Um estudo quantitativo sobre o uso de herança e interface em sistemas Java

Carlos Eduardo de Carvalho Dantas

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2017

Carlos Eduardo de Carvalho Dantas

Um estudo quantitativo sobre o uso de herança e interface em sistemas Java

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Marcelo de Almeida Maia

Uberlândia
2017

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

D192
2017

Dantas, Carlos Eduardo de Carvalho, 1982-
Um estudo quantitativo sobre o uso de herança e interface em sistemas Java / Carlos Eduardo de Carvalho Dantas. - 2017.
94 f. : il.

Orientador: Marcelo de Almeida Maia.
Dissertação (mestrado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.
Inclui bibliografia.

1. Computação - Teses. 2. Interface de programação de aplicações - Teses. 3. Java (Linguagem de programação de computador) - Teses. 4. Engenharia de Software - Teses. I. Maia, Marcelo de Almeida. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada "**Um estudo quantitativo sobre o uso de herança e interface em sistemas Java**" por **Carlos Eduardo de Carvalho Dantas** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 10 de fevereiro de 2017

Orientador: _____
Prof. Dr. Marcelo de Almeida Maia
Universidade Federal de Uberlândia

Banca Examinadora:

_____
Prof. Dr. Autran Macedo
Universidade Federal de Uberlândia

_____
Prof. Dr. Cláudio Nogueira Sant'Anna
Universidade Federal da Bahia

Este trabalho é dedicado à minha esposa Amanda e ao meu filho Eduardo Miguel.

Agradecimentos

Agradeço primeiramente à Deus por guiar meus passos.

Ao meu orientador Professor Doutor Marcelo de Almeida Maia, pelos ensinamentos, profissionalismo e paciência em lidar com as minhas limitações, me possibilitando ir até o fim.

À minha amada esposa Amanda pela compreensão e companheirismo no dia-a-dia.

Ao meu filho Eduardo Miguel por me inspirar a persistir nos momentos mais difíceis.

À minha mãe Jaida pelo apoio demonstrado.

Aos Professores Doutores Autran e Cláudio que aceitaram o convite para compor a banca de avaliação.

Aos Professores do PPGCO, especialmente os que contribuíram diretamente com a minha formação.

Aos colegas do LASCAM por compartilharem seus conhecimentos e experiências adquiridas, fornecendo dicas que contribuíram para esta dissertação.

Ao pesquisador Fábio Palomba, que gentilmente forneceu a ferramenta Code Smell Analyser, permitindo a extração de métricas e Code Smells.

Aos meus amigos e colegas do IFTM, que direta ou indiretamente exerceram algum tipo de contribuição nesta dissertação.

"Nós somos o que repetidamente fazemos. Excelência, então, não é um ato, mas um hábito."

(Aristóteles)

Resumo

O recurso de herança é uma das principais características do paradigma de Orientação a Objetos. Contudo, trabalhos anteriores recomendam cuidado quanto ao seu uso, sugerindo alternativas em Java como a adoção de composição com implementação de interfaces. Apesar de ser um tema bem estudado, ainda há pouco conhecimento se estas recomendações foram amplamente adotadas pelos desenvolvedores de maneira geral. Este trabalho possui como objetivo avaliar como os recursos de herança e interface têm sido empregados em Java, comparando sistemas mais recentes com antigos, e também versões de cada sistema em épocas distintas. Os indicadores avaliados foram a quantidade de mudanças corretivas dos sistemas, quebras de encapsulamento pelo uso do operador *instanceof*, medidas de coesão, acoplamento e ocorrências de *code smells*. Por fim, foi realizada uma medição sobre o quão frequente os recursos de herança ou interface são adicionados ou removidos das classes, e as motivações pelas quais os desenvolvedores executam estes procedimentos. Foram analisados 1.656 sistemas *open-source* construídos entre 1997 e 2013, todos hospedados nos repositórios *GitHub* e *SourceForge*. Foi constatado que os desenvolvedores ainda utilizam herança primariamente como um recurso para reaproveitamento de código, motivados pela necessidade de evitar duplicidade de código-fonte. Em projetos mais recentes, as classes na hierarquia de herança apresentaram menos mudanças corretivas e as subclasses fizeram menos uso do operador *instanceof*. No entanto, à medida que evoluem, as classes na hierarquia de herança tendem a se tornar complexas na medida em que as mudanças ocorrem. As classes que implementam interfaces mostraram pouca relação com as suas respectivas interfaces implementadas, e foram observados indícios de que este recurso ainda é subutilizado. Estes resultados mostram que ainda existe alguma falta de conhecimento sobre o uso de práticas adequadas orientadas a objetos, o que reforça a necessidade de formação de desenvolvedores sobre como projetar melhores classes.

Palavras-chave: Herança. Interfaces. Mudanças corretivas. Code Smells. Encapsulamento. Coesão. Acoplamento. GitHub. SourceForge.

Abstract

Inheritance is one of the main features in the object-oriented paradigm (OOP). Nonetheless, previous work recommend carefully using it, suggesting alternatives in Java such as the adoption of composition with implementation of interfaces. Despite of being a well-studied theme, there is still little knowledge if such recommendations have been widely adopted by developers in general. This work aims at evaluating how the inheritance and interface resources have been used in Java, comparing new projects with older ones, and also the different releases of projects. Evaluated indicators were the number of corrective changes of the projects, encapsulation breaks by the use of `instanceof` operator, measures of cohesion, coupling and occurrence of code smells. We also studied how often inheritance or interface features are added or removed from classes, and motivations by which developers perform such procedure. A total of 1,656 open-source projects built between 1997 and 2013, all hosted in the repositories GitHub and SourceForge, were analyzed. We observed developers still use inheritance primarily as a resource for code reuse, motivated by the need to avoid duplicity of source code. In newer projects, classes in inheritance hierarchy had fewer corrective changes and subclasses had fewer use of the `instanceof` operator. However, as they evolve, classes in inheritance hierarchy tend to become complex as changes occur. Classes implementing interfaces have shown little relation to the interfaces, and there is indication that they are still underutilized. These results show there is still some lack of knowledge about the use of adequate object-oriented practices, which reinforces the need for training developers on how to design better classes.

Keywords: Inheritance. Interfaces. Corrective commits. Code Smells. Encapsulation. Cohesion. Coupling. GitHub. SourceForge..

Lista de ilustrações

Lista de tabelas

Lista de siglas

API Application Programming Interface

CBO Coupling Between Objects

CVS Concurrent Version System

DER Diagrama Entidade Relacionamento

DIP Dependency Inversion Principle

DIT Depth of Inheritance

ELOC Effective Lines of Code

ISP Interface Segregation Principle

LCOM Lack of Cohesion of Methods

LSP Liskov Substitutive Principle

NBR Negative Binomial Regression

NOA Number of Attributes

NOC Number of Children

NOPA Number of Public Attributes

NOM Number of Methods

OCP Open/Closed Principle

POO Paradigma de Orientação a Objetos

RFC Response for a Class

SRP Single Responsability Principle

SVN Subversion

WMC Weighted Methods Per Class

Sumário

Capítulo 1

Introdução

Desde o surgimento do Paradigma de Orientação a Objetos (POO), o recurso de herança têm obtido destaque, principalmente pelos benefícios do reaproveitamento de código-fonte, e a possibilidade de projetar sistemas flexíveis usando polimorfismo (MEYER, 1989). Além disso, diversos *design patterns (factory method, template method*, dentre outros) foram propostos, utilizando recursos de herança (GAMMA et al., 1995). Isto motivou que, linguagens de programação e *frameworks* projetassem suas funcionalidades utilizando este recurso. Como exemplos, os projetistas da sintaxe da linguagem Java construíram hierarquias de herança em classes como *java.util.Hashtable*, *java.util.Vector*, *java.util.Properties* e *java.util.Stack* [1]. Além da linguagem, a *API* de *Servlets* foi implementada sobre hierarquias de herança com suas classes *javax.servlet.GenericServlet*, *javax.servlet.HttpServlet* [2], dentre outras. E por fim, *frameworks* como *Struts* [3], que foram projetados para o desenvolvedor obter as suas funcionalidades estendendo uma classe chave, como *org.apache.struts.Action*.

Embora o recurso de herança tenha significativa visibilidade no ensino de *POO* (BUDD, 1991), diversos trabalhos têm recomendado cuidado quanto ao seu uso, principalmente pela facilidade que este possui em quebrar o encapsulamento das classes (SNYDER, 1986). Alguns trabalhos sugerem que em Java, o uso de composição com implementação de interfaces seja priorizado em detrimento da herança (BLOCH, 2008). A quebra de encapsulamento afeta diretamente aspectos como manutenibilidade e compreensibilidade (ABBES et al., 2011) dos sistemas. Tal quebra ocorre principalmente porque subclasses dependem dos detalhes de implementação de sua superclasse para funcionar adequadamente. Com isso, é necessário conhecer minuciosamente o comportamento da superclasse para implementar subclasses adequadas. Além disso, tais classes poderão sofrer modificações ao longo do tempo, consequentemente as futuras alterações nas superclasses poderão produzir efeitos colaterais nas subclasses, e vice-versa (GAMMA et al., 1995). Outros trabalhos

[1] https://docs.oracle.com/javase/7/docs/api/java/util/Properties.html
[2] http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/package-summary.html
[3] https://struts.apache.org/

sugerem que a herança nunca seja utilizada (HOLUB, 2003), ou que pelo menos não se construam hierarquias de herança complexas, com elevada profundidade (DALY et al., 1996).

