo, introdução e conclusão de cada um desses artigos. Os artigos foram classificados utilizando os seguintes critérios de exclusão: artigo não está relacionado a IC e artigo não relacionado a *bad smells*. Deste modo, é verificado se o artigo relaciona a detecção de *bad smells* com a IC. Como resultado, foram excluídos: 3 artigos por serem sobre *security smell*, 9 focados em *test smell*, 37 sobre IC sem ligação com *bad smell*, totalizando em 49 artigos excluídos. Por fim, foram constatados que os 8 restantes podem contribuir com este trabalho sendo que 3 deles são sobre *bad smells*, 1 sobre anti-padrões considerados sinônimos de *bad smells* por alguns autores como Linares-Vásquez et al. (2014) e Palomba et al. (2015), 1 sobre dívida técnica que pelo estudo proposto por Tufano et al. (2017) cita que *bad smells* são um dos principais contribuintes para a dívida técnica e pode afetar a capacidade de manutenção de um projeto, e 3 poderiam ajudar a entender as práticas com a IC. Contudo, nenhum deles trata sobre a influência da adoção de IC no ciclo de vida dos *bad smells*. A Figura 2.24 apresenta os resultados decorrentes dessa seleção de artigos.

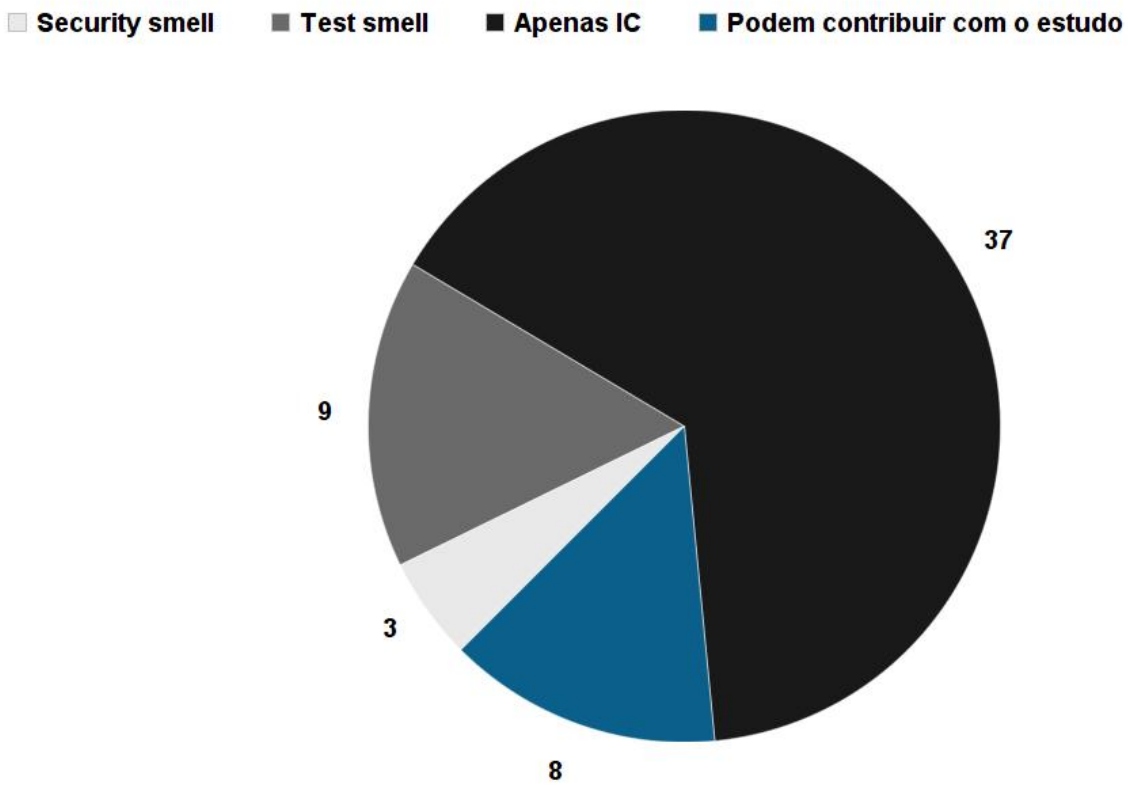

Figura 2.24: Representação da seleção dos artigos.

Como não houve retorno de nenhum artigo que relacionasse a identificação de *bad smells* com a IC a partir da pesquisa com a *Scopus*, foi realizada uma revisão na literatura em busca de pesquisas sobre identificação de *bad smells*. Para tanto, foi feita uma iteração de *Snowballing* no Google Scholar[9] à procura de todos que citavam o artigo "*Evaluating the lifespan of code smells using software repository mining*" e pela restrição de tempo foi realizado apenas uma iteração e não foi procurado para os próximos. Essa busca resultou na filtragem de 123 artigos relacionados com detecção de *bad smells*, em seguida, foi lido o título e resumo de cada um para selecionar os artigos que detectam os *bad smells* por meio da utilização de MRS.

A estratégia de busca por pesquisas referentes a identificação de *bad smells* foi adotada. Dentre elas, estão Peters e Zaidman (2012), Palomba et al. (2014) e Tufano et al. (2015) conduziram seus estudos baseados na mineração de repositórios de *software* para identificar os *bad smells* definidos por Fowler (1999). Em Peters e Zaidman (2012), os autores apresentam uma abordagem que utiliza as ferramentas de detecção de *bad smells JDeodorant* (TSANTALIS; CHAIKALIS; CHATZIGEORGIOU, 2008) e *Ptidej*[10] para determinar o tempo de vida dos *bad smells*: *God Class, Feature Envy, Data Class, Message Chain Class* e *Long Parameter List* em sete projetos de *software* escritos na linguagem Java. Este estudo mostra que os desenvolvedores estão cientes da presença dos *bad smells*, porém não os consideram importantes para efetuar os seus tratamentos.

Palomba et al. (2014) realizaram dois estudos empíricos para avaliar a abordagem proposta e verificar a eficiência na identificação dos *bad smells Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob* e *Feature Envy*. A análise foi realizada em vinte sistemas de *software* comparando com abordagem Probabilidade de Propagação de Alterações de Projeto proposta por Rao e Reddy (2007). Os autores concluíram que a abordagem proposta tem melhor precisão e *recall* comparado com a outra abordagem, tendo uma vantagem adicional por destacar os *bad smells* que podem tornar o código mais complexo, aumentando a propensão a mudanças e falhas.

Tufano et al. (2015) propuseram outra abordagem para compreender quando e porque os *bad smells* são inseridos em projetos de *software*. Eles mostraram evidências

[9] https://scholar.google.com.br/
[10] http://www.ptidej.net/

que os *bad smells*, *Duplicated Code, Shotgun Surgery* e *Divergent Change* são introduzidos desde a criação de métodos e classes, indicando que isso poderia ser evitado se houvesse uma verificação contínua da qualidade em cada *commit* realizado.

Chatzigeorgiou e Manakos (2010) estudaram a evolução dos *bad smells Long Method, Feature Envy* e *Switch Statements*. Eles também concluíram que a maioria dos *bad smells* são inseridos no momento em que o método onde residem é adicionado pela primeira vez no sistema de *software* e persistem no sistema até sua última versão, devido a sua não remoção do projeto de *software*.

2.6 Considerações finais

Nesse capítulo, foram apresentados os conceitos de mineração de repositório de *software*, dentre eles, o SCV e as técnicas de MRS. Além disso, foi introduzida uma visão geral dos *bad smells* e as métricas utilizadas na identificação de *bad smells* que serão estudados nesse trabalho. Em seguida, foi definido o conceito de IC, mostrando as suas formas de utilização e as 10 práticas que auxiliam a atingir resultados melhores. Para complementar, é apresentado o processo de busca por trabalhos relacionados que mostraram que não houve nenhum artigo que relacionasse *bad smells* com IC. Os conceitos apresentados neste capítulo foram utilizados neste trabalho para compreender o ciclo de vida dos *bad smells* e avaliar se a integração contínua auxilia na detecção e tratamento dos *bad smells*.

3 Estudo do ciclo de vida dos *bad smells*

Este capítulo descreve a abordagem desenvolvida para estudar o ciclo de vida dos *bad smells* em projetos de desenvolvimento de *software*. Para tanto, ele possui a seguinte divisão: a Seção 3.1 descreve a abordagem utilizada e mostra como é realizada a extração dos dados de *bad smells* do histórico dos projeto e a Seção 3.2 apresenta a implementação da prova de conceito.

3.1 Definição da abordagem

A abordagem proposta visa criar um mecanismo de extração automatizada para identificar o ciclo de vida de um *bad smell*, considerando a sua inserção e o eventual tratamento. Essa análise é baseada no histórico do projeto onde cada versão é visitada, uma a uma, para identificar a presença dos *bad smells* seguintes: *God Class, Data Class, Long Method, Long Parameter List* e *Duplicated Code*.

A Figura 3.1 ilustra as etapas do processo de extração dos dados. Na etapa 1, a abordagem recebe como entrada um repositório de *software*. Na etapa 2, as versões presentes no repositório são analisadas seguindo a ordem cronológica entre as versões. Na etapa 3, são aplicados os filtros para identificação da presença de *bad smells*, ou seja, são utilizados detectores de *bad smell*, representados pelas lupas, para localizar os arquivos que possuem *bad smells*. Na etapa 4, é gerado o resultado da análise, versão por versão, apontando a presença dos *bad smells* identificados em cada *commit*.

Considerando o cenário da Figura 3.1 e que as lupas identificam os *bad smells God Class, Data Class, Long Method, Long Parameter List* e *Duplicated Code*, da esquerda para a direita, é possível observar que:

- na primeira versão não houve detecção de *bad smells*, pois todas as lupas estão na cor verde;
- na segunda versão, foi identificado o *bad smell Data Class* pois apenas a segunda

lupa está vermelha;

- na terceira versão, o *Data Class* permaneceu no projeto e o *Long Method* foi inserido pois a segunda e terceira lupa estão vermelhas; e
- na quarta versão, o *Data Class* foi tratado, o *Long Method* permaneceu no projeto e o *Long Parameter List* foi identificado pois a segunda lupa se tornou cor verde, a terceira e quarta lupa estão vermelhas.

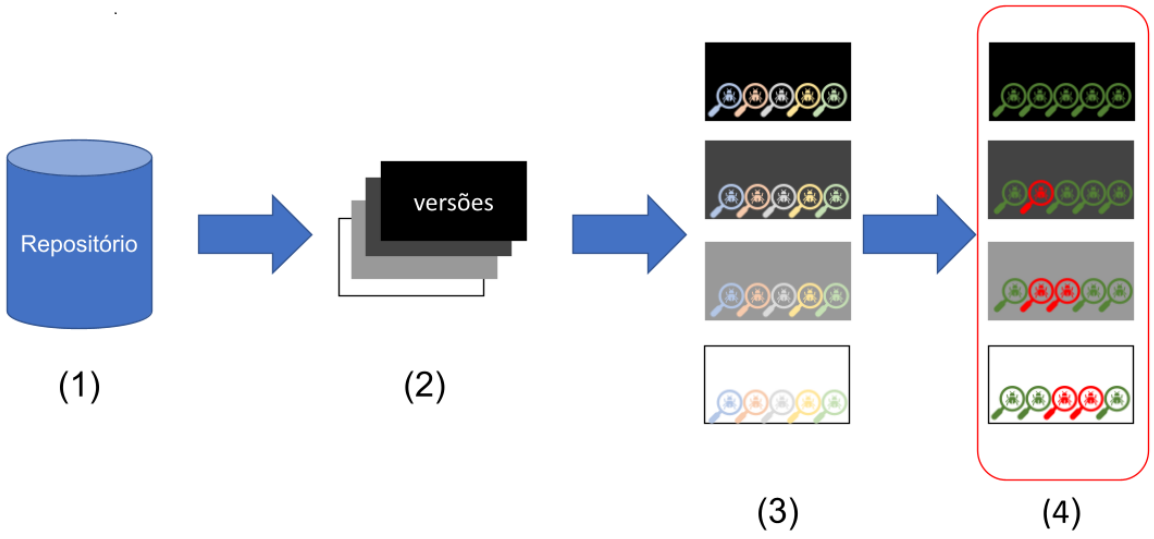

Figura 3.1: Ilustração da abordagem.

Como citado anteriormente, a etapa 3 da abordagem utiliza detectores de *bad smell* para identificá-los em uma versão do projeto. A identificação dos *bad smells God Class* e *Data Class* pode ser realizada utilizando uma das estratégias descritas nas seções 2.3.2 e 2.3.3, respectivamente. Como esses *bad smells* estão relacionados a classe, o identificador utilizado é o nome da completo da classe em que ele foi identificado. Na linguagem Java o nome completo de uma classe é obtido através da concatenação do nome do pacote com o nome da classe. Por exemplo, no projeto *Voldemort* na versão com *sha a7dbdea* foi identificado um *Data Class* na classe *RESTClientConfig*, que pertencente ao pacote *voldemort.restclient*, cujo nome completo é *voldemort.restclient.RESTClientConfig*.

A identificação do *Long Method* e *Long Parameter List* pode ser realizada utilizando uma das estratégias apresentadas nas seções 2.3.4 e 2.3.5, respectivamente. Esses

tipos de *bad smells* estão relacionados a um método presente em uma classe. Consequentemente, o identificador é composto pelo nome completo da classe e nome do método onde um desses tipos de *bad smells* foi identificado. Por exemplo, no projeto *Voldemort* na versão com *sha a7dbdea* foi identificado um *Long Method* no método *fetchFromSource* que está na classe cujo nome completo é *voldemort.store.readonly.fetcher.HdfsFetcher*.

Finalmente, a identificação do *Duplicated Code* é realizada conforme foi descrito na Seção 2.3.6. O identificador utilizado para esse *bad smell* é composto pelo caminho do arquivo com a região de código duplicado e o fragmento duplicado. Por exemplo, no projeto *Voldemort* na versão com *sha a7dbdea* foi identificado um *Duplicated Code* no caminho "/voldemort/src/java/voldemort/client/protocol/pb/VProto.java"onde o fragmento duplicado é o mostrado no código da Figura 3.2.

```
1       private boolean hasError;
2
3       private voldemort.client.protocol.pb.VProto.Error error_;
4       public boolean hasError() {
5           return hasError;
6       }
7
8       public voldemort.client.protocol.pb.VProto.Error getError() {
9           return error_;
10      }
11
12      private void initFields() {
13        error_ = voldemort.client.protocol.pb.VProto.Error.
14          getDefaultInstance();
15      }
16
17      public final boolean isInitialized() {
18        if (hasError()) {
19          if (!getError().isInitialized()) return false;
20        }
21        return true;
22      }
23
24      public void writeTo(com.google.protobuf.CodedOutputStream
25          output) throws java.io.IOException {
26        getSerializedSize();
27        if (hasError()) {
```

Figura 3.2: Fragmento do *Duplicated Code* identificado no projeto *Voldemort*.