Alguns dos efeitos negativos citados foram observados inclusive na própria *Application Programming Interface (API)* de Java. Para obter maior flexibilidade com menor acoplamento entre as classes, os desenvolvedores da linguagem construíram novas *APIs* e *frameworks*, utilizando diferentes níveis de abstrações com interfaces. Um exemplo é o *framework Collections* [4], que possui principalmente as interfaces *Collections, Set, Queue* e *List*, todas do pacote *java.util*. Utilizando estas interfaces, é possível obter um menor acoplamento entre as classes, já que quando uma classe efetua composição com a interface *Collections*, diversos tipos de coleções poderão ser instanciadas para executar os métodos definidos pela interface. Com isso, as classes *java.util.Vector* e *java.util.Stack* citadas anteriormente foram modificadas para serem incluídas no *framework Collections*, pois o *design* destas classes não estava alinhado com a meta de propiciar aos desenvolvedores a possibilidade de escrever sistemas com menor acoplamento e maior flexibilidade. Além do *framework* citado, diversas outras novas *APIs* foram projetadas usando interfaces, como por exemplo *JDBC* [5], *JPA* [6], *java.util.Comparator*, dentre outras.

Desta forma, assim como a própria *API* de Java sofreu modificações em relação a aplicação de herança e interfaces em seu projeto, pode ser que exista uma influência nos desenvolvedores de uma maneira geral em direção a este movimento. Com esta constatação, o ensino do paradigma *POO* poderá ser direcionado para estudos de caso práticos sobre como estes recursos têm sido efetivamente aplicados.

1.1 Problema

Diante do cenário apresentado, não existe um conhecimento quantitativo de como os recursos de herança e interface têm sido utilizados ao longo do tempo, considerando uma comunidade representativa de desenvolvedores. A plataforma Java existe há mais de 20 anos, e desde esta época, diversos trabalhos foram publicados, alertando sobre eventuais quebras de encapsulamento das classes com o uso de herança. Ao mesmo tempo, outros trabalhos têm mostrado que a herança têm sido empregada consistentemente em Java (TEMPERO; NOBLE; MELTON, 2008). Desta forma, faz sentido investigar se o recurso de herança vem sendo empregado de forma diferente nos sistemas mais recentes, e se de fato, menos *débito técnico* estaria sendo introduzido(SURYANARAYANA; SAMARTHYAM; SHARMA, 2015). E também verificar se existe alguma tendência na substituição deste recurso por práticas recomendadas como composição com interfaces. Com isso, as análises a serem efetuadas em classes que utilizam herança podem ser replicadas

[4] https://docs.oracle.com/javase/7/docs/technotes/guides/collections/overview.html
[5] https://jcp.org/en/jsr/detail?id=221
[6] https://jcp.org/en/jsr/detail?id=317

para as classes que implementam interfaces, e seus resultados poderiam ser comparados, verificando se houve alguma mudança significativa no uso destes recursos ao longo do tempo.

Outra motivação para este estudo envolve a mudança que têm ocorrido na metodologia empregada no desenvolvimento de sistemas. Os requisitos solicitados estão cada vez mais complexos e com maior urgência na disponibilização em ambiente de produção. Isso produz novas práticas quanto à metodologia de desenvolvimento, como o uso de testes automatizados, entrega contínua, dentre outros (CHEN, 2016). Com isso, as abstrações construídas para atender os requisitos precisam possuir um propósito bem definido, facilitando sua expansão para novas funcionalidades a serem solicitadas no futuro, e que não estejam previstas no projeto inicial do sistema (MENS; TOURWÉ, 2004). E como a herança exige um projeto rígido do desenvolvedor quanto ao comportamento das superclasses e subclasses envolvidas, uma investigação se faz necessária sobre quando as classes adquirem ou removem herança e interface. Com isso, serão observadas as circunstâncias e motivações pelas quais estes eventos ocorrem.

Para avaliar se o uso de herança vem sendo empregado de maneira diferente, diversos critérios podem ser estabelecidos. Alguns destes se referem principalmente à adoção de más práticas, como *Antipatterns* e *Code Smells*. Por exemplo, subclasses que são verificadas por condições *instanceof* geralmente representam quebra de encapsulamento, pois tal verificação é realizada em razão de alguma funcionalidade exclusiva da subclasse, e que não pode ser representada por sua superclasse ou interface. Além das más práticas, pode-se avaliar se os desenvolvedores conseguem manter estas classes em níveis adequados de coesão e acoplamento.

1.2 Objetivos e Contribuições

Diante do problema apresentado, o objetivo geral deste trabalho consiste em efetuar uma análise quantitativa sobre o uso de herança e interface em sistemas Java. Esta análise se restringe a classes que estendem superclasses ou implementam interfaces internas de cada sistema. Desta forma, serão descartadas heranças que possuem superclasses ou interfaces externas às classes do sistema, como por exemplo, superclasse *javax.swing.JFrame* ou interface *java.lang.Runnable*. Também serão analisados sistemas que foram construídos em épocas distintas, visando efetuar uma análise sobre alguns de seus indicadores, verificando se a idade do sistema exerce alguma influência sobre a variação destes. Os indicadores empregados para análise serão destacados nos objetivos específicos a seguir:

a) Avaliar o uso dos recursos de herança e implementação de interfaces em sistemas de épocas distintas.

b) Investigar o uso do operador *instanceof* em subclasses com herança e que implementam interfaces.

c) Avaliar a intensidade de alterações corretivas em classes com herança e que implementam interfaces.

d) Avaliar os indicadores de coesão e acoplamento para classes com herança e que implementam interfaces.

e) Descobrir *Code Smells* que surgem mais frequentemente em classes com herança e que implementam interfaces.

f) Medir a frequência e descobrir as motivações pelos quais os recursos de herança e implementação de interface são adicionados ou removidos dos sistemas.

Diante dos objetivos citados, a principal contribuição deste trabalho consiste em oferecer uma visão atualizada, sobre como os recursos de herança e implementação de interfaces têm sido utilizados pelos desenvolvedores em sistemas *open source*. Com isso, novos treinamentos podem ser oferecidos focando em pontos específicos onde ocorrem maior incidência de erros quanto ao uso destes recursos.

Este trabalho não possui como escopo avaliar se é recomendável usar herança ou composição com interface. A intenção é avaliar como vem sendo empregado o seu uso com base nos indicadores citados nos objetivos específicos, e analisar se a época de criação do sistema exerce alguma influência sobre as suas respectivas variações.

1.3 Estrutura da Dissertação

O restante deste trabalho está estruturado nos seguintes capítulos:

a) O Capítulo 2 irá introduzir conceitos sobre herança, boas e más práticas sobre o uso do paradigma *POO*. Também será destacado o o ferramental utilizado para a execução deste trabalho.

b) O Capítulo 3 irá descrever a metodologia empregada para responder às perguntas de pesquisa propostas neste trabalho. Desta forma, serão mostradas as etapas de coleta, modelagem e análise dos dados.

c) O Capítulo 4 irá apresentar e discutir os resultados para as perguntas de pesquisa formuladas.

d) O Capítulo 5 irá destacar os principais trabalhos relacionados envolvendo o estudo de herança, detecção de *Code Smells*, e técnicas de regressão e análise temática utilizadas em trabalhos de Engenharia de Software.

e) O Capítulo 6 irá apresentar as conclusões e lições aprendidas com os resultados desta pesquisa.

Capítulo 2

Fundamentação Teórica

Este capítulo está dividido em duas partes: na primeira serão apresentados conceitos introdutórios referente à herança, interfaces, boas e más práticas quanto ao uso do paradigma *POO* e métricas estruturais. Na segunda parte, serão apresentadas as ferramentas utilizadas para atingir os objetivos propostos. Todos os indicadores citados na Seção 1.2 serão descritos neste capítulo com maiores detalhes.

2.1 Conceitos Introdutórios

O objetivo desta seção consiste em abordar conceitos que dizem respeito ao *design* de classes, evidenciando boas e más práticas, e como isso se relaciona com o uso dos recursos de herança e interface. Também serão descritas as métricas estruturais, destacando quais métricas representam os níveis de coesão e acoplamento das classes.

2.1.1 Herança e Implementação de Interfaces

A herança é um recurso em *POO*, onde um conjunto de classes podem ser usadas em diferentes níveis de abstração, mediante o conceito de polimorfismo. Desta forma, uma classe pode consumir funcionalidades de outra sem necessariamente ter declarado a sua instância. Além disso, as classes que estão no nível mais baixo da hierarquia (subclasses) podem herdar funcionalidades das que estão em níveis mais altos (superclasses). Este recurso foi projetado visando a reutilização de código-fonte, onde os comportamentos mais gerais se concentram nas superclasses, enquanto que os mais específicos nas subclasses. E com o polimorfismo, as superclasses podem ocultar a existência das subclasses para classes externas, tornando o sistema mais flexível e com menor acoplamento, pois ao adicionar uma nova subclasse, ocorrerá um incremento de funcionalidades, que poderão ser utilizados pelas classes externas sem necessidade de manutenção. (TAIVALSAARI, 1996). Com isso, uma hierarquia de herança consiste no conjunto de superclasses e subclasses que

juntas promovem a representação de um conjunto de funcionalidades úteis ao sistema [1]. A Figura 1 *(A)* mostra um diagrama de classe (RUMBAUGH; JACOBSON; BOOCH, 2004) ilustrando uma hierarquia de herança. É importante destacar que em Java, cada subclasse pode possuir apenas uma única superclasse, consequentemente não é permitido que uma mesma classe seja utilizada em duas hierarquias de herança. E por fim, uma variação da superclasse ocorre quando esta é abstrata. Neste contexto, a superclasse não pode ser instanciada, e parte de suas definições precisam obrigatoriamente ser implementadas pelas subclasses. (BLOCH, 2008).

A implementação de interfaces é um recurso que existe apenas em um subconjunto de linguagens que suportam o paradigma *POO*, dentre os quais Java está inserido. Neste recurso, as interfaces funcionam como um contrato em que apenas assinaturas de métodos e valores de constantes são definidos. [2] Estas assinaturas caracterizam um comportamento, e as classes que se identificarem irão implementar a interface e fornecer o seu algoritmo para o comportamento. Com isso, toda a lógica do comportamento está concentrada nas classes que implementam as interfaces. Sendo assim, ao contrário da herança, não existe reaproveitamento de código em interfaces. Contudo, as interfaces oferecem os mesmos benefícios de polimorfismo que foram destacados para herança, com o adicional que classes podem implementar múltiplas interfaces, possibilitando ser usada polimorficamente em diferentes abstrações. A Figura 1 *(B)* ilustra um cenário com duas interfaces, evidenciando que a *Subclasse1* implementa apenas a *Interface1*, e a *Subclasse2* implementa *Interface1* e *Interface2*. E os comportamentos implementados pelas classes são consumidos pela *ClasseExterna*. [3].

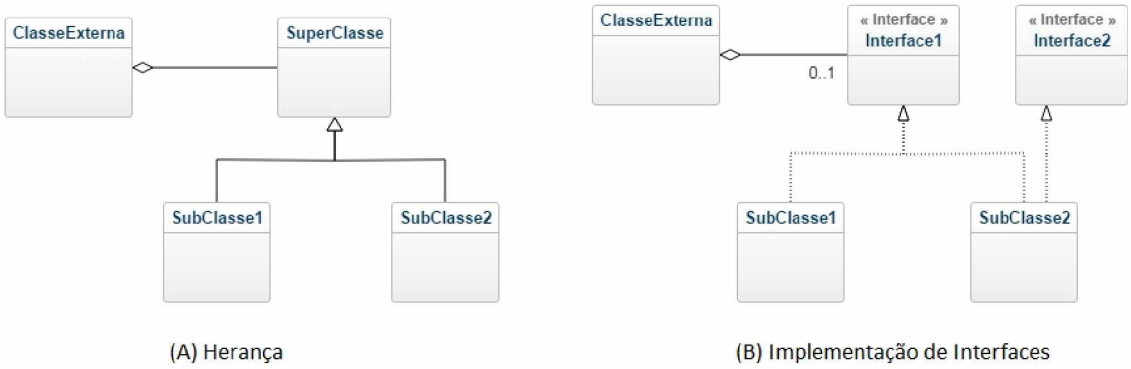


Figura 1 – Diagrama de Classe que ilustra os recursos de herança e implementação de interfaces

[1] No restante da dissertação, este termo será abreviado para *classes com herança*

[2] A partir da versão 8, a sintaxe da linguagem Java permite a definição de métodos *default* para interfaces. Contudo, esta versão foi lançada em 18/03/2014, data posterior à maioria das alterações realizadas nos sistemas analisados. Com isso, os métodos *default* serão desconsiderados.

[3] No restante da dissertação, este termo será abreviado para *classes com interface*

2.1.2 Princípios *SOLID*

Os princípios *SOLID* foram propostos por (MARTIN, 2003) e visam definir cinco princípios que formam as iniciais do termo citado. Combinando as práticas recomendadas por estes princípios, a tendência é que se construam sistemas *POO* mais fáceis de evoluir, manter, testar e reutilizar. Além disso, evita-se a adição de *débitos técnicos* (SURYANARAYANA; SAMARTHYAM; SHARMA, 2015) nos sistemas, como por exemplo os *Code Smells*. Segue a lista de cada princípio e sua respectiva definição.

a) *Single Responsability Principle (SRP)* - Classes precisam ser coesas, possuindo uma única responsabilidade. Isso irá gerar um equilíbrio de métodos e atributos para cada classe, não permitindo que cresçam de forma desproporcional, evitando adquirir funcionalidades que não estejam de acordo com o propósito da classe.

b) *Open/Closed Principle (OCP)* - Este princípio diz que classes precisam ser abertas para extensão e fechadas para modificação, ou seja, não podem ser modificadas com frequência. Ao violar este princípio, classes podem se tornar complexas ao longo das suas modificações, por acumularem diversas estruturas condicionais e caminhos distintos para executar novas funcionalidades. A consequência é que tais classes estarão propensas a sofrerem constantes manutenções. Para evitar que isso ocorra, é necessário construir abstrações, que podem ser representadas utilizando herança ou implementação de interfaces.

c) *Liskov Substitutive Principle (LSP)* - Este princípio prevê que as subclasses de herança precisam ser construídas atendendo critérios, para evitar conflitos com suas superclasses. Um critério consiste nos métodos das subclasses serem obrigados a estabelecer pré-condições de igual ou menor rigidez que os métodos das suas superclasses. E as pós-condições com igual ou maior rigidez. As subclasses também não podem lançar exceções que não são lançadas pelas superclasses. Estes critérios foram construídos para que superclasses possam representar o comportamento da herança diante das classes externas, sem que suas subclasses produzam efeitos colaterais inesperados (LISKOV, 1987).

d) *Interface Segregation Principle (ISP)* - Este princípio defende que interfaces sejam coesas, e que as assinaturas de seus métodos representem um único comportamento. A intenção é que as interfaces sejam estáveis, evitando que subclasses implementem algum método de forma indesejada.

e) *Dependency Inversion Principle (DIP)* - É inevitável que classes possuam dependências entre elas. Contudo, o acoplamento precisa ser o menor possível entre as classes envolvidas. O baixo acoplamento fará com que modificações em uma classe não sejam propagadas para suas dependências, facilitando a manutenção. Portanto, é preferencial que uma classe possua acoplamento apenas com interfaces ou superclasses, utilizando suas subclasses polimorficamente.

2.1.3 *Code Smells*

Para projetar classes eficientes, sistemas que adotam o paradigma *POO* deveriam seguir os princípios *SOLID* citados. Contudo, os desenvolvedores podem violar alguns desses princípios por vários motivos, seja pela urgência na entrega do sistema (LAVALLÉE; ROBILLARD, 2015), adoção de uma solução deficiente no projeto das classes do sistema, ou simplesmente pelo desconhecimento quanto aos princípios citados (PALOMBA et al., 2014). Estas violações resultam no surgimento de algumas anomalias nas classes desses sistemas. Várias dessas anomalias foram catalogadas por (FOWLER et al., 1999), e apelidadas de *Code Bad Smells* (ou *Code Smells*). Em linhas gerais, a presença de anomalias nas classes de um sistema pode apresentar danos como o decremento da compreensibilidade (ABBES et al., 2011) e manutenibilidade (YAMASHITA; MOONEN, 2012). Além disso, tais sistemas podem se tornar mais propensos a falhas, e sofrer mais mudanças no decorrer do tempo (KHOMH et al., 2012). Segue a lista dos *Code Smells* empregados neste trabalho:

a) *Class Data Should be Private* - Classes que expõe seus atributos, permitindo que o comportamento da classe seja manipulado externamente, quebrando o encapsulamento da classe. Este *Code Smell* afeta diretamente o princípio *DIP*.

b) *Complex Class* - Classes que são difíceis de testar e manter, por possuírem diversas estruturas condicionais e caminhos de execução distintos. Este *Code Smell* afeta diretamente o princípio *OCP*.

c) *Functional Decomposition* - Classes que declaram muitos atributos e implementam poucos métodos, fazendo pouco uso dos recursos de *POO*. Este *Code Smell* afeta diretamente o princípio *OCP*.

d) *God Class* - Classes que controlam muitos outros objetos do sistema. Geralmente possuem baixa coesão, pois acumulam diversas responsabilidades, e possuem diversas linhas de código-fonte. Este *Code Smell* afeta diretamente os princípios *SRP* e *OCP*.

e) *Lazy Class* - Classes que possuem poucas funcionalidades, não justificando a sua existência. Este *Code Smell* não afeta diretamente nenhum princípio *SOLID*, já que a classe infectada sequer deveria existir.

f) *Long Method* - Métodos que são desnecessariamente longos. Portanto, poderiam ser divididos, gerando outros métodos. Este *Code Smell* afeta diretamente o princípio *OCP*.

g) *Refused Parent Bequest* - Ocorre quando uma subclasse herda comportamentos indesejados das suas superclasses, ou necessitam implementar métodos também indesejados de interfaces ou classes abstratas. Isto ocorre pela falta de compatibilidade entre superclasse e subclasses, gerando uma série de restrições para determinadas

subclasses, e consequentemente, obrigando o desenvolvedor a conhecer sua arquitetura interna para utilizá-la. Este *Code Smell* afeta diretamente os princípios *ISP* e *LSP*.

h) *Spaghetti Code* - Classes que não representam nenhum tipo de comportamento para o sistema. Costumam possuir apenas alguns métodos longos sem parâmetros. Este *Code Smell* afeta diretamente os princípios *SRP* e *OCP*.