Os dados sobre os identificadores de cada um dos *bad smells* são sumarizados na Tabela 3.1. Baseado nesses identificadores, a abordagem consegue identificar as ocorrências de cada *bad smell* no projeto e verificar se ele foi tratado. Por exemplo, consi-

Bad smells	Identificador(es)	Exemplo
God Class	Nome completo da classe (nome do pacote. nome da classe)	voldemort.src.java.voldemort. client.protocol.pb.VProto
Data Class	Nome completo da classe (nome do pacote. nome da classe)	voldemort.src.java.voldemort. client.protocol.pb.VProto
Long Method	Nome completo da classe e nome do método	voldemort.src.java.voldemort. client.protocol.pb.VProto e initFields
Long Parameter List	Nome completo da classe e nome do método	voldemort.src.java.voldemort .client.protocol.pb.VProto e initFields
Duplicated Code	Caminho do arquivo e o fragmento do código duplicado	"C:\Users\seany\Desktop\ voldemort\src\java\voldemort\ client\protocol\pb\ VAdminProto.java"e o exemplo de fragmento mostrado na Figura

Tabela 3.1: Os identificadores dos *bad smells*.

derando que as versões estão organizadas em ordem cronológica com o sha_i variando seu índice i de 0 até n-1, onde n é o número total de versões em um projeto. Suponha que um *God Class* foi encontrado na versão com sha_k e inserido na lista *godClassesIdentificados*. Para identificar se o *bad smell* foi tratado, a abordagem verifica a partir da versão do projeto com sha_{k+1} se a lista ainda contém o *bad smell*. Caso o *God Class* não estiver presente na lista de alguma versão posterior, a abordagem considera que ele foi tratado. Contudo, caso a busca chegue na última versão do projeto com sha_{n-1} e ele ainda estiver presente na lista, a abordagem considera que o *God Class* não foi tratado. Vale ressaltar que esse processo é válido para os demais *bad smells*.

Por exemplo, com o apoio da abordagem foi possível observar que no projeto *Vlcj*[11] o *bad smell God Class* foi identificado na versão *e07b589* da classe *uk.co.caprica.vlcj. runtime.install.NativeLibraryManager* no dia 18/01/2012 às 12:49 e tratado na versão com *sha 409d6fd* no dia 05/09/2013 às 17:30, permanecendo durante 14.308,68 horas no

[11]https://github.com/caprica/vlcj

repositório.

Após identificar os *bad smells* e seus tratamentos, a abordagem tem como saída uma planilha composta pelas colunas:

- nome do *bad smell*, apresenta o nome de cada um dos cinco *bad smells* analisados;
- quantidade de *bad smells* identificados, que define o número de *bad smells* identificados;
- quantidade de *bad smells* tratados, que apresenta o número de *bad smells* que tiveram tratamento;
- porcentagem de tratamento, que apresenta a porcentagem de *bad smells* que tiveram tratamento;
- duração média, que representa a duração média de um *bad smell no projeto*; e
- duração total, que apresenta o tempo em que o *bad smell* demorou para ser tratado em horas.

Essa tabela é utilizada como entrada para responder às questões de pesquisa propostas neste trabalho.

3.1.1 Extração dos dados

Para ser possível realizar a análise do ciclo de vida dos *bad smells*, conforme discutido na Seção 3.1, é preciso saber o passo a passo de como é feita a extração dos dados. Ou seja, como os *bad smells* são identificados em um projeto.

Os passos para a identificação dos *bad smells* são apresentados no Algoritmo 1. Na linha 1, o algoritmo recebe como entrada o diretório de um projeto e retorna o repositório. Na linha 2 é realizada a identificação das versões de um repositório através da coleta dos identificadores das versões (*e.g.*, *sha* ou índice inteiro que representa uma versão) ordenados cronologicamente. Por exemplo, o projeto *Voldemort* no dia 01/07/2021 possuía 4959 *commits*. Na Figura 3.3 é possível observar às cinco primeiras versões do projeto, cuja primeira é representada pelo identificador *fbd0f95...*.

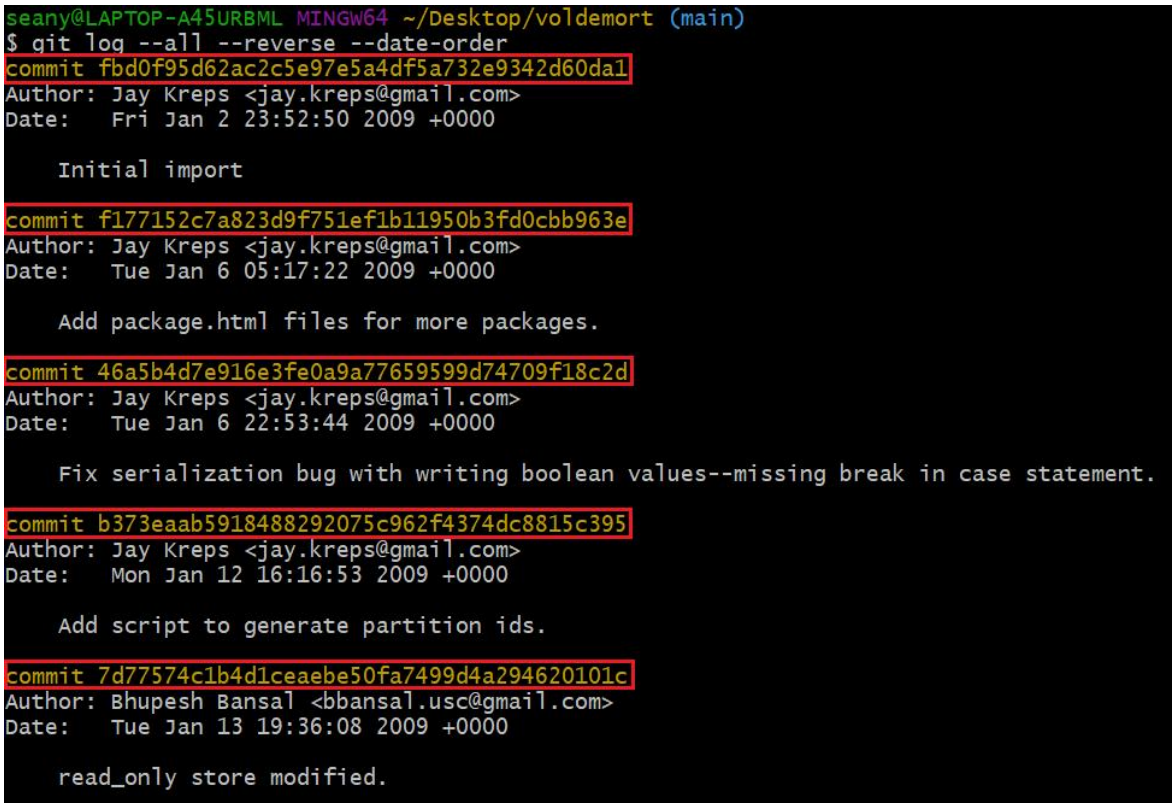

Figura 3.3: Cinco primeiras versões do projeto *Voldemort*.

A linha 3 considera o histórico das versões para visitar cada uma delas e coletar os *bad smells* identificados. Para visitar cada versão, a lista com os identificadores é percorrida e são utilizados os comandos específicos do SCV escolhido para recuperar todos os artefatos presentes nesta versão. Por exemplo, no SCV *Git* é executado o comando *checkout*.

Da linha 4 até a linha 8 são criadas cinco listas para armazenar cada um dos *bad smells* identificados em cada versão com o auxílio de uma ferramenta de detecção de *bad smells*. Por exemplo, com o *PMD*, o *God Class* é localizado no projeto com a ajuda da regra *GodClass* que usa a estratégia de métricas mencionada na Seção 2.3.2. Deste modo, ao identificar um *GodClass*, o algoritmo guarda na *listaGodClass* o identificador do *bad smell* e o identificador da versão. Algo semelhante ocorre na identificação do *Long Parameter List*. Entretanto, neste caso, é utilizada a regra *ExcessiveParameterList*, que reporta quando o número de parâmetros que um método recebe passou do limite permitido, e os dados são armazenados na *listaLongParameterList*.

Durante a extração dos *bad smells*, o sha_i é importante ser armazenado, pois possibilita descobrir se houve o tratamento do *bad smell* em versões futuras. Logo, os *bad smells* armazenados nas listas devem armazenar o sha_i da versão em que o *bad smell* foi identificado e o seu identificador.

Na linha 10 são armazenados os resultados da identificação dos *bad smells* para futuras análises. Vale ressaltar que o resultado de cada lista é armazenado em uma base diferente ordenada cronologicamente para facilitar a verificação se um *bad smell* foi tratado em versões posteriores.

Algoritmo 1 Extração de dados.

```
1: repositorio ← retornaRepositorio(diretorioRepositorio)
2: listaSha ← pegaListaCommitHashGit(repositorio)
3: for listaSha = sha_0, sha_1, ..., sha_i do
4:    listaGodClass ← extraiGodClass(repositorio, sha_i)
5:    listaDataClass ← extraiDataClass(repositorio, sha_i)
6:    listaLongMethod ← extraiLongMethod(repositorio, sha_i)
7:    listaLongParameterList ← extraiLongParameterList(repositorio, sha_i)
8:    listaDuplicateCode ← extraiDataClass(repositorio, sha_i)
9: end for
10: armazenarBadSmells(listaGodClass, listaDataClass, listaLongMethod,
    listaLongParameterList, listaDuplicateCode)
```

3.2 Implementação

Para verificar a viabilidade da abordagem, foi desenvolvida uma prova de conceito em Java que segue a abordagem proposta para determinar o ciclo de vida dos *bad smells*. A implementação utiliza o *PMD* para detectar quais são os *bad smells* identificados em cada versão do projeto de *software*. Ele foi escolhido por ter disponibilidade para *download*, ser independente de IDE, identificar os cinco *bad smells* de interesse desse trabalho e ser amplamente utilizado na comunidade de pesquisa (*e.g,* na área de degradação da arquitetura (LENHARD et al., 2017) e de aprendizado de máquina onde suas técnicas são usadas para detectar *bad smells* (LUJAN et al., 2020)). Finalmente, o SCV utilizado na implementação é o *Git* por ser um dos mais utilizados e por possuir uma ampla quantidade de repositórios públicos na plataforma *GitHub*[12].

[12]https://github.com/

A implementação recebe como entrada um repositório *Git* e extrai a sequência de versões em ordem cronológica. Para obter o repositório é executado o comando de *clone* e a partir dele é extraída a listagem dos identificadores das versões do projeto em ordem cronológica utilizando o comando *git log –all –reverse –pretty=%H –date-order*, onde esses termos servem para:

- *git log*: mostra os *logs* dos *commits*;
- *–all*: lista todos os *commits* independente do ramo;
- *–reverse*: coloca os *commits* escolhidos para serem mostrados na ordem reversa;
- *–pretty=%H*: mostra apenas os *commits* na saída; e
- *–date-order*: coloca os *commits* na ordem cronológica;

Com base na lista com todos os *sha*, é inicializada a varredura da versão com sha_0 até sha_{n-1} por *bad smells*. Vale ressaltar que o *sha* é utilizado pelo *Git* para identificar as versões de um repositório de forma única e o n é o número total de versões. Deste modo, cada versão é visitada para que o *PMD* possa detectar os *bad smells* presentes na versão.

O *PMD* foi integrado ao projeto através do Java *Runtime*[13] que executa comandos no terminal. A Tabela 3.2 ilustra o exemplo do comando *PMD* para coletar cada um dos cinco *bad smells* no terminal *Windows*. Conforme descrito anteriormente, esse processo é repetido para todas as versões do projeto.

Bad smells	Comando no terminal Windows
God Class	*pmd.bat -d caminho -R category/java/design.xml/ GodClass -f xml*
Data Class	*pmd.bat -d caminho -R category/java/design.xml/ DataClass -f xml*
Long Method	*pmd.bat -d caminho -R category/java/design.xml/Excessive MethodLength -f xml*
Long Parameter List	*pmd.bat -d caminho -R category/java/design.xml/Excessive ParameterList -f xml*
Duplicated Code	*cpd.bat –minimum-tokens 100 –files caminho –format xml*

Tabela 3.2: Os comandos para identificar os *bad smells* no *PMD*.

[13]https://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html

O comando *pmd.bat*, utilizado para coletar o *God Class, Data Class, Long Method* e *Long Parameter List*, requisita os parâmetros: -R que define a regra à ser utilizada, -f que define o formato da exportação do relatório e -d que define o caminho um nome de arquivo, diretório, arquivo *jar* ou *zip* contendo os códigos a serem analisados.

O comando *cpd.bat*, utilizado para coletar o *Duplicated Code*, requer os seguintes parâmetros: *–minimum-tokens* que define o comprimento mínimo do *token* que deve ser relatado como uma duplicata, *–files* que define o caminho para os arquivos ou diretórios a serem analisados; e o *–format* que define o formato da exportação do relatório. Vale pontuar que o valor utilizado para o *–minimum-tokens* neste trabalho foi 100.

A etapa de coleta é realizada em paralelo ao processo de detecção dos *bad smells* pelo *PMD* que retorna um XML com as seguintes saídas: *linha de início (beginline), linha de fim (endline), coluna de início (begincolumn), coluna de fim (endcolumn), regra (rule), ruleset (arquivo de configuração XML que descreve a coleção de regras a serem executadas em uma execução PMD), conteúdo (content), classe (classFound), pacote da classe (classPackage), url (externalInfoUrl), nível de prioridade (priority), nome do método (methodName), caminho do arquivo (filePath) e o fragmento duplicado (codefragment)* referentes aos *bad smells*. Essas informações são utilizadas para construir o identificador de cada *bad smell*. A Figura 3.4 ilustra um exemplo do XML exportado durante a coleta do *Data Class* na versão *sha a7dbdea5* do projeto *Voldemort*. Nessa figura, é mostrado a *tag File* que possui o atributo *name* com o caminho onde foi encontrado o *Data Class* e a *tag Violation* que fornece as informações sobre a violação do *bad smell*.

Para essa implementação os dados relevantes para identificar os *bad smells* são *classFound* e *classPackage* no caso dos tipos *God Class* ou *Data Class*. Para *Long Method* ou *Long Parameter List* é incluído o *methodName*. Finalmente, para o *Duplicated Code* são considerados apenas o *filePath* e o *codeFragment*.

Para processar o documento XML gerado pelo *PMD* é utilizada a API SAX[14]. Essa API manipula o documento XML utilizando um *visitor* que dispara eventos que identificam a abertura e o fechamento das *tags*, viabilizando o processamento dos documentos, a coleta dos dados relevantes para identificar quais os *bad smells* presentes em

[14]https://www.devmedia.com.br/processamento-de-xml-em-java-com-a-api-sax/25061

```
<file
        name="C:\Users\seany\Desktop\voldemort\contrib\restclient\src\java\voldemort\
        restclient\RESTClientConfig.java"
>
<violation
        beginline="24" endline="205" begincolumn="8" endcolumn="1" rule="DataClass"
        ruleset="Design" package="voldemort.restclient" class="RESTClientConfig"
        externalInfoUrl="https://pmd.github.io/pmd-6.29.0/pmd_rules_java_design.html#dataclass"
        priority="3"
>
        The class 'RESTClientConfig' is suspected to be a Data Class (WOC=25.000%, NOPA=0,
        NOAM=6, WMC=26)
</violation>
</file>
```

Figura 3.4: Exportação do *Data Class* em XML.

cada versão e o armazenamento desses dados em sua respectiva lista.