2.1.4 Sobre Herança versus Composição

Os princípios *SOLID* e *Code Smells* representam boas e más práticas que precisam ser observadas em todas as classes de um sistema. Para conseguir o êxito em respeitar tais princípios nas classes, existe um conjunto de práticas que são aconselhadas. Por exemplo, uma prática recomendada consiste em *Programar para Interface ou Superclasse e não para Implementação.* O objetivo é reduzir o acoplamento das classes, favorecendo a implementação do princípio *DIP* (MARTIN, 2003).

Dentre estas práticas recomendadas, uma das mais polêmicas consiste em *Evitar a Herança e Favorecer a Composição* (GAMMA et al., 1995). O objetivo desta recomendação consiste em minimizar possíveis quebras de encapsulamento das classes. Na prática, toda vez em que se faz necessário conhecer o funcionamento interno de uma classe para consumi-la, obtém-se uma quebra de encapsulamento (SNYDER, 1986). Contudo, nem todas as quebras de encapsulamento ocorrem em virtude do mau uso de herança. Por exemplo, uma classe que permite a manipulação direta dos seus atributos por outras classes está afetada pela quebra de encapsulamento. Outro cenário ocorre quando uma classe expõe parte de sua lógica interna para suas dependências. Neste caso, o desenvolvedor precisará ter conhecimento prévio que, para um método da classe funcionar de maneira adequada, será necessário também executar outros métodos da mesma classe, fornecendo parâmetros específicos.

Apesar da quebra de encapsulamento ser um problema geral entre as classes de um sistema, a prática de evitar herança é recomendada porque o seu mau uso pode intensificar as quebras de encapsulamento, como por exemplo quando uma classe é infectada com o *Code Smell Refused Parent Bequest.* Em função destas quebras, todos os princípios *SOLID* podem ser violados. O princípio *DIP* pode ser violado pois as classes externas podem ser obrigadas a se acoplarem com as subclasses, em virtude de possíveis exclusividades no seu comportamento. O princípio *OCP* pode ser violado pois comandos condicionais utilizando o operador *instanceof* podem ser utilizados, para observar as restrições de cada subclasse, gerando uma série de estruturas condicionais, tornando a classe mais complexa (HAMMER; SCHAADE; SNELTING, 2008). O princípio *LSP* pode ser violado pois uma subclasse com restrições exclusivas não poderá mais ser representada por sua superclasse. O mesmo ocorrerá com o princípio *ISP*, caso a classe implemente um método definido

pela interface que não representa o seu comportamento. E por fim, o princípio *SRP* pode ser violado, pois ao adicionar comportamentos de forma descontrolada nas superclasses, a tendência é que esta possua mais responsabilidades do que deveria, perdendo coesão.

Diante do que foi abordado, existe a recomendação de se utilizar composição porque a subclasse poderá escolher quais funcionalidades irá consumir de outra classe. Com a herança não existe essa possibilidade, já que todo o comportamento da superclasse será estendido para suas subclasses. E o uso de interfaces é recomendado para possibilitar que seja ocultada qual subclasse forneceu a implementação de determinada funcionalidade. Isto fará com que o desenvolvedor não precise conhecer o comportamento das classes que implementam as interfaces. Além disso, as classes que dependem destas funcionalidades irão realizar chamadas desconhecendo o funcionamento da classe que implementa a interface, reduzindo o acoplamento. Com isso, a classe dependente irá se concentrar apenas em manutenções referentes ao seu propósito, favorecendo o princípio *OCP* (BLOCH, 2008). A Subseção 2.1.4.1 irá apresentar um exemplo, onde foi construída uma hierarquia de herança na sintaxe da linguagem Java visando obter reaproveitamento de código. Contudo esta hierarquia gerou uma quebra de encapsulamento nas classes envolvidas.

2.1.4.1 Quebra de encapsulamento e sugestão de *refactoring* na API de Java

Nesta seção será mostrado um exemplo onde ocorreu uma quebra de encapsulamento na classe *java.util.Properties* [4] da *API* de Java, em virtude de uma herança mal projetada. Esta classe foi construída para que sejam adicionados e recuperados pares de *strings* do tipo chave/valor. Por isso, foram criados os métodos *getProperty()* e *setProperty()*. Para armazenamento e recuperação das *Strings*, sugeriu-se utilizar uma *tabela hash*. Como a classe *java.util.Hashtable* [5] possui implementações prontas desta tabela, a classe *Properties* foi construída como subclasse de *HashTable*, visando reaproveitar seu código-fonte. Contudo, como a classe *Hashtable* foi projetada para permitir adicionar e recuperar chaves e valores de qualquer tipo com os métodos *get()* e *put()*, o princípio *LSP* foi violado, já que a subclasse está restringindo as pré-condições estabelecidas pela superclasse. Isso caracteriza uma quebra de encapsulamento.

A Figura 2 apresenta os efeitos negativos desta quebra de encapsulamento. Na linha 10, o método *put()* é executado diretamente na superclasse, adicionando a data corrente do sistema. Apenas essa linha já é suficiente para evidenciar a quebra de encapsulamento, pois a subclasse herdou um método da superclasse que viola as suas condições. Contudo, as linhas 11 e 12 mostram outra consequência negativa da quebra de encapsulamento. Na linha 11, o método *get()* é executado diretamente na superclasse, recuperando e imprimindo a data. E na linha 12, o método *getProperty()* é executado na subclasse, e o sistema imprime a resposta *null*. Este problema ocorre porque, para impor suas pré-condições,

[4] https://docs.oracle.com/javase/7/docs/api/java/util/Properties.html
[5] https://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html

```java
package br.ufu.facom.mestrado.util;

import java.util.Date;
import java.util.Properties;

public class PropertiesClient {

    public static void main(String[] args) {
        Properties p = new Properties();
        p.put("currentDate",new Date());
        System.out.println("Get "+p.get("currentDate"));
        System.out.println("GetProperty "+p.getProperty("currentDate"));
    }

}
```

Figura 2 – Classe PropertiesClient.java que faz uso da classe java.util.Properties

a subclasse precisa definir uma nova instância de *tabela hash*. Com isso, na prática a classe *Properties* oferece duas *tabelas hash*: uma que aceita apenas *strings*, onde o método *setProperty()* armazena e *getProperty()* recupera. E a segunda herdada da superclasse *Hashtable* que aceita qualquer tipo, onde o método *put()* armazena e *get()* recupera.

```java
package br.ufu.facom.mestrado.util;

import java.util.Hashtable;

public class Properties {
    private Hashtable<Object,Object> table = new Hashtable<Object, Object>();

    public void put(Object key, Object value) {
        table.put(key, value);
    }
    public void setProperty(String key, String value) {
        put(key, value);
    }
    public Object getProperty(Object key) {
        return table.get(key);
    }
    public Object get(Object key) {
        return getProperty(key);
    }
}
```

Figura 3 – Redefinição da classe java.util.Properties visando composição

O exemplo mostrado evidencia dois problemas: no primeiro, *Hashtable* dificilmente poderá substituir *Properties* polimorficamente. E no segundo, o desenvolvedor que for consumir esta classe precisará conhecer todas estas restrições para implementar suas funcionalidades sem efeitos colaterais. Ao perceber o erro, os projetistas da linguagem Java poderiam ter corrigido este defeito. Contudo, com a versão lançada em produção, diversos desenvolvedores já construíram sistemas utilizando estes métodos de *Properties*, logo a retirada do método poderia gerar quebras de compatibilidade entre versões. De qualquer forma, é possível efetuar uma operação de *refactoring* para evitar que ao menos o efeito

colateral de imprimir *null* não ocorra. A Figura 3 apresenta a proposta de *refactoring* para a classe *java.util.Properties*, seguindo o princípio de favorecer composição ao invés de herança (GAMMA et al., 1995). Desta forma, a classe *Properties* controla as funcionalidades que irá consumir, e ao mesmo tempo, quais métodos públicos irá expor às classes externas, mantendo seu comportamento interno encapsulado. Com as alterações apresentadas, o código-fonte mostrado em Figura 2 iria imprimir a data correta em ambos os casos.

2.1.4.2 Quebra de encapsulamento por uso inadequado de *instanceof*

O operador *instanceof* é uma palavra reservada na sintaxe da linguagem *Java*, sendo utilizado quando se faz necessário verificar qual é a real instância da variável em tempo de execução. Por exemplo, em Java, *java.lang.Object* é superclasse de todas as outras classes do sistema. Desta forma, ao obter uma variável do tipo *Object*, qualquer classe do sistema pode estar instanciada nesta variável. Com isso, o operador *instanceof* irá confirmar a instância da variável, possibilitando ao desenvolvedor que efetue um *cast* do tipo mais geral para o tipo mais específico que foi confirmado por *instanceof*. Esta conversão ocorre principalmente porque o desenvolvedor deseja executar funcionalidades do tipo mais específico, que não podem ser utilizadas diretamente em variáveis que apontam para o tipo mais geral. O problema desta abordagem ocorre quando é realizada para se obter a instância de subclasses de herança ou interface, executando comportamentos específicos. Com isso, esta verificação quebra o encapsulamento e viola os princípios *LSP* para herança e *ISP* para interface.

```
public void reconcile(WSDLElement element, Element changedElement) {
    ReconciliationBPELReader reader = getReader(element, changedElement);
    if (element instanceof Activity) {
        reader.xml2Activity((Activity)element, changedElement);
    } else if (element instanceof Process) {
        reader.xml2Process(changedElement);
    } else if (element instanceof Import) {
        reader.xml2Import((Import)element, changedElement);
    } else if (element instanceof Condition) {
        reader.xml2Condition((Condition)element, changedElement);
    } else if (element instanceof CompletionCondition) {
        reader.xml2CompletionCondition((CompletionCondition)element, changedElement);
    } else if (element instanceof Branches) {
        reader.xml2Branches((Branches)element, changedElement);
    } else if (element instanceof Expression) {
        reader.xml2Expression((Expression)element, changedElement);
    } else if (element instanceof Documentation) {
        reader.xml2Documentation((Documentation)element, changedElement);
    } else if (element instanceof Link) {
        reader.xml2Link((Link)element, changedElement);
    } else if (element instanceof Links) {
        reader.xml2Links((Links)element, changedElement);
    } else if (element instanceof ElseIf) {
        reader.xml2ElseIf((ElseIf)element, changedElement);
    } else if (element instanceof Else) {
        reader.xml2Else((Else)element, changedElement);
```

Figura 4 – Quebra de encapsulamento por *instanceof*

Um exemplo do uso incorreto de *instanceof* é mostrado na Figura 4. O método RECONCILE() da classe ORG.ECLIPSE.BPEL.MODEL.UTIL.RECONCILIATIONHELPER recebe como argumento uma variável do tipo ELEMENT, que é uma superclasse. E também recebe uma variável do tipo WSDLELEMENT, que será usada para obter uma variável do tipo RECONCILIATIONBPELREADER. Esta variável por sua vez possui um método distinto para cada subclasse de ELEMENT. Por isso, este método efetua uma verificação de *instanceof* para descobrir qual subclasse de ELEMENT é a real instância que foi enviada como parâmetro, e executar o método apropriado de RECONCILIATIONBPELREADER. Esta abordagem gera uma série de efeitos colaterais, sendo que o mais visível é a necessidade de adicionar um novo *else if* neste método toda vez em que ELEMENT receber uma nova subclasse, violando o princípio *OCP*. Além disso, a exposição do comportamento das superclasses gera fortes acoplamentos entre as classes. Com isso, toda vez em que for necessário alterar a superclasse ELEMENT ou algumas de suas subclasses, a alteração irá refletir nas classes dependentes, gerando uma propagação em cascata de alterações no sistema.

2.1.5 Métricas Estruturais

As métricas estruturais foram propostas para calcular a complexidade das classes em sistemas *POO* (CHIDAMBER; KEMERER, 1994). Neste caso, combinando valores de métricas, é possível verificar se existe tendência de violar algum princípio *SOLID*, ou se está infectada com algum *Code Smell*. Para todas as métricas listadas a seguir, quanto maior for o valor encontrado para determinada classe, maior será a sua depreciação em relação à medida calculada pela métrica:

a) *Coupling Between Objects (CBO)* - Calcula o nível de acoplamento da classe com outras.

b) *Effective Lines of Code (ELOC)* - Obtém a quantidade de linhas de código. Neste caso, são removidas linhas que contém espaços em branco ou comentários.

c) *Lack of Cohesion of Methods (LCOM)* - Calcula o nível de coesão da classe mediante os acessos para um ou mais atributos em comum pelos serviços da classe.

d) *Number of Attributes (NOA)* - Obtém a quantidade de atributos da classe.

e) *Number of Methods (NOM)* - Obtém a quantidade de métodos da classe.

f) *Number of Public Attributes (NOPA)* - Obtém a quantidade de atributos públicos da classe.

g) *Response for a Class (RFC)* - Indica a capacidade de resposta que a classe possui ao receber mensagens. Estão incluídos métodos que foram herdados e construtores.

h) *Weighted Methods Per Class (WMC)* - Efetua uma somatório da complexidade de todos os métodos da classe.

2.2 Ferramental Utilizado

O objetivo desta seção consiste em destacar as ferramentas utilizadas para atingir os objetivos deste trabalho.

2.2.1 Code Smell Analyser

Code Smell Analyser é uma ferramenta para detecção de *Code Smells*, que também extrai os valores das métricas estruturais. Foi construída por (TUFANO et al., 2015), e implementada com base nas regras de (MOHA et al., 2010), que identifica os *Code Smells* baseado nos valores das métricas estruturais. A seguir, serão apresentadas as regras que esta ferramenta utiliza para detectar cada *Code Smell*:

a) *Class Data Should be Private* - O sistema irá detectar este *Code Smell* caso a quantidade de atributos públicos (*NOPA*) da classe seja maior do que 10.

$$NOPA > 10$$

b) *Complex Class* - Este *Code Smell* será identificado caso a complexidade da classe seja maior do que 200.

$$WMC > 200$$

c) *Functional Decomposition* - Será detectado caso a quantidade de métodos (*NOM*) da classe seja menor do que 3 e o nome da classe possua alguma das palavras chave: ('make','create', 'creator','execute','exec','compute','display','calculate')

$$(NOM < 3) \wedge NAME(\text{"}make\text{"} \vee \text{"}create\text{"} \vee \text{"}creator\text{"} \vee \text{"}execute\text{"}$$

$$\vee\text{"}exec\text{"} \vee \text{"}compute\text{"} \vee \text{"}display\text{"} \vee \text{"}calculate\text{"})$$

d) *God Class* - Será identificado quando o indicador de coesão da classe (*LCOM*) for maior do que 350 e a quantidade efetiva de linhas do código-fonte (*ELOC*) da classe for maior do que 500. Também será detectado quando a soma dos atributos (*NOA*) e métodos (*NOM*) da classe for maior do que 20 e a quantidade efetiva de linhas do código-fonte (*ELOC*) da classe for maior do que 500.

$$((LCOM > 350) \vee (NOA + NOM > 20)) \wedge (ELOC(CLASS) > 500)$$

e) *Lazy Class* - Será identificado quando a soma dos atributos (*NOA*) e métodos (*NOM*) da classe for menor do que 3.

$$NOA + NOM < 3$$

f) *Long Method* - Será identificado quando algum método possuir a quantidade efetiva de linhas do código-fonte (ELOC) maior do que 100, e a quantidade de parâmetros do método em questão for maior ou igual a 2.

$$(ELOC(METHOD) > 100) \wedge (PARAMETERS >= 2)$$

g) *Spaghetti Code* - Será identificado quando a quantidade efetiva de linhas do código-fonte (ELOC) da classe for maior do que 600, e a classe possuir algum método sem parâmetros e com mais de 100 linhas efetivas de código-fonte.

$$(ELOC(CLASS) > 600 \wedge ELOC(METHOD) > 100 \wedge PARAMETERS == 0)$$

2.2.2 *BOA* - Infra-estrutura de consulta em dados de repositórios

BOA é uma infra-estrutura que inclui *datasets* dos repositórios *GitHub* (até setembro/2015) e *SourceForge* (até setembro/2013), com milhares de sistemas escritos em diversas linguagens de programação. O *BOA* também possui uma linguagem *DSL (Domain Specific Language)* para minerar tais sistemas, visando buscar informações sobre o histórico de alterações em seus arquivos através da criação de *scripts*. Estes *scripts* podem ser executados por sua interface *Web* [6] ou diretamente por *plugins* escritos em Java (DYER et al., 2013). Neste trabalho, o BOA foi utilizado para buscar o histórico de alterações nas classes de cada projeto analisado.

2.2.3 Controladores de Histórico de Versões

Para obter as métricas e *Code Smells* da ferramenta *Code Smell Analyser*, é necessário primeiramente obter o código-fonte de cada sistema. Embora o *BOA* disponibilize informações sobre o conteúdo dos arquivos que compõe os sistemas, este não permite acesso ao seu código-fonte. Por isso, foi necessário efetuar o *download* dos sistemas diretamente no seu respectivo controlador de histórico de versões (*VCS*). Um *VCS* é um sistema que registra o histórico de alterações dos sistemas ao longo do tempo, possibilitando realizar um conjunto de funcionalidades, como recuperar versões desses sistemas, analisar o histórico de alterações de qualquer arquivo, dentre outras. (KOC A., 2011).

Neste trabalho, foram utilizados sistemas que utilizam os *VCS GIT*, *Subversion (SVN)* e *Concurrent Version System (CVS)*. Os sistemas *GIT* utilizados neste trabalho estão armazenados no repositório *GitHub*, enquanto que os sistemas *SVN* e *CVS* estão armazenados no repositório *SourceForge*. As *URLs* destes sistemas foram extraídas do *BOA*.