Para viabilizar análises posteriores o resultado das coletas são armazenados em um arquivo CSV que contém as seguintes colunas: o nome do projeto examinado, a data do *commit*, o identificador da versão *sha*, o nome do *bad smell* coletado (*e.g, GodClass, DataClass*) e o objeto *bad smell* com todos os dados extraídos do *PMD*, data de identificação e data de tratamento em formato JSON. A Tabela 3.3 mostra um exemplo do armazenamento no arquivo CSV dos *bad smells* coletados no projeto *Repairnator*[15]. Para facilitar a leitura das colunas, o nome do projeto e o objeto *bad smell* (JSON) foram omitidas, e a coluna data do *commit* foi simplificada. A visão completa do arquivo está disponível em um repositório do Google Drive no link ⟨https://drive.google.com/file/d/1tq-_De-qF1_SLjkkEmRhbwQUOI1-Ytwy/view?usp=sharing⟩.

Data do *commit*	*Identificador da versão (sha)*	Tipo do *bad smell*
23/12/2016 11:55:58	42185ef2373e2d511e925f2ce6c26ee59c8cdc6d	DataClass
23/12/2016 11:55:58	42185ef2373e2d511e925f2ce6c26ee59c8cdc6d	GodClass
23/12/2016 11:55:58	42185ef2373e2d511e925f2ce6c26ee59c8cdc6d	DuplicateCode
12/02/2020 09:35:50	c34c1497fce6fc913694fd3c486ff9ab6a0d63c1	LongMethod

Tabela 3.3: Parte do conjunto dos *bad smells* encontrados no projeto *Repairnator*.

Para auxiliar na análise dos dados de *bad smells*, a abordagem recupera os dados armazenados no arquivo CSV. Nesta etapa, cada linha do arquivo é percorrida para carregar os dados e realizar possíveis conversões. Por exemplo, o dado de *bad smell* em

[15]https://github.com/eclipse/repairnator

formato JSON é convertido para objeto Java através da biblioteca Gson[16] e a data do *commit* é convertida de *String*[17] para *Date*[18]. De posse dos dados, o algoritmo preenche uma lista das versões do projeto chamada *projectVersions* pV. Cada posição dessa lista possui o identificador *sha* da versão, data do *commit* e uma lista para cada um dos cinco tipos de *bad smells* coletados.

A execução da análise de identificação é baseada na verificação da primeira ocorrência de cada *bad smell* coletado. Essa análise utiliza os identificadores de *bad smell* da Tabela 3.1 e o nome do *bad smell* para poder diferenciar os *bad smells* e inseri-los na sua lista de identificação correspondente. Para realizar a inserção, é verificado se o *bad smell* está presente na lista de identificação do seu tipo, em caso negativo, ele não foi marcado como identificado ainda. Logo, ele é inserido na lista e tem sua data de identificação alterada para a data do *commit* em que ele foi identificado pela primeira vez.

A partir da identificação do *bad smell*, é inicializada a procura pelo seu tratamento nas versões posteriores. Em outras palavras, a busca pelo tratamento é realizada nas versões posteriores a identificação. Por exemplo, a busca pelo tratamento dos *bad smells* identificados na primeira versão começará atuar a partir da segunda versão do projeto. Então, considerando que pV_j onde j é o índice para percorrer cada versão do projeto na lista pV. Essa análise é verificada pegando a versão do projeto anterior pV_{j-1} e a atual pV_j para compará-las considerando a lista de *bad smell* identificados na versão pV_{j-1} e verificar se ele foi tratado na versão pV_j.

Com base nessa análise, é verificado se algum dos *bad smells* pertencentes à pV_{j-1} não está mais presente na pV_j e se tal *bad smell* já foi identificado anteriormente por meio da confirmação da sua presença na lista de identificação do seu tipo. Caso negativo, nada é efetuado, caso contrário, é constatado que o *bad smell* teve tratamento. Então, dessa maneira, é alterado sua data de tratamento para a data em que a versão atual pV_j ocorreu. Esse procedimento de pegar a versão anterior, atual e a lista de identificação de cada tipo de *bad smell*, vai se repetir para as versões seguintes presentes na lista pV.

Após realizar o processo de identificação e tratamento dos *bad smells*, o resultado

[16]https://github.com/google/gson
[17]https://docs.oracle.com/javase/7/docs/api/java/lang/String.html
[18]https://docs.oracle.com/javase/7/docs/api/java/util/Date.html

dessa análise é gerado e armazenado em uma arquivo CSV contendo: o nome do tipo do *bad smell*, quantidade de *bad smells* identificados, quantidade de *bad smells* tratados, porcentagem de tratamento, duração média e total em que o *bad smell* permaneceu no repositório em horas. Os dados dessa planilha serão utilizados para fazer a avaliação do estudo proposto.

A Tabela 3.4 mostra um exemplo do armazenamento no arquivo CSV dos *bad smells* analisados no projeto *Repairnator*. Para facilitar a leitura das colunas, apenas o total de duração até ser tratado (horas) foi omitido. A visão completa do arquivo está disponível em um repositório do Google Drive no link ⟨https://docs.google.com/spreadsheets/d/1gW2XzRvGrAUruCQW5tfLVDVB5lza4w8pr91mqyC5tv4/\edit?usp=sharing⟩. Vale pontuar que a célula marcada com - representa que o tal *bad smell* não foi encontrado em nenhum momento durante o desenvolvimento do projeto. E, consequentemente, não houve tratamentos.

Tipo do *bad smell*	Quantidade de identificados	Quantidade de tratados	Porcentagem de tratamento	Média de duração até ser tratado (horas)
GodClass	18	10	55,56	6.685,94
DataClass	42	16	38,10	2.628,67
LongMethod	15	11	73,33	2.031,07
LongParameterList	0	0	-	-
DuplicateCode	311	210	67,52	2.396,12

Tabela 3.4: Parte do conjunto dos *bad smells* analisados no projeto *Repairnator*.

3.3 Considerações finais

Nesse capítulo, foi apresentada a abordagem desenvolvida para analisar o ciclo de vida dos *bad smells*. Além de demonstrar o passo a passo de como é realizada a extração dos dados que será utilizada por ela. Em seguida, foi apresentada uma implementação da abordagem para detecção de *bad smells* em projetos na linguagem Java. Portanto, os resultados obtidos através da execução dessa abordagem foram utilizados para avaliar se a adoção de integração contínua influencia na identificação e tratamento dos *bad smells*.

4 Avaliação

Este capítulo define como foi realizada a avaliação para compreender se a IC influencia no ciclo de vida dos *bad smells* durante o desenvolvimento de *software*. Para tanto, este capítulo está organizado em cinco seções. A Seção 4.1 descreve as questões de pesquisas que guiam este estudo. A Seção 4.2 apresenta os critérios para seleção dos oito projetos utilizados neste experimento. A Seção 4.3 aborda as análises que foram realizadas para responder cada uma das três questões de pesquisa. A Seção 4.4 discute os resultados obtidos para cada uma das questões de pesquisa. Finalmente, a Seção 4.5 define os aspectos que podem ameaçar a validade deste estudo.

4.1 Questões de pesquisa

O objetivo deste estudo é verificar se a IC tem influência no ciclo de vida dos *bad smells*. Para atingir esse objetivo, são propostas as seguintes Questões de Pesquisa (QP):

QP1 - Qual a frequência em que os *bad smells* são identificados?

Esta pergunta visa verificar qual a frequência da identificação dos *bad smells* no histórico de desenvolvimento do sistema do *software*. Vale ressaltar, que as frequências são medidas em função da quantidade de *commits* e dias, e serão utilizadas para comparar a frequência na qual os *bad smells* são identificados em projetos que utilizam IC e em projetos que não utilizam IC.

QP2 - Os *bad smells* são tratados?

Esta questão visa investigar se os *bad smells* introduzidos nos projetos são removidos. Ela pode ser respondida considerando o conjunto de *bad smells* identificados e examinar a eventual remoção do *bad smell* do *software*, considerando o histórico de evolução posterior a identificação. Para tanto, serão avaliados os percentuais de tratamentos dos *bad smells* para entender se nos projetos que utilizam IC, os *bad smells* possuem o volume de tratamento maior do que nos projetos não utilizam IC.

QP3 - Quanto tempo os *bad smells* persistem no histórico do projeto

de *software* até serem tratados?

Esta pergunta visa averiguar quanto tempo os *bad smells* demoram para serem tratados. O objetivo é investigar a duração da permanência, em horas, do *bad smell* no sistema de *software* até seu tratamento. Nessa questão é avaliada a quantidade de horas que os *bad smells* demoram para serem tratados, verificando se nos projetos que utilizam IC o intervalo de tempo de tratamento é menor.

4.2 Seleção dos projetos

A análise proposta visa identificar se a adoção da técnica de IC influencia no ciclo de vida dos *bad smells*. Deste modo, foram selecionados projetos de *software* que obedecem aos seguintes critérios:

- ser escrito na linguagem Java, uma das linguagens suportadas pela ferramenta *PMD*;
- ter o acesso público ao repositório do *Github*; e
- possuir pelo menos mil *commits*, estando em um estágio maduro de desenvolvimento e contendo um histórico suficiente para determinar o tempo de vida do *bad smell*.

Devido à limitação de tempo, foram selecionados oito projetos de *software* utilizados pela comunidade de pesquisa para o estudo de *bad smells*. Esses projetos foram divididos em dois grupos: os que utilizam técnicas de IC e os que não utilizam. O grupo que utiliza IC é composto pelos projetos *Commons digester*[19], *Jgnash*[20], *JUnit4*[21] e *Repairnator*, e o grupo que não utiliza IC é composto pelos projetos *Log4j*[22], *Vlcj, Voldemort* e *Vrapper*[23]. Vale ressaltar que dentre esses oito projetos, o *Log4j, JUnit4* e *Commons digester* foram selecionados a partir da leitura do artigo *An Empirical Study on the Evolution of Design Smells* (AVERSANO; CARPENITO; IAMMARINO, 2020) da área de detecção de *bad smells* e os outros cinco foram extraídos do através de buscas realizadas

[19]https://github.com/apache/commons-digester
[20]https://github.com/ccavanaugh/jgnash
[21]https://github.com/junit-team/junit4
[22]https://github.com/apache/log4j
[23]https://github.com/vrapper/vrapper

no *Github*. Os projetos que utilizam IC, *Jgash* e *Repainator*, foram selecionados após observar se o repositório utilizava o *GitHub Actions*[24].

O grupo de projetos que não utilizam IC possuem quantidade de *commits*, duração e domínio variados. Dentre eles, o *Log4j* é um projeto para gestão de registos no ambiente Java que possui 3561 *commits* onde o primeiro *commit* foi realizado em 14/12/2000 e o último ocorreu em 04/06/2015. O *Vlcj* fornece uma estrutura Java para permitir que uma instância de um reprodutor de mídia VLC nativo seja incorporada em uma aplicação Java. Ele possui 2976 *commits* que ocorreram entre os dias 09/05/2010 e 15/05/2021. O *Voldemort* é um sistema distribuído de armazenamento de chave-valor que contém 4959 *commits* onde o primeiro *commit* analisado foi realizado no dia 02/01/2009 e o último no dia 30/08/2017. Finalmente, o *Vrapper* é um plugin Eclipse que atua como um *wrapper* para os editores de texto Eclipse e contém 2509 *commits*, onde o primeiro *commit* foi no dia 29/12/2008 e o último *commit* foi no dia 19/09/2020.

Compondo o conjunto de projetos selecionados para avaliar a utilização de IC, o *Commons digester* é um projeto que permite a configuração de módulos XML. Ele contém 2586 *commits* que ocorreram entre os dias 03/05/2001 e 06/06/2021. O *JUnit4* é um *framework* de código aberto para escrever testes na linguagem de programação Java. Ele possui 2556 *commits* que aconteceram entre os dias 22/04/2003 e 11/06/2021. O *Jgnash* é um projeto oferece um gestor financeiro pessoal gratuito que possui 4562 *commits* onde o primeiro *commit* foi no dia 13/03/2012 e o último ocorreu no dia 05/03/2021. Por fim, o *Repairnator* é um projeto de código aberto para a reparação automática de programas. Ele possui 2976 *commits*, onde o primeiro *commit* ocorreu no dia 09/05/2010 e o último no dia 15/05/2021.

Na Tabela 4.1 é apresentado um resumo destes projetos incluindo o número total de *commits*, a primeira e última data de *commit* e se adota ou não IC. Os projetos selecionados possuem histórico de desenvolvimento que varia de 4 anos a 20 anos e a quantidade de *commits* varia de 1411 a 4959.

[24]https://github.com/features/actions

Projetos	*Commits*	Primeira—Última data de *commit*	contém IC?
Commons digester	2586	03/05/2001 — 06/06/2021	Sim
Jgnash	4562	13/03/2012 — 05/03/2021	Sim
JUnit4	2656	22/04/2003 — 11/06/2021	Sim
Repairnator	1411	21/12/2016 — 17/07/2021	Sim
Log4j	3561	14/12/2000 — 04/06/2015	Não
Vlcj	2976	09/05/2010 — 15/05/2021	Não
Voldemort	4959	02/01/2009 — 30/08/2017	Não
Vrapper	2609	29/12/2008 — 19/09/2020	Não

Tabela 4.1: Projetos selecionados para o experimento.

4.3 Análise dos experimentos

Esta seção está organizada em três subseções que visam responder às três QP propostas na Seção 4.1 baseado nos resultados obtidos a partir dos repositórios dos oito projetos selecionados.

QP1 - Qual a frequência em que os *bad smells* são identificados?

Essa questão busca compreender a frequência em que os *bad smells* são inseridos nos projetos. Além disso, com a utilização de projetos que utilizam IC e não utilizam, será possível verificar se projetos que usam IC possuem alguma diferença com relação à frequência de inserção de *bad smells*.

Para calcular a frequência de ocorrência de *bad smells* são utilizadas duas métricas: a frequência baseada em *commits* e a frequência baseada em dias. Para calcular a frequência da identificação dos *bad smells* em relação ao número de *commits* é utilizada a Equação 4.1 que divide a quantidade de *bad smells* identificados em um projeto pela quantidade de *commits* do projeto. Para calcular a frequência de identificação dos *bad smells* em relação ao número de dias é utilizada a Equação 4.2 que divide quantidade de *bad smells* de um dado tipo pela quantidade de dias entre o primeiro e o último *commit* do projeto. Essas duas frequências fornecem perspectivas diferentes para analisar a frequência em que cada tipo de *bad smell* é identificado em cada um dos oito projetos.

$$fc = \frac{quantidade\ de\ bad\ smells\ identificados}{quantidade\ de\ commits} \tag{4.1}$$

$$fd = \frac{quantidade\ de\ bad\ smells\ identificados}{duração\ (dias)} \tag{4.2}$$

Os gráficos da Figura 4.1 até a Figura 4.5 exibem a frequência na qual os *bad smells* de cada tipo são inseridos nos oito projetos. Cada figura ilustra a frequência baseada na quantidade de *commits*, representadas pelas colunas em verde, e na quantidade de dias, representadas pelas colunas em azul. Além disso, a Figura 4.6 mostra a frequência de identificação considerando a quantidade total de *bad smell* detectados em cada projeto.