Para obter informações sobre as versões dos sistemas e efetuar o *download* das mesmas, é necessário executar comandos sobre os *VCS* citados. Como este trabalho efetua análises

[6] http://boa.cs.iastate.edu/

sobre uma quantidade razoavelmente grande de sistemas, este procedimento necessita ser realizado de maneira automática. Por isso, foram utilizados os seguintes *frameworks*:

a) *CVS - NetBeans CVSClient* [7]

b) *SVN - SVNKIT* [8]

c) *GIT - JGIT* [9]

2.2.3.1 *Commit*

Um *commit* representa um conjunto de alterações do sistema que foi efetivada no *VCS*. Cada *commit* possui informações, como a data de realização, um texto descritivo sobre as alterações realizadas, o usuário que executou este procedimento, a lista de arquivos que sofreram as modificações, o tipo da alteração de cada arquivo (inclusão, modificação ou exclusão), o código-fonte de cada arquivo alterado, dentre outras. Quando se obtêm a lista de *commits* de um sistema ordenadas por data, é possível obter todo o histórico de acontecimentos em um sistema de forma sequencial, ou seja, as modificações que ocorreram ao longo do seu ciclo de vida (KOC A., 2011).

Um *commit* pode ser realizado para gerar um incremento sobre uma versão em desenvolvimento do sistema, como também pode representar uma correção de algum *bug* detectado. Os *VCS* não possuem um campo que indica qual foi a motivação pelo qual determinado *commit* foi realizado. Contudo, o *BOA* disponibiliza uma função chamada *isfixingrevision()* onde é possível verificar se um *commit* é do tipo *fixing revision* [10]. Esta função verifica um conjunto de expressões regulares no texto descritivo do *commit* para verificar se houve correção de *bugs* em alguma versão do sistema.

2.2.3.2 *Branch*

Um *branch* simboliza um fluxo de trabalho com seu conjunto de *commits*. Cada *commit* realizado no sistema está associado com um *branch*. Sendo assim, todo *VCS* precisa possuir pelo menos um *branch*, que neste caso representa a linha principal de alterações do sistema. No *GIT*, este *branch* principal possui por padrão o nome de *master*. Já no *CVS* e *SVN*, o *branch* principal é simbolizado pelo *trunk*. A diferença é que nos sistemas *GIT*, não existe obrigatoriedade que a *branch* principal seja sincronizada com as demais, embora este procedimento seja recomendável. Com o *trunk* já existe esta obrigatoriedade, pois este representa o estado atual do sistema. Em sistemas *GIT* que foram migrados do *CVS* ou *SVN*, é comum que a *branch* principal receba o nome de *trunk* ao invés de *master*.

Além do *branch* principal, outros *branches* podem ser criados a partir de qualquer outra *branch*. Quando uma equipe de desenvolvedores inicia o trabalho em uma nova

[7] https://versioncontrol.netbeans.org/javacvs/library/
[8] https://svnkit.com/
[9] https://eclipse.org/jgit/
[10] http://boa.cs.iastate.edu/docs/dsl-functions.php

versão do sistema, é recomendável criar uma nova *branch*, e após a conclusão, seja feito um *merge* entre as alterações desta versão com a *branch* principal do sistema.

Neste trabalho, foram consideradas apenas as alterações realizadas sobre a *branch* principal de cada *VCS*, pois o *BOA* possui o histórico de alterações somente desta *branch*. Também é comum que trabalhos relacionados efetuem mineração apenas na *branch* principal (TUFANO et al., 2015). Como exemplo, o sistema LUCENE-SOLR [11] possui 26.524 *commits* apenas na *branch* principal. Ao efetuar um somatório dos *commits* em todas as *branches*, este valor ultrapassa 500.000 *commits*. Sendo assim, para obter uma quantidade representativa de sistemas analisados, é inviável analisar todas as *branches* em virtude do elevado tempo de processamento.

2.2.3.3 *Tags*

Uma *tag* é um apontador para um *commit* específico do sistema. Quando uma *tag* for requisitada para um *VCS*, obtém-se o estado do sistema até aquele *commit* apontado pela *tag*. Por isso, é comum que os desenvolvedores criem uma *tag* para cada versão do sistema(KOC A., 2011). A Figura 5 apresenta a interface *web* dos repositórios *SourceForge* e *GitHub*, onde é possível verificar as *tags* de um sistema específico para cada *VCS*.

Neste trabalho, foram utilizados os *frameworks VCS* para obter as *tags* de cada sistema analisado. E após análise, efetuar o *download* das versões escolhidas e utilizá-las de entrada para a ferramenta *Code Smell Analyser*.

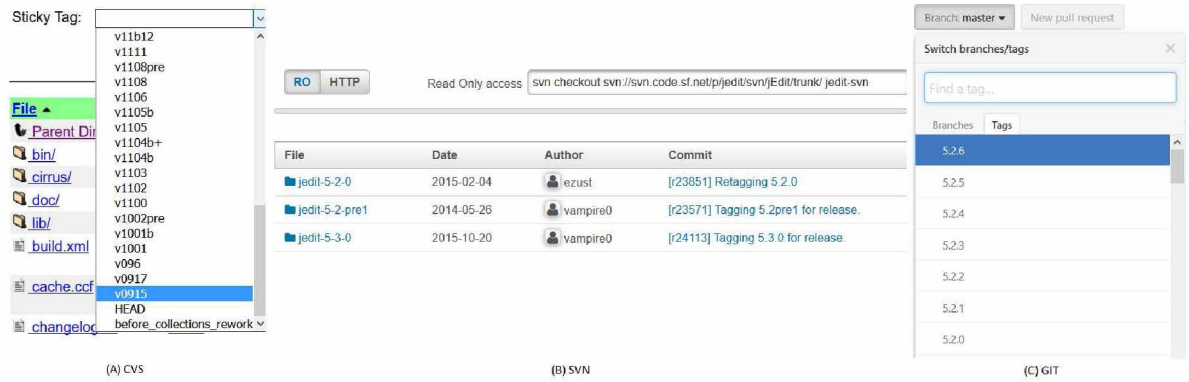


Figura 5 – Tags de sistemas armazenados nos *VCS CVS,SVN* e *GIT*

Fonte: http://www.sourceforge.net e http://www.github.com

[11] https://github.com/apache/lucene-solr

Capítulo 3

Metodologia

Com a contextualização do problema e os fundamentos teóricos estabelecidos, neste capítulo serão propostas as perguntas de pesquisa. Em seguida, serão descritas as etapas empregadas para responder estas perguntas, começando pelos critérios de seleção dos sistemas, modelagem dos dados, e por fim, os métodos estatísticos utilizados na análise de dados.

3.1 Perguntas de pesquisa

Como mencionado na Seção 1.2, o objetivo deste trabalho consiste em investigar o uso de herança e interface em sistemas escritos na linguagem Java, analisando o histórico de alterações dos sistemas, além de versões específicas de cada. Esta análise se restringe a classes estendendo superclasses ou interfaces internas. Com isso, foram descartadas heranças e implementações de interface como *extends javax.swing.JFrame* ou *implements java.lang.Runnable*. Mais especificamente, o estudo visa abordar as seguintes seis perguntas de pesquisa (*RQs #*):

a) RQ #1: A época em que o sistema foi construído exerce alguma influência sobre a frequência no uso de herança ou implementação de interfaces?

   Com esta pergunta de pesquisa, espera-se responder se a prática de *evitar a herança e favorecer a composição com interfaces* têm sido ao menos parcialmente seguida pelos desenvolvedores. Em caso afirmativo, a tendência seria a detecção de um crescimento no uso de interfaces em sistemas mais recentes, e ao mesmo tempo, decréscimo no uso de herança.

b) RQ #2: A época em que o sistema foi construído exerce alguma influência sobre a quantidade das quebras de encapsulamento por *instanceof*?

   Neste caso, será verificado se sistemas mais recentes têm obtido menos quebras de encapsulamento pelo uso do operador *instanceof* em classes de herança ou implementação de interfaces. O resultado desta pergunta de pesquisa irá apontar se

os desenvolvedores têm observado esta prática, evitando este tipo de quebra de encapsulamento.

c) RQ #3: A época em que o sistema foi construído exerce alguma influência sobre a quantidade de alterações corretivas em classes com herança e interface?

   Com esta pergunta de pesquisa, será analisado se as classes de herança e implementação de interface têm adquirido maior estabilidade em sistemas mais recentes, com menos alterações corretivas.

d) RQ #4: Classes com herança ou interface possuem níveis adequados de coesão e acoplamento?

   Será analisado se classes com herança ou interface possuem maior tendência de possuir indicadores negativos de coesão e o acoplamento. Esta medição será feita comparando com as classes que não possuem herança ou interface. O resultado desta análise irá ajudar a caracterizar melhor quais indicadores *POO* ainda não são corretamente aplicados em classes com herança ou interface. Além disso, será verificado se irá ocorrer alguma alteração entre estes indicadores entre a primeira e última versão dos sistemas.

e) RQ #5: Quais *Code Smells* ocorrem predominantemente em classes com herança ou interface?

   Será analisado se classes com herança ou interface possuem maior tendência de estarem infectados com *Code Smells* específicos, e também, se estas infecções (caso existam) continuam até a última versão do sistema. O resultado dessa análise ajudará a entender quais falhas de projeto são mais prováveis de ocorrer em classes de herança ou implementação de interface.

f) RQ #6: Com qual frequência ocorre adição ou remoção de herança e implementação de interfaces sobre as classes? E por quais razões estas operações são realizadas?

   Nesta pergunta de pesquisa, espera-se entender com qual frequência e como os recursos de herança e interface são inseridos e removidos das classes. Estas motivações para estas operações podem explicar como herança e interface são de fato empregadas nos sistemas, sobre o ponto de vista do desenvolvedor.

3.2 Extração e Análise dos Sistemas

Nesta seção serão apresentados os passos utilizados para a realização da filtragem dos sistemas, que envolve toda a coleta, seleção e análise dos sistemas utilizados como amostragem para este trabalho. A Figura 6 apresenta de forma sucinta os passos empregados

para a filtragem dos sistemas. As cinco Subseções a seguir irão descrever as etapas de forma ordenada.

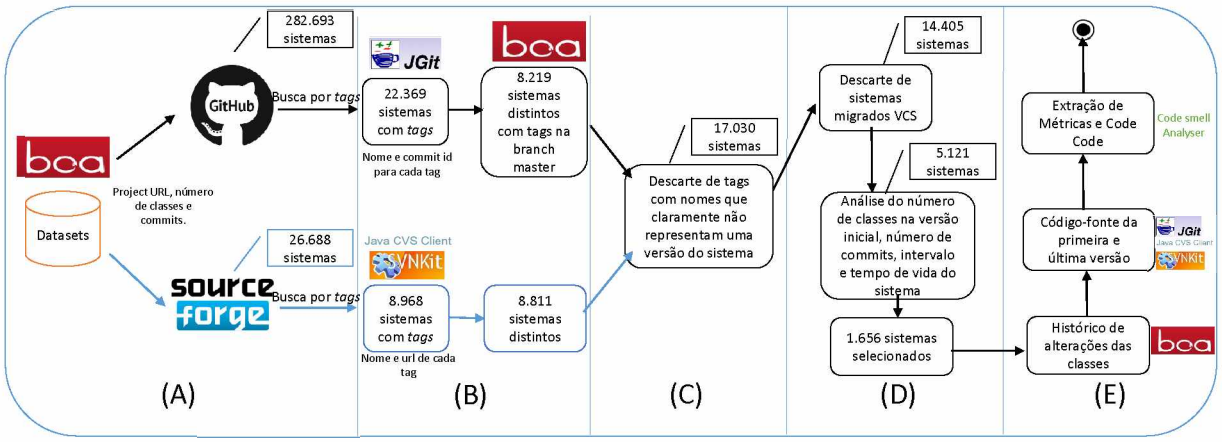


Figura 6 – Etapas para a realização da filtragem dos sistemas

3.2.1 Extração de Sistemas no *BOA*

Para obter a amostragem inicial dos sistemas, foi utilizada a infraestrutura *BOA*[1], como mostra a Figura 6 *(A)*. Foi construído um *script* para obter a *URL*, quantidade de classes e *commits* para 309.381 sistemas escritos em Java. A escolha desta linguagem ocorreu em função da limitação da ferramenta *Code Smell Analyser*, que retorna os valores de métricas estruturais e detecta *Code Smells* apenas para sistemas construídos nesta linguagem. Deste total, 282.693 sistemas estão disponíveis no repositório *GitHub* [2], e 26.688 no repositório *SourceForge* [3].

Para obter uma maior distribuição quanto à época em que os sistemas foram construídos, utilizou-se sistemas dos repositórios *GitHub* e *SourceForge*. A Figura 7 apresenta um histograma evidenciando a quantidade de sistemas agrupados pelo ano de criação, separados pelos dois repositórios citados. O repositório *SourceForge* possui maior concentração de sistemas construídos entre 2000 e 2009, enquanto que o *GitHub* concentra sistemas construídos entre 2010 e 2013. Desta forma, ao obter sistemas de ambos os repositórios, a tendência é que se obtenha um aumento na variabilidade de épocas em que foram construídos os sistemas a serem selecionados.

3.2.2 Busca de Sistemas com *Tags*

Para verificar se a época em que o sistema foi construído exerce alguma influência na utilização do uso de classes com herança ou interfaces, é necessário obter a versão inicial

[1] <http://boa.cs.iastate.edu/>
[2] <http://www.github.com/>
[3] <http://www.sourceforge.net/>

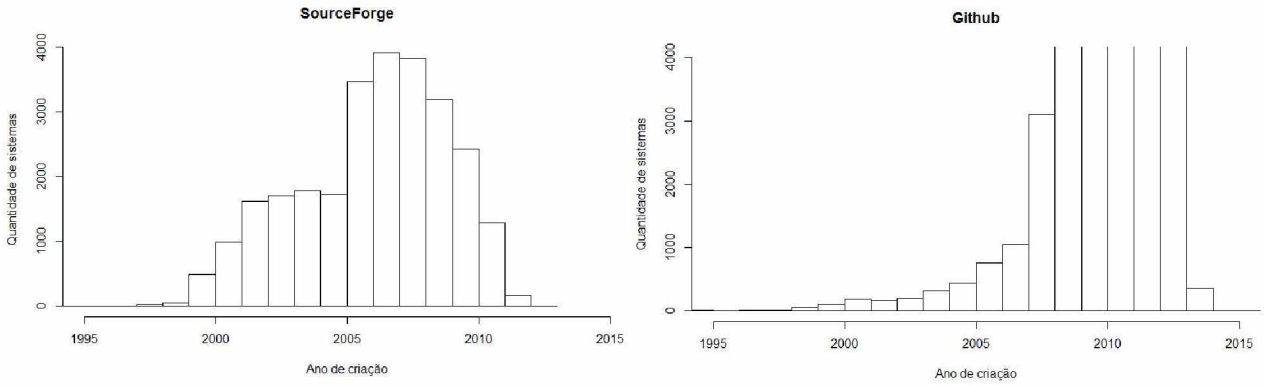


Figura 7 – Histograma com a quantidade de sistemas por ano de criação, separados pelos repositórios *SourceForge* e *GitHub*

de cada sistema, pois esta reflete a forma como cada sistema foi projetado. Para obter as versões dos sistemas, é necessário considerar apenas os que possuem *tags*. Por isso, foi necessário utilizar os *frameworks* citados na Seção 2.2.3 para buscar as *tags* de cada sistema nos seus respectivos *VCS GIT,CVS* e *SVN*. Esta etapa é apresentada na Figura 6 (B). Desta forma, todos os sistemas que não possuem *tags* foram descartados. Sendo assim, houve descarte de 17.720 (66,39%) sistemas armazenados no repositório *SourceForge*. Para os sistemas armazenados no *GitHub*, 59.352 sistemas (20,99%) exigiram autenticação para efetuar a consulta, e 200.977 (71,09%) sistemas não possuíam *tags*, totalizando 260.329 sistemas (92,08%) descartados.

Ao consultar as *tag*, os *VCS CVS* e *SVN* retornam o nome de cada *tag* e sua respectiva *url* para download. Para os sistemas *GIT*, são retornados o nome de cada *tag* e o *commit* associado à mesma. Como sistemas *GIT* podem possuir diversas *branches*, e o *BOA* possui *commits* referentes apenas à *branch* principal de cada sistema, é necessário descartar *tags* que não referenciam *commits* da *branch* principal. Também foram descartados sistemas que foram copiados de outros repositórios (*fork*). Com isso, foram descartados 14.150 sistemas (63,25%).

Para os sistemas armazenados no repositório *SourceForge*, foram descartados 157 sistemas (1,75%) por possuírem duplicidade nos seus respectivos *VCS CVS* e *SVN*. Neste caso, foi descartada a versão do *SVN*, pois foi verificado manualmente que, por ser mais antiga, a versão do *CVS* possuía todo o histórico de *tags*, diferentemente da versão *SVN*, que resumiu todas as *tags* criadas no *CVS* em apenas uma.

3.2.3 Análise das *Tags*

Com 17.030 sistemas (5,50%) restantes para os dois repositórios, foi realizada uma avaliação sobre os nomes das *tags*. Esta avaliação deve ser feita porque para este trabalho, as *tags* devem representar a conclusão de uma versão do sistema. Contudo, foi constatado que

alguns nomes de *tags* representam marcos para o início da construção de uma versão. Com isso, foi construído um script (*distinct UPPER(REGEXP_REPLACE(name_tag,'[^A-Za-z]',''))* que retorna apenas as letras dos nomes das *tags*. Sendo assim, obteve-se uma planilha com 7.371 nomes de *tags* distintas, que foram analisadas manualmente. Desta forma, foram detectadas e descartadas *tags* que possuem nomes como *start, initial, test, before, beta, alpha, pre, demo, old, init, none, dev, example, first import, experimental, hello world, inicio, readme,first commit,RC[0..9]* e *CR[0..9]*. Este filtro descartou *tags* de 6.498 sistemas, e 2.625 sistemas (15,27%) foram descartados porque somente possuíam *tags* com estas palavras-chave. Este fluxo é mostrado na Figura 6 (C).

3.2.4 Análise dos Sistemas

Nesta seção, será descrito como os 14.405 sistemas (4,65%) restantes foram filtrados. Este fluxo é mostrado na Figura 6 (D). O primeiro critério empregado consistiu em eliminar sistemas que foram migrados de outros *VCS*. Sistemas migrados geralmente não preservam o histórico das suas alterações, possuindo adição da maior parte de suas classes logo nos seus primeiros *commits*. Com isso, a data de criação que o *VCS* irá mostrar não representa a época em que foram realmente criadas. Com isso, foi adotado como métrica o descarte de sistemas que possuem pelo menos 50% do seu total de classes Java incluídos nos 20 primeiros *commits* do sistema.