Como os projetos que utilizam IC podem não ter iniciado com a adoção de IC, foi realizado um estudo específico para verificar se a adoção de IC gera alguma alteração na frequência de *bad smells* identificados. As Tabelas 4.2 até Tabela 4.7 exibem a frequência em que os *bad smells* são inseridos considerando antes e depois da adoção de IC. Essas tabelas estão organizadas nas seguintes colunas: projetos, que apresenta os nomes dos projetos analisados; fc (antes) e fc (depois), que apresentam as frequências de *bad smells* identificados por *commits* antes e depois da adesão de IC; fd (antes) e fd (depois) que representam as frequências de *bad smells* identificados por dia antes e depois da adesão de IC. É importante ressaltar que foram colocados apenas os quatro projetos que utilizarem IC, para os demais a frequência continua igual como ilustrado nos gráficos das Figuras 4.1 a 4.6. Ademais, as células marcadas com \-" indicam que o dado tipo de *bad smell* não foi encontrado em nenhum momento durante o desenvolvimento do projeto. Finalmente, é importante pontuar que para realizar essa análise da frequência foi feita uma aproximação da data da suposta adoção de IC com base nas documentações disponíveis desses projetos (*e.g*, arquivo README.md do *GitHub*, site oficial do projeto).

O gráfico da Figura 4.1 mostra a distribuição da quantidade de *Long Method* em função da quantidade de *commits* e de dias para os oito projetos. Por exemplo, é válido mencionar que o projeto *JUnit4* não teve nenhuma ocorrência de *Long Method* por isso os dados de sua coluna não estão visíveis. Além disso, com base nos dados do gráfico, é possível afirmar que a maior frequência de *Long Method* ocorreu no projeto *Repairnator* que utiliza IC e a menor frequência ocorreu no projeto *Vlcj* que não utiliza IC, considerando o número de *commits* e o número de dias do projeto.

Esse resultado é confirmado pela análise considerando os dois conjuntos de projetos. Ou seja, o conjunto de projetos que utiliza IC possui uma frequência média maior na identificação do *Long Method* em relação à *commits* e à dias. Os projetos com IC possuem

uma frequência média de 0,0037 identificados por *commit* e de 0,0139 identificados por dia. Em contrapartida, os projetos sem IC possuem frequência média de identificação de 0,0019 por *commit* e de 0,0020 por dia. Ou seja, em média são mais frequentes as identificações de *Long Method* em projetos que utilizam IC.

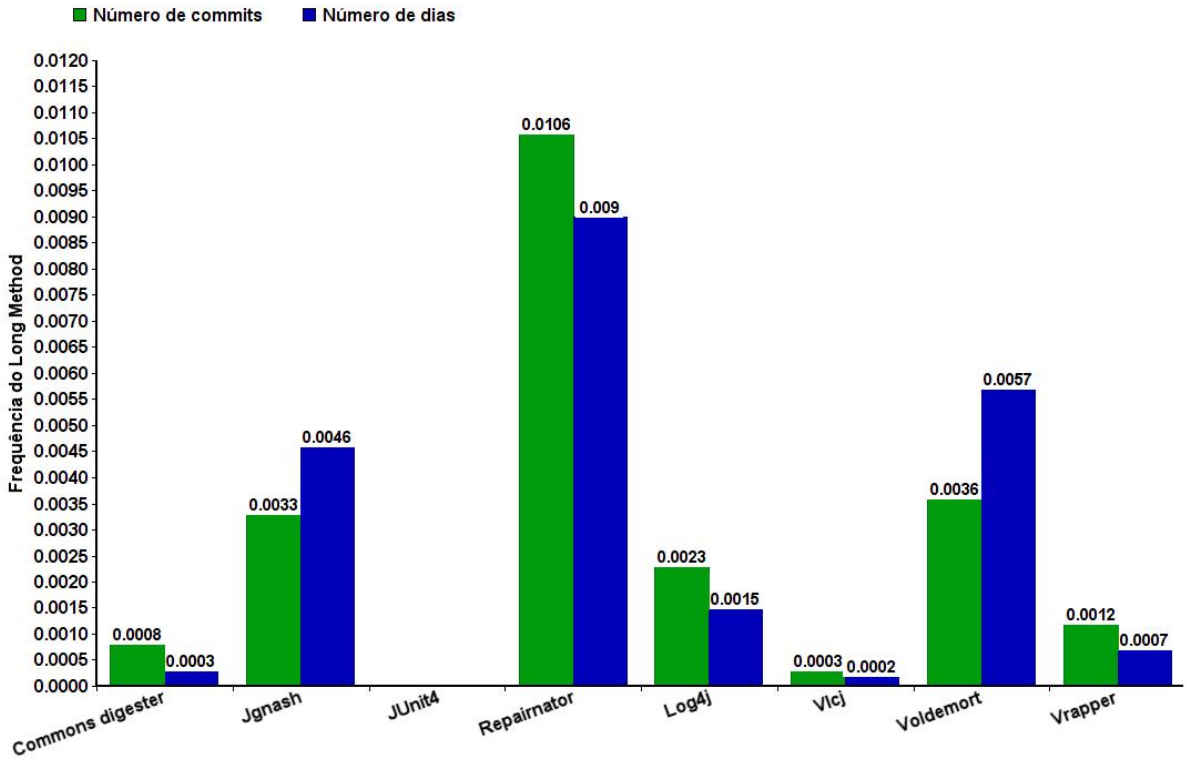

Figura 4.1: Frequência da identificação do *Long Method*.

Conforme discutido anteriormente, os projetos podem não ser concebidos com a adoção de IC e foi realizado um estudo mais profundo para verificar se a adoção de IC no projeto pode gerar alguma alteração na frequência de *Long Methods*. A Tabela 4.2 apresenta a frequência em que o *Long Method* é introduzido considerando o desenvolvimento antes e depois da adoção de IC nos quatro projetos. Nesta tabela é possível visualizar que após a adoção de IC os projetos *Commons digester* e *Repairnator* tiveram aumento na frequência em que o *Long Method* é identificado em função de *commits* e de dias, pois antes de ser adotada não teve nenhuma identificação de *Long Method*. Por outro lado, para o projeto *Jgnash* a frequência de *Long Methods* identificados por *commits* e dias ficou zerada depois da adoção de IC, pois não houve nenhuma identificação de *bad smells* após a IC ser aderida. Vale pontuar que o *Long Method* não foi inserido no projeto *JUnit4*.

Projetos	fc (antes)	fc (depois)	fd (antes)	fd (depois)
Commons digester	0,0000	0,0008	0,0000	0,0003
Jgnash	0,0033	0,0000	0,0046	0,0000
JUnit4	-	-	-	-
Repairnator	0,0000	0,0106	0,0000	0,0268

Tabela 4.2: Frequência do *Long Method* antes e depois da adoção de IC.

O gráfico da Figura 4.2 mostra a frequência de *Data Class* em função da quantidade de *commits* e dias para os projetos. Neste gráfico é possível observar que a maior frequência (em *commits* e em dias) na identificação do *Data Class* ocorreu no projeto *Repairnator* e a menor no projeto *JUnit4*, ambos utilizam IC.

Porém, ao comparar a média da frequência do conjunto de projetos que usam IC com o conjunto dos que não usam, é possível concluir que o grupo de projetos que utilizam IC possuem uma frequência média maior na identificação do *Data Class* em referência à *commits* e à dias. Sendo que os projetos com IC possuem frequência média de identificação de 0,0159 por *commit* e de 0,0111 por dia. Por outro lado, os projetos sem IC possuem uma frequência média de 0,0097 identificados por *commit* e de 0,0073 identificados por dia. Ou seja, em média são mais frequentes as identificações de *Data Class* em projetos que utilizam IC.

A frequência em que o *Data Class* é introduzido considerando o desenvolvimento antes e depois da adoção de IC nos quatro projetos que adotam IC é ilustrada na Tabela 4.3. Nesta tabela é possível visualizar que novamente após a adesão de IC os projetos *Commons digester* e *Repairnator* obtiveram um aumento na frequência em que o *Data Class* é identificado em função de *commits* e de dias. Por outro lado, para os projetos *Jgnash* e *JUnit4* a frequência de identificados por *commits* e de dias ficou zerada após a adesão de IC, pois não houve nenhuma inserção após a adesão de IC.

Projetos	fc (antes)	fc (depois)	fd (antes)	fd (depois)
Commons digester	0,0073	0,0182	0,0026	0,0064
Jgnash	0,0068	0,0000	0,0095	0,0000
JUnit4	0,0015	0,0000	0,0005	0,0000
Repairnator	0,0057	0,0241	0,0048	0,0204

Tabela 4.3: Frequência do *Data Class* antes e depois da adoção de IC.

A distribuição da frequência de *Duplicated Code* em função de *commits* e dias nos oito projetos é ilustrada no gráfico da Figura 4.3. É possível notar que a maior e menor

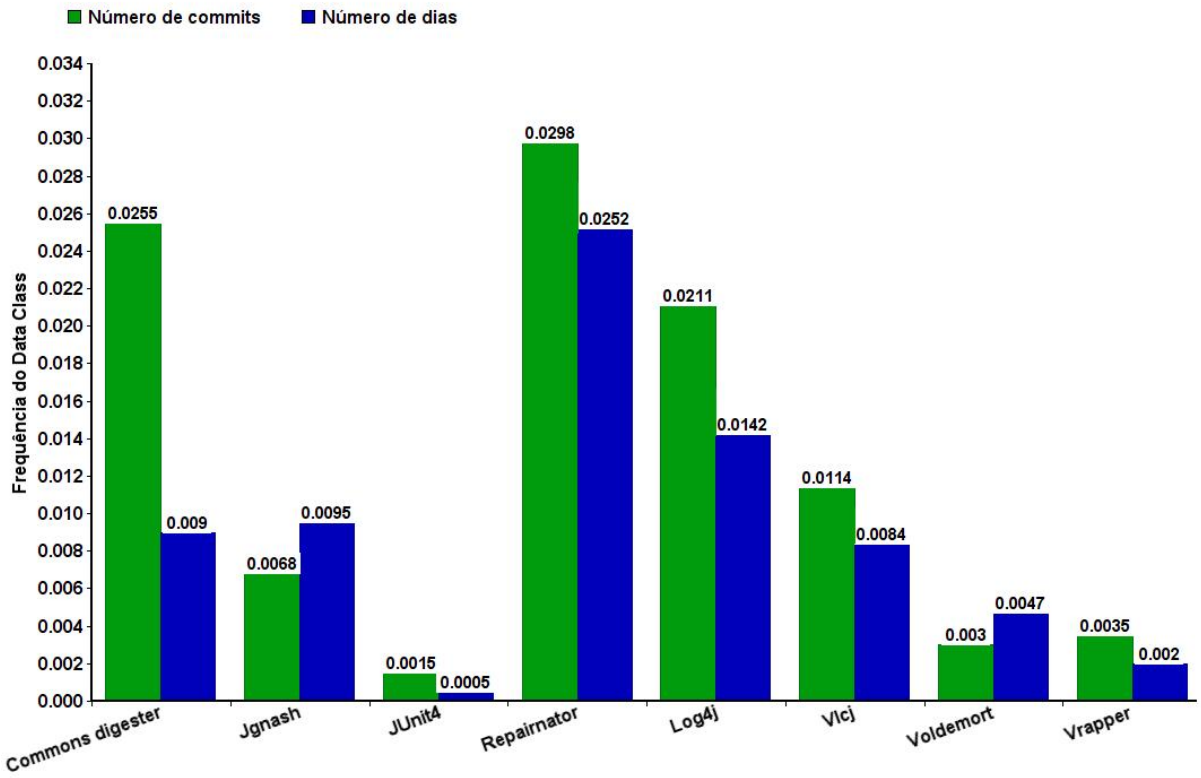

Figura 4.2: Frequência da identificação do *Data Class*.

frequência (em *commits* e em dias) na identificação do *Duplicated Code* foram em projetos que não utilizam IC, sendo a maior no projeto *Log4j* e a menor no projeto *Vrapper*. Além disso, ao comparar a média da frequência do conjunto de projetos que usam IC com o conjunto dos que não usam, é possível concluir que o conjunto de projetos que não utilizam IC possui uma frequência média maior na identificação do *Duplicated Code* considerando o número de *commits* e dias. Os projetos sem IC possuem frequência média de identificação de 0,2127 por *commit* e de 0,1695 por dia. Enquanto, os projetos com IC possuem uma frequência média de 0,0643 identificados por *commit* e de 0,0073 identificados por dia. Ou seja, em média são mais frequentes as identificações de *Duplicated Code* em projetos que não utilizam IC.

A Tabela 4.4 exibe a frequência em que o *Duplicated Code* é inserido considerando o desenvolvimento dos quatro projetos antes e depois da adoção de IC. Nesta tabela é possível observar que nos projetos *Commons digester* e *Repairnator*, a frequência em que o *Duplicated Code* é identificado em função de *commits* e de dias aumentou depois da adoção de IC. Por outro lado, no projeto *Jgnash* a frequência na identificação de *Duplicated Code*

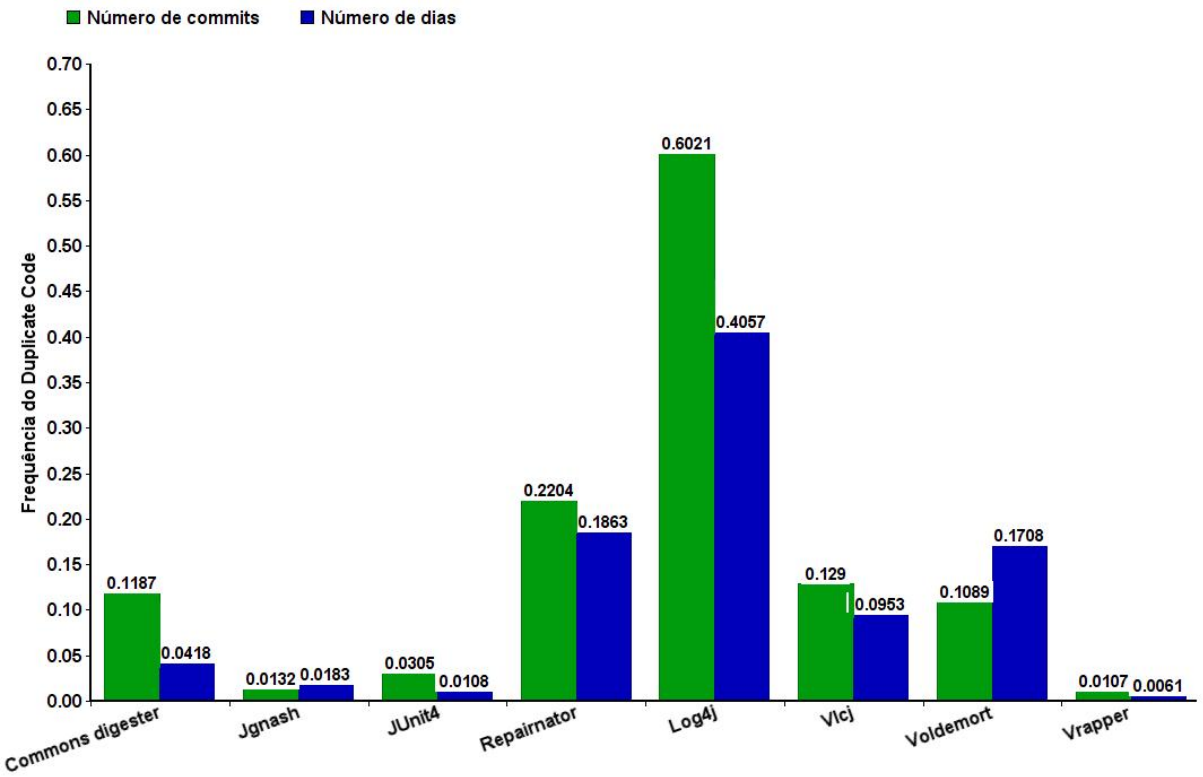

Figura 4.3: Frequência da identificação do *Duplicated Code*.

por *commits* e dias ficou zerada. Por fim, para o *JUnit4* a frequência de *Duplicated Code* identificados diminuiu, pois o número de *Duplicated Code* identificados foi menor após a IC ser aderida.

Projetos	fc (antes)	fc (depois)	fd (antes)	fd (depois)
Commons digester	0,0538	0,0650	0,0189	0,0229
Jgnash	0,0132	0,0000	0,0183	0,0000
JUnit4	0,0297	0,0008	0,0105	0,0003
Repairnator	0,0043	0,2162	0,0036	0,1827

Tabela 4.4: Frequência do *Duplicated Code* antes e depois da adoção de IC.

A distribuição da quantidade de *God Class* em função da quantidade de *commits* e de dias nos oito projetos é ilustrada na Figura 4.4. Nela é possível observar que a maior frequência na identificação do *God Class* em relação aos *commits* foi no projeto *Log4j* e a menor no projeto *Vlcj*, ambos não utilizam IC. Em contrapartida, a maior frequência em relação ao número de dias foi no projeto *Voldemort* e a menor foi novamente no projeto *Vlcj*. Por outro lado, ao comparar a média da frequência do conjunto de projetos que usam IC com o conjunto dos que não usam, é possível concluir que o grupo de projetos

que utilizam IC possui uma frequência média maior na identificação do *God Class* em referência ao número de *commits* e os projetos sem IC possuem uma frequência maior em relação ao número de dias. Sendo que os projetos com IC possuem frequência média de identificação de 0,0089 *God Class* por *commit* e de 0,0076 por dia. Enquanto, os projetos sem IC possuem uma frequência média de 0,0086 *God class* identificados por *commit* e de 0,0086 identificados por dia.

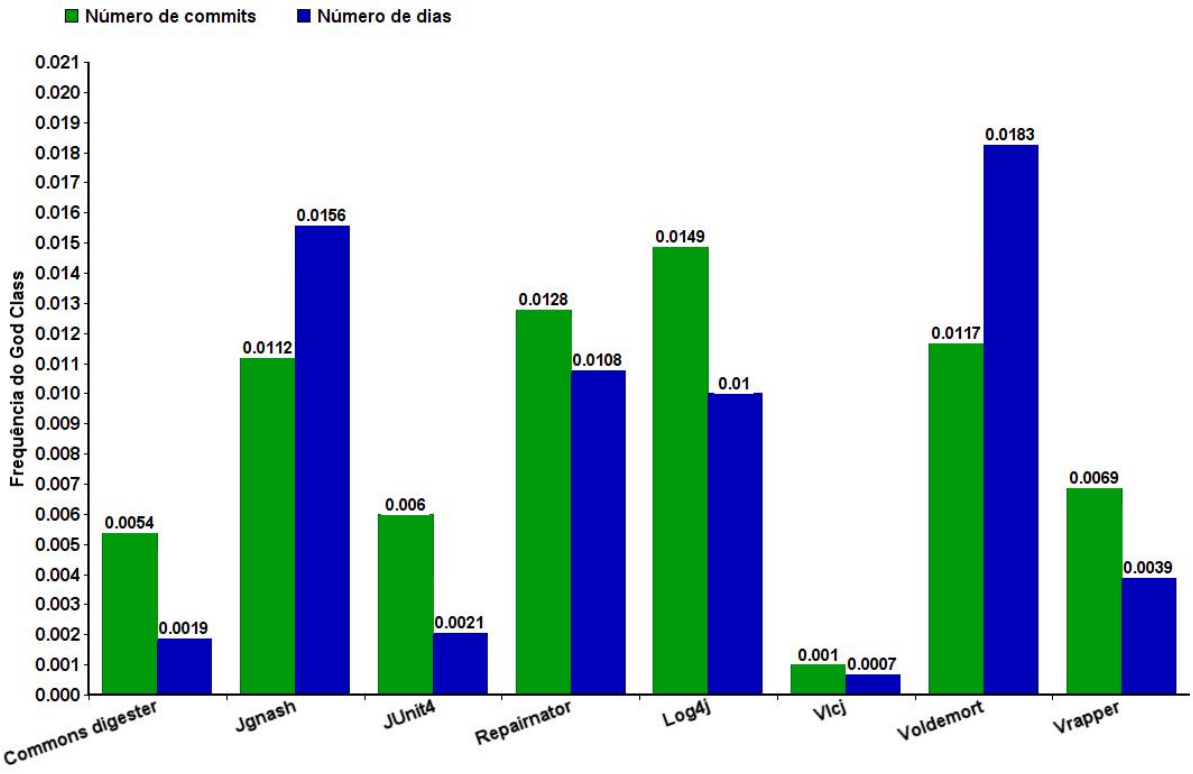

Figura 4.4: Frequência da identificação do *God Class*.

A Tabela 4.5 exibe a frequência em que o *God Class* é inserido considerando o desenvolvimento dos quatro projetos antes e depois da adoção de IC. Nesta tabela é possível observar que no *Repairnator* ocorreu um aumento da frequência em que o *God Class* é identificado em função de *commits* e de dias. Além disso, no *Commons digester* aconteceu um aumento da frequência identificada por *commit*, porém a frequência por dias foi igual antes e depois da adoção de IC. Por outro lado, para os projetos *Jgnash* e *JUnit4* a frequência em que o *God Class* é identificado por *commit* e dias ficou zerada depois de sua adesão, pois não teve nenhum *God Class* inserido no código depois da adesão de IC no projeto.

Para analisar a frequência do *Long Parameter List* nos oito projetos, o gráfico da

Projetos	fc (antes)	fc (depois)	fd (antes)	fd (depois)
Commons digester	0,0000	0,0027	0,0010	0,0010
Jgnash	0,0112	0,0000	0,0156	0,0000
JUnit4	0,0060	0,0000	0,0021	0,0000
Repairnator	0,0021	0,0106	0,0018	0,0090

Tabela 4.5: Frequência do *God Class* antes e depois da adoção de IC.

Figura 4.5 mostra a distribuição da quantidade desse *bad smell* em função do número de *commits* e do número de dias. É válido mencionar que não houve nenhuma identificação de *Long Parameter List* nos projetos *Commons digester*, *Repairnator* e *Vrapper* por isso os dados de suas respectivas colunas não estão visíveis. Além disso, a maior frequência (em *commits*) na identificação do *Long Parameter List* foi no projeto *Log4j* que não utiliza IC e a menor foi no projeto *JUnit4* que utiliza IC. Em contrapartida, a maior frequência (em dias) foi no projeto *Voldemort* que não utiliza IC e a menor foi novamente no projeto *JUnit4*. Por outro lado, comparando a média da frequência do conjunto de projetos que usam IC com o conjunto dos que não usam, conclui-se que o conjunto de projetos que não utilizam IC possui uma frequência média maior na identificação do *Long Parameter List* considerando o número de *commits* e o número de dias. Sendo que os projetos sem IC possuem uma frequência média de 0,0027 identificados por *commit* e de 0,0024 identificados por dia. Em contrapartida, os projetos com IC possuem frequência média de identificação de 0,0004 por *commit* e de 0,0004 por dia.

A frequência em que o *Long Parameter List* é inserido considerando o desenvolvimento antes e depois da adoção de IC nos quatro projetos é mostrada na Tabela 4.6. Nesta tabela é observado que o *Long Parameter List* não foi introduzido nos projetos *Commons digester* e *Repairnator*. Por outro lado, após a adesão de IC os projetos *Jgnash* e *JUnit4* tiveram a frequência de identificação do *Long Parameter List* em função de *commits* e de dias reduzida, devido a não inserção de *Long Parameter List* depois de a IC ser adotada.

Projetos	fc (antes)	fc (depois)	fd (antes)	fd (depois)
Commons digester	-	-	-	-
Jgnash	0,0009	0,0000	0,0012	0,0000
JUnit4	0,0008	0,0000	0,0003	0,0000
Repairnator	-	-	-	-

Tabela 4.6: Frequência do *Long Parameter List* antes e depois da adoção de IC.

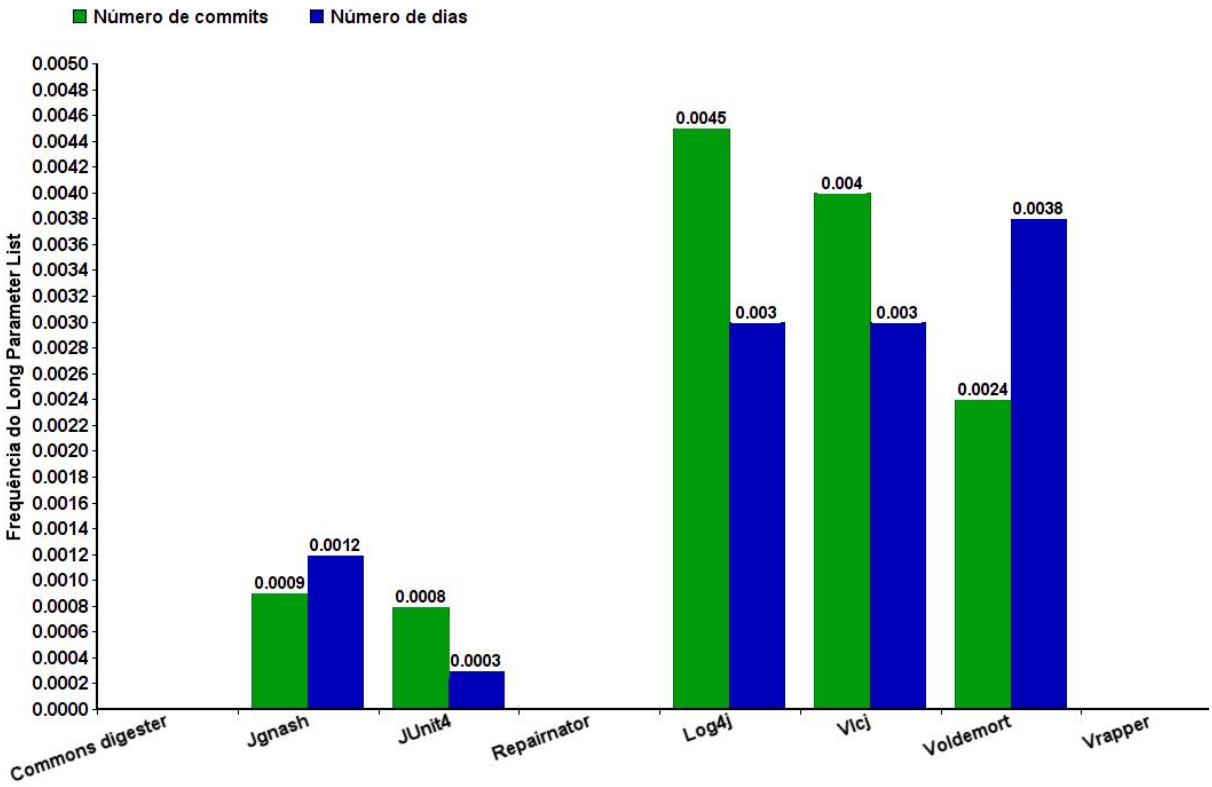

Figura 4.5: Frequência da identificação do *Long Parameter List*.

Por fim, para investigar a frequência dos *bad smells* sem distinção de seu tipo foi realizado o somatório da quantidade dos cinco tipos de *bad smells* identificados em cada projeto para calcular a frequência por *commits*(fc) e dias(fd). O gráfico da Figura 4.6 ilustra a frequência dos *bad smells*. É possível observar que a maior e a menor frequência (em *commits* e em dias) na identificação do *bad smell* em geral foram em projetos que não utilizam IC, sendo a maior no projeto *Log4j* e a menor no projeto *Vrapper*. Ademais, ao comparar a média da frequência do conjunto de projetos que usam IC com o conjunto dos que não usam, é possível notar que o conjunto de projetos que não utilizam IC possui uma frequência média maior na identificação do *bad smell* em referência à *commits* e à dias. Sendo que os projetos sem IC possuem frequência média de identificação de 0,2181 por *commit* e de 0,1621 por dia. Enquanto, os projetos com IC possuem uma frequência média de 0,1246 identificados por *commit* e de 0,0869 identificados por dia.

A frequência em que os *bad smells* são inseridos levando em consideração o desenvolvimento antes e depois da adoção de IC nos quatro projetos é mostrada na Tabela 4.7. Nesta tabela é observado que para os projetos *Commons digester* e *Repairnator*, a

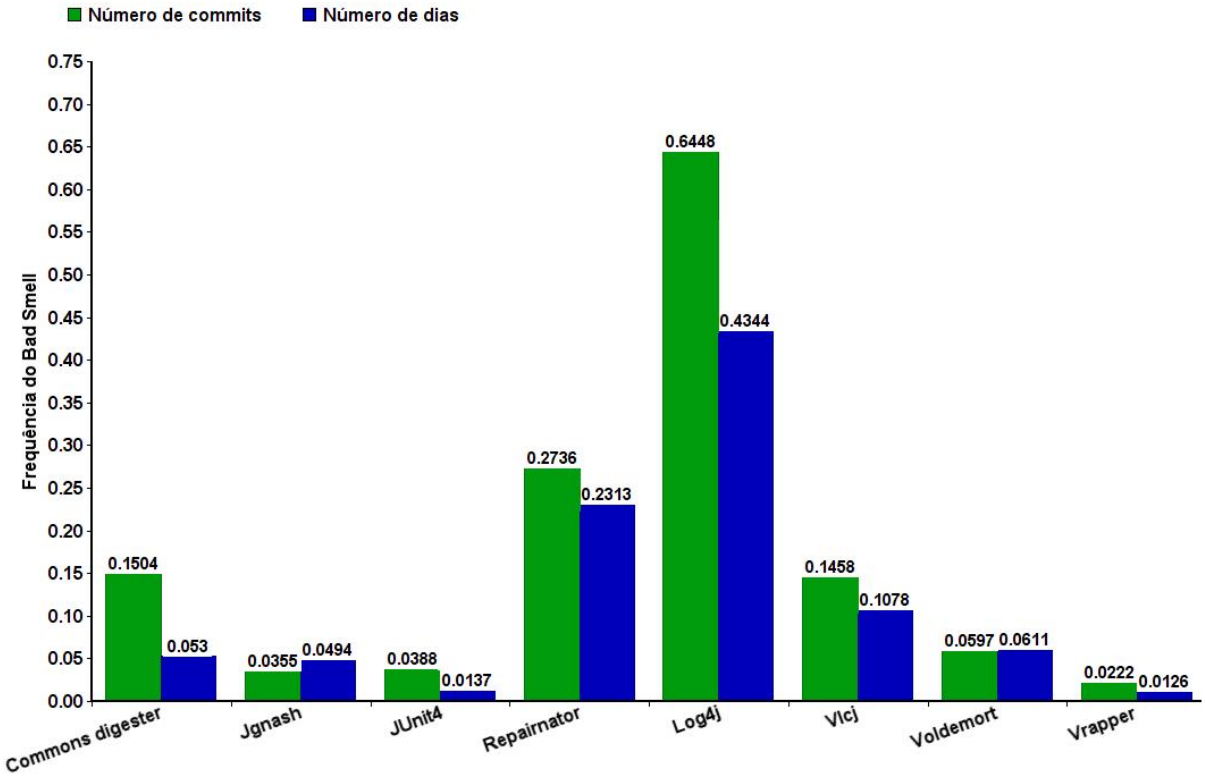

Figura 4.6: Frequência da identificação dos cinco tipos de *bad smell* juntos.

frequência em que um *bad smell* é identificado em função de *commits* e dias aumentou depois da adoção de IC. Por outro lado, no projeto *Jgnash* as frequências ficaram zeradas após a IC, pois nenhum dos cinco tipos de *bad smells* analisados foi inserido depois da IC ser adotada. Por fim, no projeto *JUnit4* a frequência na identificação de *bad smell* por *commits* e dias diminuiu, pois houve a redução de *bad smells* identificados depois de sua adesão.

Projetos	fc (antes)	fc (depois)	fd (antes)	fd (depois)
Commons digester	0,0638	0,0866	0,0225	0,0305
Jgnash	0,0353	0,0000	0,0491	0,0000
JUnit4	0,0380	0,0008	0,0135	0,0003
Repairnator	0,012	0,2608	0,0102	0,2205

Tabela 4.7: Frequência de *bad smells* antes e depois da adoção de IC.

QP2 - Os *bad smells* são tratados?

Essa questão de pesquisa procura entender se os *bad smell* identificados no projeto são tratados. Deste modo, é possível verificar se o desenvolvedor se preocupa com a remoção de *bad smell* para a prevenção do envelhecimento do *software*. Para responder essa pergunta, é aplicada a Equação 4.3 que calcula a porcentagem de tratamento de *bad smells* considerando a quantidade de *bad smells* tratados dividido pela quantidade de *bad smells* identificados.

$$tratamento = \frac{quantidade\ de\ bad\ smells\ tratados}{quantidade\ de\ bad\ smells\ identificados} \tag{4.3}$$

A Tabela 4.8 mostra as porcentagens de tratamento dos *bad smells* em cada projeto. Está tabela está dividida em sete colunas. A coluna de projetos apresenta os nomes dos oito projetos analisados e as demais colunas apresentam os percentuais de tratamentos de cada um dos cinco tipos de *bad smell* em cada projeto. Por fim, na coluna *Bad smell* é apresentado o percentual considerando o total geral de *bad smells*. Vale pontuar que as células marcadas com − indicam que o tipo de *bad smell* não foi encontrado em nenhum momento durante o desenvolvimento do projeto. Por exemplo, o *bad smell Long Parameter List* não foi encontrado no projeto *Repairnator*.

Projetos	*Long Method*	*Data Class*	*Duplicated Code*	*God Class*	*Long Parameter List*	*Bad smell*
Commons digester	100,00%	100,00%	100,00%	100,00%	-	100,00%
Jgnash	80,00%	83,87%	0,00%	64,71%	75,00%	45,96%
JUnit4	-	100,00%	95,06%	81,25%	100,00%	93,20%
Repairnator	73,33%	38,10%	67,52%	55,56%	-	63,99%
Log4j	100,00%	100,00%	99,81%	100,00%	100,00%	99,83%
Vlcj	100,00%	100,00%	97,92%	100,00%	66,67%	97,24%
Voldemort	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
Vrapper	0,00%	55,56%	60,71%	22,22%	-	44,83%

Tabela 4.8: Porcentagem de tratamento dos *bad smells* nos projetos.

Com base na análise desses dados, a Equação 4.4 é utilizada para calcular a porcentagem média de *bad smells* tratados no conjunto de projetos que utilizam IC e o conjunto de projetos que não utilizam, considerando n o número de projetos no conjunto. Baseado nesses dados, os projetos que utilizam IC tiveram mais tratamentos, 75,79%, do

que os projetos que não utilizam IC, 60,48%.

$$média = \frac{\sum_{i=1}^{n} tratamento}{n} \tag{4.4}$$

Após realizar o cálculo da Equação 4.4 para os **bad smells**. Os resultados mostram que nos projetos com IC, o *Long Method*, *Data Class*, *Duplicated Code*, *God Class*, *Long Parameter List* e *bad smells* foram tratados numa porcentagem média de 63,33%, 80,49%, 65,65%, 75,38%, 43,75% e 75,79%, respectivamente. Por outro lado, em projetos sem IC foram tratados 50%, 63,89%, 64,61%, 55,56%, 41,67% e 60,48%, respectivamente.

Analisando os dados da Tabela 4.8, ela mostra que no projeto *Commons digester*, todos os *bad smells* identificados foram tratados. Os projetos *JUnit4, Repairnator Log4j* e *Vlcj* também tiveram alto índice de tratamento dos *bad smells* tendo quase 100% de tratados. Por outro lado, o *Jgnash* e *Vrapper* tiveram quase metade dos *bad smells* tratados. Já, o projeto *Voldemort* não teve nenhum *bad smell* tratado.

Como discutido anteriormente, os projetos que utilizam IC podem não adotá-la desde o início. Para analisar as porcentagens de tratamento dos *bad smells* considerando antes e depois da adoção de IC são utilizadas as Tabelas 4.9 e 4.10 que foram divididas para melhorar a visualização. A coluna projetos apresenta os nomes dos projetos analisados. Nas colunas, pt (antes) e pt (depois) são apresentadas a porcentagem de tratamento de *bad smells* antes e depois da adesão de IC. Vale ressaltar que foram colocados apenas estes quatro projetos por utilizarem IC, para os demais a porcentagem continua valendo igual mostra a Tabela 4.8. Ademais, as células marcadas com "-"representam que o dado tipo de *bad smell* não foi encontrado em nenhum momento durante o desenvolvimento do projeto, então não foi calculado a porcentagem. Além disso, vale relembrar que para realizar essa análise da porcentagem de tratamento foi feito uma aproximação da data da suposta adoção de IC com base nas documentações disponíveis desses projetos (*e.g*, arquivo README.md do *GitHub*, site oficial do projeto).

A Tabela 4.9 mostra as porcentagens de tratamento do *God Class, Data Class* e *Long Method* considerando os projetos antes e depois da adesão de IC. Com ela é possível observar que nos projetos *Jgnash* e *JUnit4*, esses *bad smells* tiveram o volume de tratamento maior antes da IC ser aderida, sendo que todos os tratamentos ocorreram

Projetos	*God Class*		*Data Class*		*Long Method*	
	pt(antes)	pt(depois)	pt(antes)	pt(depois)	pt(antes)	pt(depois)
Commons digester	50,00%	50,00%	28,79%	71,21%	0,00%	100,00%
Jgnash	64,71%	0,00%	83,87%	0,00%	100,00%	0,00%
JUnit4	81,25%	0,00%	100,00%	0,00%	-	-
Repairnator	0,00%	55,56%	11,90%	26,19%	0,00%	73,33%

Tabela 4.9: Porcentagem de tratamento do *God Class, Data Class* e *Long Method* nos projetos com IC.

Projetos	*Long Parameter List*		*Duplicated Code*		*Bad smell*	
	pt(antes)	pt(depois)	pt(antes)	pt(depois)	pt(antes)	pt(depois)
Commons digester	-	-	45,28%	54,72%	42,42%	57,58%
Jgnash	75,00%	0,00%	0,00%	0,00%	45,96%	0,00%
JUnit4	100,00%	0,00%	95,06%	0,00%	93,20%	0,00%
Repairnator	-	-	0,00%	67,52%	1,30%	62,69%

Tabela 4.10: Porcentagem de tratamento do *Long Parameter List, Duplicated Code* e *Bad smell* nos projetos com IC.

antes de sua adesão e nenhum tratamento ocorreu depois. Vale ressaltar, o *Long Method* não foi encontrado no projeto *JUnit4*, então o cálculo do tratamento não foi realizado. Por outro lado, nos projetos *Commons digester* e *Repairnator* tiveram o volume de tratados igual ou maior após a adesão de IC.

A Tabela 4.10 mostra as porcentagens de tratamento do *Long Parameter List, Duplicated Code* e de todos os *bad smell* considerando os projetos antes e depois da adesão de IC. Com ela é possível notar que o *Duplicated Code* e os *bad smell* em geral nos projetos *Commons digester* e *Repairnator* tiveram aumento no volume de tratamento após a IC ser aderida. Além disso, nesses dois projetos o *Long Parameter List* não foi encontrado, então o cálculo do tratamento não foi realizado. Entretanto, nos projetos *Jgnash* e *JUnit4* o número de *Long Parameter List* tratados depois da adoção de IC ficou zerado. Ademais, no projeto *Jgnash* nenhum *bad smell* foi tratado depois da adoção de IC e o *Duplicated Code* não teve tratamento em nenhum momento do projeto. Finalmente, no projeto *JUnit4*, nenhum *bad smells* teve tratamento depois da adoção de IC.

QP3 - Quanto tempo os *bad smells* persistem no histórico do projeto de *software* até serem tratados?

Após saber que os *bad smells* são identificados e tratados, essa questão visa verificar em quanto tempo os *bad smells* são tratados considerando os intervalos de tempo em horas. A duração média de cada tipo de *bad smell* para cada projeto é calculada utilizando a Equação 4.5 que divide o somatório da diferença em horas entre a data de tratamento e a data de identificação do *bad smell* pela quantidade de *bad smells* identificados.

$$duração = \frac{\sum_{i=1}^{n}(data\ de\ tratamento\ -\ data\ de\ identificação)}{quantidade\ de\ bad\ smells\ identificados} \tag{4.5}$$

As Tabelas 4.11 e 4.12 ilustram as durações em horas que os *bad smells* permaneceram no repositório até serem tratados para cada projeto. Vale ressaltar que na Tabela 4.11 os *bad smells* não tratados são excluídos do cálculo. Por outro lado, na Tabela 4.12 eles são incluídos e o intervalo entre a identificação e o tratamento considera a diferença em horas da identificação té a data do último *commit*. Essas tabelas estão organizadas em colunas, na coluna projetos são apresentados os nomes dos oito projetos analisados e nas demais colunas são apresentados os intervalos em que cada um dos cinco tipos de *bad smell* foram tratados. Por fim, a coluna *Bad smell* exibe o intervalo de tempo em horas no qual um *bad smell*, sem considerar seu tipo, demora para ser tratado em média. Vale pontuar que as células marcadas com $-$ indicam que o tipo de *bad smell* não foi encontrado e as células marcadas com x representam que não ocorreu o tratado do *bad smell* durante o desenvolvimento do projeto. Essa observação será válida para as demais tabelas.

A Tabela 4.11 mostra que os cinco tipos de *bad smell* identificados no projeto *Voldemort* e o *Long Method* do projeto *Vrapper*, que não utilizam IC, nunca foram tratados. Além disso, no projeto *Jgnash*, que utiliza IC, o *Duplicated Code* não teve nenhum tratamento.

Além de mostrar que o *Long Method*, *Data Class*, *Duplicated code* e *Long Parameter List* tiveram o ciclo de vida maior nos projetos *Commons digester*, *Jgnash* e *JUnit4*, que utilizam IC. Já o *God Class* durou mais tempo no projeto *Vlcj* que não utiliza IC. Em média, os *bad smells* tiveram o ciclo de vida maior em projetos com IC.

Por outro lado,a Tabela 4.11 mostra que o *Long Method* e *Data Class* tiveram o ciclo de vida menor nos projetos *Repairnator* e o *God Class* permaneceu por menos

tempo no *Commons digester*, que utilizam IC. Por outro lado, o *Duplicated code* e *Long Parameter List* duraram menos tempo no projeto *Log4j*, que não utiliza IC. Em média, os *bad smells* tiveram o ciclo de vida menor em projetos sem IC.

Com base na análise desses dados, é possível concluir que os *bad smells* em projetos que não utilizam IC possuem o tratamento mais rápido, quando são tratados. Esse fato pode ter ocorrido por não considerar as durações dos *bad smells* que nunca foram tratados.

Projetos	*Long Method*	*Data Class*	*Duplicated Code*	*God Class*	*Long Parameter List*	*Bad smell*
Commons digester	19.701,50	6.862,76	1.914,40	3.741,36	-	2.911,17
Jgnash	11.803,87	12.592,52	x	21.715,84	16.974,00	14.771,13
JUnit4	-	3.516,25	4.456,02	9.310,25	33.995,00	5.747,16
Repairnator	2.031,07	2.628,67	2.396,12	6.685,94	-	2.609,11
Log4j	3.658,75	3.748,00	639,00	5.054,43	2.174,69	863,70
Vlcj	8.697,00	8.110,82	3.487,10	25.438,67	11.838,00	4.243,97
Voldemort	x	x	x	x	x	x
Vrapper	x	11.968,56	3.391,96	4.455,28	-	4.877,16

Tabela 4.11: Duração média dos *bad smells* (excluindo os não tratados) até serem tratados nos projetos.

Foi visto anteriormente que ao excluir os *bad smells* não tratados da análise, em média, os *bad smells* tem o ciclo de vida menor em projetos que não utilizam IC. Por outro lado, considerando a data de tratamento dos *bad smells* que nunca foram tratados como a data do último *commit* do projeto é possível concluir que os *bad smells*, em média, duraram menos tempo em projetos que utilizam IC. A Tabela 4.12 mostra essa observação, onde todos *bad smells* no projeto *Voldemort*, *Long Method* no *Vrapper* e *Duplicated Code* no *Jgnash* receberam o valor de sua média de duração em cada um desses projetos.

Com a Tabela 4.12 é possível mencionar que todos os cinco tipos de *bad smells* analisados se mantiveram por mais tempo no projeto *Voldemort* que não utiliza IC. Ademais, em média, os *bad smells* permaneceram por mais tempo no conjunto de projetos que não adotaram IC, quando todos os *bad smell* identificados são considerados.

Nessa análise, os resultados para os *bad smells Duplicated Code, God Class* e *Long Parameter List* se mantiveram iguais aos obtidos anteriormente pela Tabela 4.11.

Por outro lado, o *Long Method* teve o ciclo de vida menor no projeto *Log4j* que não utiliza IC. O *Data Class* permaneceu por menos tempo no projeto *JUnit4* que utiliza IC. Ademais, em média, os *bad smells* obtiveram seus tratamentos mais rapidamente no conjunto de projetos que adotaram IC.

Com a análise nesses dados, é possível concluir que os *bad smells* em projetos que utilizam IC possuíram o tratamento mais rápido. Isso se deu pelo fato de terem sido incluídas as durações dos *bad smells* que nunca foram tratados.

Projetos	*Long Method*	*Data Class*	*Duplicated Code*	*God Class*	*Long Parameter List*	*Bad smell*
Commons *digester*	19.701,50	6.862,76	1.914,40	3.741,36	-	2.911,17
Jgnash	20.075,73	21.762,16	78.686,00	41.203,04	36.645,25	53.293,05
JUnit4	-	3.516,25	6.676,78	28.304,19	33.995,00	10.444,09
Repairnator	4.084,07	9.317,86	5.562,48	11.598,50	-	6.196,94
Log4j	3.658,75	3.748,00	623,59	5.054,43	2.174,69	883,69
Vlcj	8.697,00	8.110,82	3.897,68	25.438,67	18.393,42	4.788,50
Voldemort	75.888,00	75.888,00	75.888,00	75.888,00	75.888,00	75.888,00
Vrapper	50.108,33	33.169,44	21.367,43	49.749,11	-	33.493,48

Tabela 4.12: Duração média dos *bad smells* (incluindo os não tratados) até serem tratados nos projetos.

Para analisar quanto tempo os *bad smells* duram no projeto considerando antes e depois da adoção de IC. As Tabelas 4.13 e 4.14 exibem as médias das durações em horas que os *bad smells* (excluindo os não tratados) permaneceram até serem tratados em cada projeto. Na coluna de projetos, são apresentados os nomes dos quatro projetos analisados. Nas demais colunas são apresentados o dm (antes) e dm (depois) que representa duração média em horas que demorou para o tipo de *bad smell* ser tratado antes e depois da IC.

Projetos	*God Class*		*Data Class*		*Long Method*	
	dm(antes)	dm(depois)	dm(antes)	dm(depois)	dm(antes)	dm(depois)
Commons digester	3.450,00	291,36	4.993,59	1.869,17	0,00	19.701,50
Jgnash	21.715,84	0,00	12.592,52	0,00	11.803,87	0,00
JUnit4	9.310,25	0,00	3.516,25	0,00	-	-
Repairnator	0,00	6.685,94	2,05	2.626,62	0,00	2.031,07

Tabela 4.13: Duração média do *God Class, Data Class* e *Long Method* desconsiderando os não tratados nos projetos com IC.

Projetos	*Long Parameter List*		*Duplicated Code*		*Bad smell*	
	dm(antes)	dm(depois)	dm(antes)	dm(depois)	dm(antes)	dm(depois)
Commons digester	-	-	1.072,31	842,09	1.817,68	1.093,49
Jgnash	16.974	0,00	x	x	14.771,13	0,00
JUnit4	33.995,00	0,00	4.456,02	0,00	5.747,16	0,00
Repairnator	-	-	0,00	2.396,12	2,05	2607,06

Tabela 4.14: Duração média do *Long Parameter List, Duplicated Code* e *bad smell* desconsiderando os não tratados nos projetos com IC.

A Tabela 4.13 exibe as durações médias do *God Class*, *Data Class* e *Long Method* até serem tratados nos projetos que utilizam IC. Com ela é possível observar que no projeto *Commons digester*, o *God Class* e o *Data Class* tiveram o ciclo de vida reduzido depois da adesão de IC. Entretanto, o *Long Method* aumentou pelo fato de ter sido inserido somente após a IC ser aderida. Vale ressaltar que mesmo ocorrendo esse aumento para o *Lonsg Method* neste projeto, todos os *Long Method* identificados foram tratados. Ademais, os projetos *Jgnash* e *JUnit4* tiveram o tempo de tratamento zerado após adoção IC, pois não foram identificados esses tipos *bad smells* depois da adoção de IC. Por fim, no *Repairnator* os três tipos de *bad smell* duram em média mais tempo. Porém, o volume de tratados também aumentou após IC.

A Tabela 4.14 exibe quanto tempo em média os outros *bad smells*, *Long Parameter Liste Duplicated Code*, e todos os *bad smell* demoram para serem tratados nos projetos que utilizam IC. Com ela é possível observar que no projeto *Commons digester*, o *Duplicated Code* e os *bad smells* ao todo tiveram o ciclo de vida reduzido depois da adesão de IC. Além de não ter sido introduzido nenhum *Long Parameter List.* Os projetos *Jgnash* e *JUnit4* tiveram sua duração zerada após adoção IC, pois após ser adotada nenhum desses *bad smells* foram identificados nem tratados. Vale ressaltar que no *Jgnash*, o *Duplicated Code* nunca teve tratamento. Por fim, no *Repairnator* os *bad smells* duram em média mais tempo no projeto após a IC ser aderida, todavia o volume de tratados também aumentou, pois a maioria os tratamentos ocorreram após sua adesão.

Após a analisar as durações dos *bad smells* sem considerar os *bad smells* não tratados. As Tabelas 4.15 e 4.16 exibem quanto tempo em horas que os *bad smells* (incluindo os não tratados) permaneceram até serem tratados em cada projeto considerando antes e

depois da adesão de IC. Sendo assim, os *bad smells* não tratados são incluídos no cálculo. Na coluna de projetos, são apresentados os projetos analisados. Nas demais colunas são apresentadas a dm (antes) e dm (depois) que representa a duração média em horas para o tipo de *bad smell* ser tratado antes e depois da IC. Vale pontuar que a célula marcada com − significa que o tal tipo de *bad smell* não foi encontrado.

Projetos	*God Class*		*Data Class*		*Long Method*	
	dm(antes)	dm(depois)	dm(antes)	dm(depois)	dm(antes)	dm(depois)
Commons digester	3.450,00	291,36	4.993,59	1.869,17	0,00	19.701,50
Jgnash	21.715,84	19.487,20	12.592,52	9.169,65	11.803,87	8.271,87
JUnit4	9.310,25	18.993,94	3.516,25	0,00	-	-
Repairnator	0,00	11.598,50	2,05	9.315,81	0,00	4.084,07

Tabela 4.15: Duração média do *God Class, Data Class* e *Long Method* considerando os não tratados nos projetos com IC.

Projetos	*Long Parameter List*		*Duplicated Code*		*Bad smell*	
	dm(antes)	dm(depois)	dm(antes)	dm(depois)	dm(antes)	dm(depois)
Commons digester	-	-	1.072,31	842,09	1.817,68	1.093,49
Jgnash	16.974,00	19.671,25	0,00	78.686,00	14.771,13	38.521,92
JUnit4	33.995,00	0,00	4.456,02	2.220,75	5.747,16	4.696,93
Repairnator	-	-	0,00	5.562,48	2,05	6.194,90

Tabela 4.16: Duração média do *Long Parameter List, Duplicated Code* e *bad smell* considerando os não tratados nos projetos com IC.

A Tabela 4.15 mostra a análise da duração do *God Class*, *Data Class* e *Long Method* até serem tratados nos projetos que utilizam IC. Com ela é observado que no projeto *Commons digester* novamente o *God Class* e *Data Class* tiveram o ciclo de vida reduzido depois da adesão de IC. Entretanto, o *Long Method* teve o aumento na duração pelo fato de ter sido inserido somente após a IC ser aderida. No *Jgnash*, os três tipos de *bad smell* tiveram o tempo de duração reduzido após a IC ser adotada. No *JUnit4*, o *God Class* teve aumento do tempo, pois os que não foram tratados antes da IC foram tratados depois. A duração do *Data Class* ficou zerada após a IC, pois todos os *Data Class* foram inseridos e tratados antes de a IC ser aderida. Finalmente, no *Repairnator* houve aumento do tempo, pois a maioria dos *bad smells* foram adicionados depois da IC.

A Tabela 4.16 mostra a análise da duração do *Long Parameter*, *Duplicated Code*

e *bad smells* em geral até serem tratados nos projetos que utilizam IC. Com ela é possível observar que no projeto *Commons digester* os *bad smells* em geral tiveram uma duração menor depois da IC. Por outro lado, no *Jgnash*, esses *bad smells* tiveram o tempo de duração aumentada após a IC. No *JUnit4*, o *Duplicated Code* e os *bad smells* em geral duram menos tempo, pois a maioria dos *bad smells* que foram inseridos antes da IC foram tratados apenas depois da adoção de integração contínua e a duração do *Long Parameter List* ficou zerada depois da IC, pois todos os *Long Parameter List* foram inseridos e tratados antes da IC. Finalmente, no *Repairnator* houve aumento do tempo, pois a maioria dos *bad smells* foram adicionados depois da IC.

4.4 Discussão dos resultados

Com base nos resultados dos experimentos obtidos a partir dos oito sistemas de *software* é possível responder às três questões de pesquisa. Nessa seção são apresentadas as respostas objetivas de cada uma das três questões de pesquisa.

QP1 - Qual a frequência em que os *bad smells* são identificados?

Durante a investigação, em média, é notado que a frequência, considerando *commits* e dias, de identificação dos *bad smells Long Method* e *Data Class* são mais frequentes em projetos que utilizam IC. No entanto, para o *Duplicated Code* e o *Long Parameter List* são mais frequentes em projetos que não utilizam IC. Por outro lado, o *God Class* teve a média de identificação por *commits* maior em projetos que utilizam IC e a média por dia foi maior em projetos que não utilizam IC. Finalmente, quando a análise de frequência considera os cinco tipos de *bad smells* dos projetos, pode-se observar que a identificação de *bad smells* é mais frequentes em projetos que não utilizam IC.

Para enriquecer os resultados foi incluído a análise da frequência considerando os projetos que adotam IC antes e depois da adoção de IC. Os resultados mostraram que após a adesão de IC, os projetos *Commons digester* e *Repairnator* obtiveram um aumento na frequência em que os *bad smells* em geral são identificados, isso devido ao aumento do número de identificações depois da IC. Por outro lado, esses dois projetos não tiveram nenhuma identificação do *Long Parameter List* durante todo o desenvolvimento. Á primeira vista, esse fenômeno ocorre devido a esses *bad smells* estarem mais associado

ao período de desenvolvimento do que com a IC. Em contrapartida, depois da adoção de IC, os projetos *Jgnash* e *JUnit4* obtiveram a redução da frequência em que os cinco tipos de *bad smells* são identificados. Uma possível explicação pode estar no fato de que a IC foi adotada num estágio mais maduro dos dois projetos. A redução da frequência é positiva ao mostrar que houve um possível amadurecimento dos desenvolvedores em não introduzir novos *bad smells* mostrando que eles podem estar cientes da existência dos *bad smells*.

A IC se mostraria útil em questão da frequência de *bad smells* em vista de que ela faz o monitoramento contínuo do código. Então, caso ocorresse uma alta frequência de *bad smells* introduzidos, os desenvolvedores poderiam ser alertados e ferramentas poderiam ser sugeridas para identificar e tratar os *bad smells* de maneira mais eficiente. Por outro lado, em caso de redução da frequência de *bad smells* pode dar indícios que a IC, por meio do *feedback* rápido, pode indicar aos desenvolvedores que há algo errado no código do projeto e gerar aprendizado.

QP2 - Os *bad smells* são tratados?

Os resultados mostraram que além de terem sido menos frequentes a identificação de *bad smells* em projetos que utilizam IC, também houve um volume maior de tratamentos, em média, dos cinco tipos de *bad smells* quando comparado com o conjunto de projetos que não utilizam IC. Ou seja, 75,79% dos *bad smells* identificados foram tratados no conjunto de projetos que utilizam IC e 60,48% nos projetos que não utilizam IC.

Para enriquecer esses resultados foi incluída a análise do tratamento considerando antes e depois da adoção de IC nos quatro projetos que utilizam IC. Após a adesão de IC, os projetos *Commons digester* e *Repairnator* tiveram aumento no volume de tratamento de todos os *bad smells*. Por outro lado, nos projetos *Jgnash* e *JUnit4*, parte dos *bad smells* que foram identificados tiveram seus tratamentos antes da adoção de IC, após não houveram nenhum tratamento.

Em relação ao número de tratamentos, a IC pode ser benéfica, pois a partir do alerta aos desenvolvedores de que há a ocorrência de *bad smells* sendo inseridos no código, seria mais fácil começar a adotar medidas para eliminar o *bad smell*. Ademais, verificaria se os desenvolvedores se preocupam com a questão da qualidade do *software* e a remoção de

bad smell para prevenir a degradação do *software*. Os resultados mostraram que apesar de terem sido identificados mais *bad smells* depois a IC ser aderida, também houve aumento no volume de tratamento e quanto maior é esse volume menos inconsistências de *design* de *software* permanecem no código. Nos projetos que adotaram a IC num estágio mais maduro de desenvolvimento mostrou que não houve nenhum tratamento após a adoção de IC isso por nenhum *bad smell* ter sido introduzido depois da IC e boa parte dos *bad smells* foram tratados antes.

QP3 - Quanto tempo os *bad smells* persistem no histórico do projeto de *software* até serem tratados?

Os resultados da QP2 mostraram que a porcentagem de tratamento é maior em projetos que utilizam IC. Ou seja, neles o volume de *bad smells* tratados é maior. Entretanto, esta investigação considerando os *bad smells* tratados mostra que nos projetos que não utilizam IC, o tratamento são realizados em um intervalo de tempo menor. Ou seja, neles os *bad smells* tem um ciclo de vida menor. Isso se dá pelo fato de não ter sido considerado as durações de todos *bad smells* no *Voldemort*, do *Long Method* no *Vrapper* e do *Duplicated Code* no *Jgnash* que nunca foram tratados. Por outro lado, considerando que os *bad smells* não tratados estão no repositório até a data do último *commit* do projeto, os projetos que utilizam IC além de terem um volume de tratados maior, também tratariam os *bad smells* identificados em menos tempo.

Para enriquecer esses resultados, também foi realizada uma análise de quanto tempo os *bad smells* permanecem nos quatro projetos que adotaram IC considerando antes e depois da adesão de IC. Considerado a análise excluindo os não tratados, o resultado mostrou que três dos projetos obtiveram a diminuição do ciclo de vida dos *bad smells*. Entretanto, ao incluir os não tratados mostrou que apenas dois deles tiveram o êxito de reduzir o tempo que os *bad smells* ficam no código do projeto. Por outro lado, os resultados mostraram que apesar de nem todos esses projetos terem tido a redução do tempo dos *bad smells*, a duração em média obtida foi menor nos projetos após sua adoção de IC.

Em relação ao intervalo de tempo em que os *bad smells* são tratados. A IC seria vantajosa, pois ao dar *feedback* rápido, os *bad smells* ao serem identificados mais rápido

poderiam ter seu tratamento imediato ou pelo menos ser tratado de modo mais rápido, o que pode explicar a redução do tempo que permanece no projeto. Esse fato pode ser explicado pelos resultados que mostraram uma redução no intervalo de tratamento tratamentos após a adoção de IC.

4.5 Ameaças à validade

Embora o estudo proposto tente minimizar as limitações da avaliação, alguns fatores podem ter influenciado nas conclusões descritas nesse documento. Nesta seção são descritos alguns dos aspectos identificados que podem ameaçar a validade deste estudo.

Uma das limitações é que as versões analisadas dos projetos foram percorridas em ordem cronológica, sem considerar a sequência de diferentes ramos. Deste modo, um possível efeito colateral pode ocorrer se considerar um *commit* de um ramo e depois um *commit* de outro diferente ramo. Neste caso, é possível que um *bad smell* seja introduzido em apenas um ramo e caso os *commits* analisados estiverem em ramos distintos é possível que sejam identificadas várias identificações e tratamentos do mesmo *bad smell*.

Outra limitação está associada a utilização da ferramenta *PMD* que identifica os *bad smells* de uma maneira específica. Como cada ferramenta pode implementar suas regras diferentemente, os *bad smells* identificados neste estudo podem não ter sido identificados no ambiente de desenvolvimento original. Por exemplo, pode ser que o *PMD* identifique um *Long Method* que não é considerado por outra ferramenta pelo fato de existirem diversas interpretações do que é um *Long Method*.

O estudo de caso foi realizado com oito projetos de código aberto escritos em Java. Portanto, pode ser que os resultados não sejam totalmente generalizáveis para outros projetos desenvolvidos em outros paradigmas ou linguagens de programação. O mesmo é válido para os diferentes tipos de *bad smells* descritos na literatura, pois este estudo explora apenas cinco tipos de *bad smells*. Então, pode ser que para outros tipos *bad smells*, a análise tenha um comportamento diferente.

4.6 Considerações finais

Os resultados da QP1 mostraram que os *bad smells* são menos frequentes em projetos que utilizam IC. Na QP2 é mostrado que o volume de tratados foi maior no conjunto de projetos que utilizam IC. Por fim, a QP3 mostrou que se excluir os *bad smells* não tratados da análise, o ciclo de vida dos *bad smells* é menor no conjunto de projetos que não utilizam IC. Já se incluí-los na análise, o ciclo de vida dos *bad smells* é menor no conjunto de projetos que utilizam IC.

Com base nos resultados das questões QP2 e QP3, foi possível ter concepções diferentes sobre o tratamento do *bad smell*, pois alguns projetos possuem um volume de tratamento maior e outros projetos possuem um intervalo menor de tratamentos. Qualquer uma das duas soluções podem funcionar, dependendo do perfil do projeto e de como as equipes vão tratar os *bad smells*. Ou seja, alguns projetos que priorizaram o volume maior de tratamento e outros projetos priorizaram o menor intervalo de tratamento. Por outro lado, como visto anteriormente nas práticas de IC, a IC visa dar o *feedback* rápido que possibilitaria o tratamento mais rápido. Portanto, os projetos que trataram os *bad smells* mais rápido estão mais alinhados aos princípios da IC.

5 Conclusões

Conforme a literatura consultada, não existe trabalho que relaciona a IC com a identificação e tratamento de *bad smells*. Portanto, este estudo desenvolve uma abordagem implementada em Java utilizando *PMD* e *GitHub* que investiga o ciclo de vida de *bad smells* em projetos de *software* para analisar se a IC possui influência na identificação e tratamento de *bad smells*. Como estudo de caso, a abordagem foi aplicada em oito projetos de *software*, explorando cinco tipos de *bad smells* detectados pelo *PMD* para responder às três questões de pesquisa sobre o ciclo de vida desses cinco *bad smells*.

A questão QP1 foi respondida a partir da análise da frequência da identificação de *bad smells* que mostraram que os *bad smells* são introduzidos com mais frequência em projetos que não utilizam IC. Na questão QP2 foi feita uma análise considerando os tratamentos desses *bad smells* mostrando que além da frequência da introdução de *bad smells* ser menor em projeto que utilizam IC, o volume de tratados neles é maior. Por fim, a questão QP3 foi respondida analisando o ciclo de vida, observado o intervalo de tempo em que esses *bad smells* demoraram para serem tratados. Essa análise mostrou duas perspectivas diferentes, uma que apesar terem mais *bad smells* em projetos que não utilizam IC, o ciclo de vida deles é menor. Ou seja, neles os *bad smells* permanecem por menos tempo. Isso ocorreu devido à exclusão dos *bad smells* sem tratamento da análise. Por outro lado, quando se analisa incluindo os *bad smells* não tratados, o resultado mostra que os *bad smells* também permanecem por menos tempo em projetos que utiliza IC.

Portanto, este estudo demonstra que a IC teve impacto positivo nos projetos que a utilizaram, pois ao proporcionar *feedback* rápido possibilita que após a identificação, os *bad smells* sejam eliminados em maior quantidade e de maneira mais rápida e eficiente pelos desenvolvedores. Logo, ela ajuda a impedir que o *bad smell* seja uma das causas da degradação do *software*.

Contribuições. Este estudo traz as seguintes contribuições:

- Uma ferramenta para analisar diversas versões de sistemas (Java) armazenados em repositórios *Git* para encontrar os cinco tipos de *bad smells*: *God Class, Data Class*

Long Method, Long Parameter List e *Duplicated Code.*

- Resultados do experimento. O estudo de oito projetos Java de código aberto mostra que a IC possui uma influência positiva na identificação e tratamento de *bad smells* quando utilizada.

5.1 Trabalhos futuros

Para consolidar e validar as afirmações realizadas neste estudo, serão necessárias mais pesquisas usando variáveis diferentes. Então, por questão do pouco tempo para realizar os experimentos, um possível trabalho futuro é utilizar mais repositórios de diferentes categorias de sistemas de *software*. Essa prática pode aumentar o poder dos resultados, possibilitando uma generalização para um grupo de sistemas. Outra possibilidade de trabalho futuro consiste na extensão da implementação para outras ferramentas de detecção de *bad smells* pelo fato do *PMD* detectar *bad smells* ter uma implementação específica e um conjunto limitado de *bad smells* que podem ser capturados. Deste modo, outras ferramentas podem dar uma perspectiva de detecção de *bad smells* diferente para a pesquisa. Outro trabalho possível é a extensão para outros tipos de *bad smells* além dos cinco utilizados para ver se a IC tem influência na identificação e tratamento de outros tipos de *bad smells*. Outra possibilidade de trabalho é estender para outras linguagens de programação, como *JavaScript* e *Python*, para verificar se os resultados podem ser generalizados. Finalmente, outro trabalho é realizar consulta aos desenvolvedores dos sistemas para verificar se as conclusões apresentadas foram realmente influenciadas pela IC ou por outros fatores que não foram observados nesse estudo.

Bibliografia

ABARBANELL, T. *Deploying from Travis CI to dokku*. 2017. [Online; accessed November 06, 2020]. Disponível em: ⟨https://blog.abarbanell.ch/linux/2017/09/09/deploy-from-travis-to-dokku/⟩.

AVERSANO, L.; CARPENITO, U.; IAMMARINO, M. An empirical study on the evolution of design smells. *Information*, Multidisciplinary Digital Publishing Institute, v. 11, n. 7, p. 348, 2020.

BAR, M.; FOGEL, K. *Open Software Development with CVS*. [S.l.]: Paraglyph Press, Scottsdale (AZ),, 2003.

BARRETO, A. S.; MURTA, L. G. P.; ROCHA, A. R. Uma abordagem para definição de processos baseada em reutilização visando à alta maturidade em processos. In: SBC. *Anais do XI Simpósio Brasileiro de Qualidade de Software*. [S.l.], 2012. p. 429–443.

BERCZUK, S. Pragmatic software configuration management. *IEEE Software*, IEEE, v. 20, n. 2, p. 15–17, 2003.

BIEMAN, J. M.; KANG, B.-K. Cohesion and reuse in an object-oriented system. In: *Proceedings of the 1995 Symposium on Software Reusability*. New York, NY, USA: Association for Computing Machinery, 1995. (SSR '95), p. 259–262. ISBN 0897917391. Disponível em: ⟨https://doi.org/10.1145/211782.211856⟩.

CHACON, S. About git. *línea]. Available: https://git-scm.com/[Último acceso: 06 11 2020]*, 2009.

CHATZIGEORGIOU, A.; MANAKOS, A. Investigating the evolution of bad smells in object-oriented code. In: IEEE. *2010 Seventh International Conference on the Quality of Information and Communications Technology*. [S.l.], 2010. p. 106–115.

DASKALANTONAKIS, M. K. A practical view of software measurement and implementation experiences within motorola. *IEEE Transactions on Software Engineering*, IEEE Computer Society, v. 18, n. 11, p. 998, 1992.

DIAS, A. F. *Conceitos Básicos de Controle de Versão de Software — Centralizado e Distribuído*. 2016. [Online; accessed February 07, 2021]. Disponível em: ⟨https://blog.pronus.io/posts/controle-de-versao/conceitos-basicos-de-controle-de-versao-de-software-centralizado-e-distribuido/⟩.

DUVALL, P. M.; MATYAS, S.; GLOVER, A. *Continuous integration: improving software quality and reducing risk*. [S.l.]: Pearson Education, 2007.

FITZPATRICK, B. W.; PILATO, C. M.; COLLINS-SUSSMAN, B. *Version control with Subversion*. [S.l.]: O'Reilly Media, 2004.

FONTANA, F. A. et al. Code smell detection: Towards a machine learning-based approach. In: IEEE. *2013 IEEE International Conference on Software Maintenance*. [S.l.], 2013. p. 396–399.

FOWLER, M. *Refactoring: improving the design of existing code*. [S.l.]: Addison-Wesley Professional, 1999.

FOWLER, M. *Continuous Integration*. 2010. [Online; accessed 11-Outubro-2020]. Disponível em: ⟨http://martinfowler.com/articles/continuousIntegration.html⟩.

FU, S.; SHEN, B. Code bad smell detection through evolutionary data mining. In: IEEE. *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.], 2015. p. 1–9.

HASSAN, A. E. The road ahead for mining software repositories. In: IEEE. *2008 Frontiers of Software Maintenance*. [S.l.], 2008. p. 48–57.

KAGDI, H.; COLLARD, M. L.; MALETIC, J. I. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of software maintenance and evolution: Research and practice*, Wiley Online Library, v. 19, n. 2, p. 77–131, 2007.

KEELE, S. et al. *Guidelines for performing systematic literature reviews in software engineering*. [S.l.], 2007.

KHOMH, F.; PENTA, M. D.; GUEHENEUC, Y.-G. An exploratory study of the impact of code smells on software change-proneness. In: IEEE. *2009 16th Working Conference on Reverse Engineering*. [S.l.], 2009. p. 75–84.

LANZA, M.; MARINESCU, R. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. [S.l.]: Springer Science & Business Media, 2006.

LEHMAN, M. M.; RAMIL, J. F. Towards a theory of software evolution-and its practical impact. In: IEEE COMPUTER SOCIETY. *Principles of Software Evolution, International Symposium on*. [S.l.], 2000. p. 2–2.

LENHARD, J. et al. Are code smell detection tools suitable for detecting architecture degradation? In: *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*. [S.l.: s.n.], 2017. p. 138–144.

LINARES-VÁSQUEZ, M. et al. Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. In: *Proceedings of the 22nd International Conference on Program Comprehension*. [S.l.: s.n.], 2014. p. 232–243.

LUJAN, S. et al. A preliminary study on the adequacy of static analysis warnings with respect to code smell prediction. In: *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*. [S.l.: s.n.], 2020. p. 1–6.

MAHMOOD, N. et al. Mining software repository for cleaning bugs using data mining technique. *CMC-COMPUTERS MATERIALS & CONTINUA*, TECH SCIENCE PRESS 871 CORONADO CENTER DR, SUTE 200, HENDERSON, NV 89052 USA, v. 69, n. 1, p. 873–893, 2021.

MANDELIN, D. et al. Jungloid mining: helping to navigate the api jungle. *ACM Sigplan Notices*, ACM New York, NY, USA, v. 40, n. 6, p. 48–61, 2005.

MARINESCU, C. et al. iplasma: An integrated platform for quality assessment of object-oriented design. In: *In ICSM (Industrial and Tool Volume*. [S.l.]: Society Press, 2005. p. 77–80.

MENEELY, A.; WILLIAMS, L. Strengthening the empirical analysis of the relationship between linus' law and software security. In: *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. [S.l.: s.n.], 2010. p. 1–10.

MICHAIL, A.; XIE, T. Helping users avoid bugs in gui applications. In: IEEE. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* [S.l.], 2005. p. 107–116.

MOCKUS, A.; ZHANG, P.; LI, P. L. Predictors of customer perceived software quality. In: IEEE. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* [S.l.], 2005. p. 225–233.

O'SULLIVAN, B. *Mercurial: The Definitive Guide: The Definitive Guide*. [S.l.]: "O'Reilly Media, Inc.", 2009.

PALOMBA, F. et al. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, IEEE, v. 41, n. 5, p. 462–489, 2014.

PALOMBA, F. et al. Landfill: An open dataset of code smells with public evaluation. In: IEEE. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. [S.l.], 2015. p. 482–485.

PARNAS, D. L. Software aging. In: IEEE. *Proceedings of 16th International Conference on Software Engineering*. [S.l.], 1994. p. 279–287.

PETERS, R.; ZAIDMAN, A. Evaluating the lifespan of code smells using software repository mining. In: IEEE. *2012 16th European Conference on Software Maintenance and Reengineering*. [S.l.], 2012. p. 411–416.

PETERSEN, K. et al. Systematic mapping studies in software engineering. In: *12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12*. [S.l.: s.n.], 2008. p. 1–10.

PMD. *Data Class*. 2021. ⟨https://pmd.github.io/latest/pmd_rules_java_design.html#dataclass⟩. Accessed: 2021-04-20.

PMD. *Excessive Method Length*. 2021. ⟨https://pmd.github.io/latest/pmd_rules_java_design.html#excessivemethodlength⟩. Accessed: 2021-04-20.

PMD. *Excessive Parameter List*. 2021. ⟨https://pmd.github.io/latest/pmd_rules_java_design.html#excessiveparameterlist⟩. Accessed: 2021-04-20.

RAO, A. A.; REDDY, K. N. Detecting bad smells in object oriented design using design change propagation probability matrix 1. Citeseer, 2007.

RAYMOND, E.; YOUNG, B. *The cathedral and the bazaar-musings on Linux and open source by an accidental revoltionary,(rev. ed.)*. [S.l.]: O'reilly, 2001.

ROBLES, G. et al. Mining large software compilations over time: another perspective of software evolution. In: *Proceedings of the 2006 international workshop on Mining software repositories*. [S.l.: s.n.], 2006. p. 3–9.

SINGH, R.; BINDAL, A.; KUMAR, A. Long method and long parameter list code smells detection using functional and semantic characteristics. *International Journal of Recent Technology and Engineering (IJRTE)*, v. 8, n. 6, p. 245–253, 2020.

SINGH, S. *Here's all one should know about God Class in Java*. 2020. [Online; accessed 10-Novembro-2021]. Disponível em: ⟨https://herownhelloworld.medium.com/heres-all-one-should-know-about-god-class-in-java-e318acbb9717⟩.

TSANTALIS, N.; CHAIKALIS, T.; CHATZIGEORGIOU, A. Jdeodorant: Identification and removal of type-checking bad smells. In: IEEE. *2008 12th European Conference on Software Maintenance and Reengineering*. [S.l.], 2008. p. 329–331.

TUFANO, M. et al. When and why your code starts to smell bad. In: IEEE. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. [S.l.], 2015. v. 1, p. 403–414.

TUFANO, M. et al. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, IEEE, v. 43, n. 11, p. 1063–1088, 2017.

VIDAL, S. et al. Jspirit: a flexible tool for the analysis of code smells. In: IEEE. *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*. [S.l.], 2015. p. 1–6.

WOMAKERSCODE. [Tutorial Git] git branch: O que são branches (ramos) no Git? *DEV Community*, DEV Community, Apr 2021. Disponível em: ⟨https://dev.to/womakerscode/tutorial-git-o-que-sao-branches-ramos-no-git-57pn⟩.

WOMAKERSCODE. *[Tutorial Git] git branch: O que são branches (ramos) no Git?* 2021. [Online; accessed 05-Janeiro-2022]. Disponível em: ⟨https://dev.to/womakerscode/tutorial-git-o-que-sao-branches-ramos-no-git-57pn⟩.

XIE, T.; PEI, J. Mapo: Mining api usages from open source repositories. In: *Proceedings of the 2006 international workshop on Mining software repositories*. [S.l.: s.n.], 2006. p. 54–57.

ZOLKIFLI, N. N.; NGAH, A.; DERAMAN, A. Version control system: A review. *Procedia Computer Science*, Elsevier, v. 135, p. 408–415, 2018.