Aplicando o critério mencionado, foram descartados 9.284 sistemas (64,44%), sendo que 3.684 destes (39,68%) possuíam no máximo 20 *commits*. Como exemplo, destaca-se o projeto OPENJDK7 [4], onde 10.103 classes Java (65,28%) foram acrescentadas nos seus 10 primeiros *commits*. Durante todo o ciclo de vida do sistema, contabilizou-se um total de 15.476 classes. Como várias destas foram construídas em versões anteriores deste sistema, o descarte se faz necessário. Além disso, caso este sistema fosse analisado, mereceria análise separada dada sua especificidade em relação à equipe e ao propósito.

O próximo critério empregado consistiu em analisar todos os sistemas restantes, separados por repositório, utilizando quatro medidas para cada sistema, como mostra a Figura 8. Estas medidas são: número de classes da versão inicial, número de *commits*, tempo de vida (intervalo entre o primeiro e último *commit* em meses) e intervalo (em meses) entre a data de criação e a versão inicial. Para as três primeiras medidas citadas, foram descartados os sistemas que estão no primeiro quartil da distribuição. A primeira medida foi empregada visando descartar sistemas que possuem as primeiras versões excessivamente pequenas. Estas versões podem não representar a arquitetura do sistema, por ainda estarem em fase preliminar. A segunda medida foi empregada para descartar sistemas com poucas alterações. Neste caso, sistemas pouco alterados tendem a possuírem poucas modificações nas suas classes. A terceira medida foi empregada para descartar sistemas com intervalo de tempo excessivamente pequeno, pois para responder as perguntas

[4] <https://github.com/ikeji/openjdk7-jdk>

de pesquisa *RQ #4* e *RQ #5*, é necessário que exista um intervalo considerável entre as versões inicial e final do sistema.

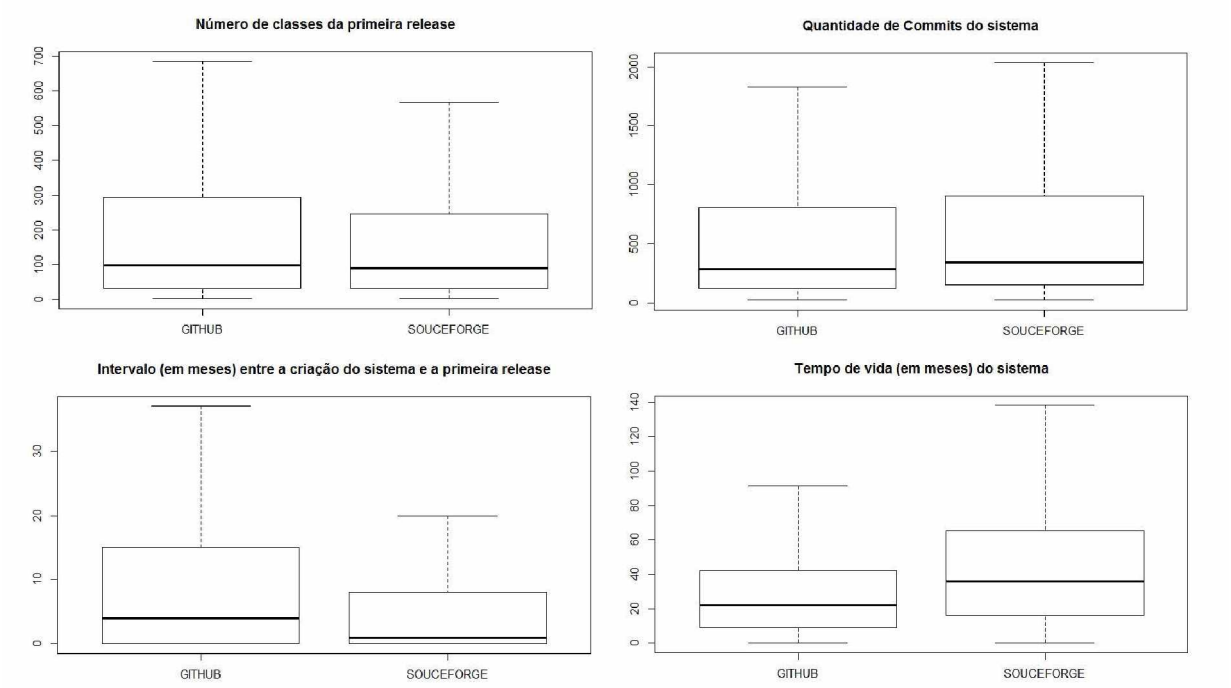


Figura 8 – Boxplot para análise dos sistemas restantes, agrupados pelos repositórios *GitHub* e *SourceForge*

Para a quarta medida, foram descartados os sistemas que estão abaixo do primeiro ou acima do terceiro quartil. Sistemas abaixo do primeiro quartil tendem a possuir a versão inicial logo no início do seu ciclo de vida, enquanto que acima do terceiro quartil tendem a possuir a versão inicial depois de um longo período de existência. Este critério foi empregado porque não é possível avaliar com exatidão se a *tag* escolhida representa de fato a versão inicial do sistema. Portanto, *tags* que possuem data muito próxima da criação do sistema podem representar um marco inicial, enquanto *tags* que possuem data muito distante da criação do sistema podem não representar mais a versão inicial. Portanto este descarte tende a minimizar a escolha de sistemas com *tags* que não estejam na versão inicial.

Aplicando estes critérios, foram descartados os sistemas com menos de 32 classes na versão inicial para ambos os repositórios. Também foram descartados sistemas com menos de 122 *commits* para o *Github* e 150 para o *SourceForge*. Sobre o tempo de vida, foram descartados sistemas com intervalo de *commits* inferior a 9 meses no *Github* e 16 meses no *SourceForge*. E por fim, foram selecionados sistemas em que a data de criação e a versão inicial possuam intervalo inferior a 15 meses para o *Github* e 8 meses para o *SourceForge*. Nesta análise, o primeiro quartil retornou 0 meses para ambos os repositórios.

Com a conclusão da análise mencionada, restaram 880 sistemas (0,31%) *Github* e 776 sistemas (2,9%) *SourceForge*, totalizando 1.656 sistemas (0,53%). Com isso, os sistemas selecionados possuem maior equilíbrio entre as suas características, representando a amostra que será empregada neste trabalho.

3.2.5 Informações sobre os Sistemas Selecionados

Com a seleção dos 1.656 sistemas, foram importadas informações sobre todo o histórico de *commits* e classes no *BOA*, além de efetuar o *download* das versões inicial e final de cada sistema com os *frameworks VCS* já citados. E por fim, foi realizada a extração de métricas e *Code Smells* com a ferramenta *Code Smell Analyser*. Este fluxo é mostrado na Figura 6 (E).

A Figura 9 apresenta um histograma evidenciando a quantidade dos sistemas empregados neste trabalho por ano de criação, onde observa-se que houve uma distribuição mais adequada dos sistemas entre os anos 2000 e 2013.

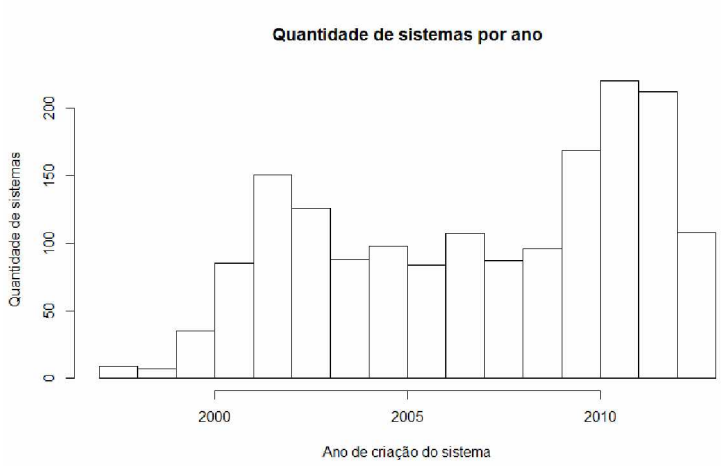


Figura 9 – Histograma com a quantidade de sistemas por ano de criação

A Tabela 1 apresenta informações sobre os sistemas selecionados. As colunas *Min,Med* e *Max* representam a quantidade mínima, mediana e máxima de classes e *commits* dos sistemas. Comparando com outros trabalhos, foi observado que este trabalho possui uma amostragem representativa de classes e *commits* para análise (TEMPERO; NOBLE; MELTON, 2008)(TUFANO et al., 2015). Sobre o intervalo de criação entre os sistemas, alguns sistemas foram migrados para o *GitHub* preservando o seu histórico de *commits* com as datas corretas, o que justifica este repositório possuir sistemas mais antigos, criados em 1997.

Tabela 1 – Características dos sistemas analisados

Repositório	Sistemas	Commits				Classes e Interfaces				Criação
		Min	Med	Max	Total	Min	Med	Max	Total	
Github	880	122	543	6.700	558.403	31	248	18.359	516.327	1997-2013
Sourceforge	776	155	798	28.570	743.453	33	481	20.944	780.872	1997-2011
Total	1.656	-	-	-	1.301.856	-	-	-	1.297.199	-

3.3 Modelagem dos Dados

Para armazenar os dados importados das ferramentas empregadas neste trabalho, foi construído um sistema que persiste os resultados em um banco de dados relacional. O Diagrama Entidade Relacionamento (DER) deste sistema pode ser visualizado na Figura 10.

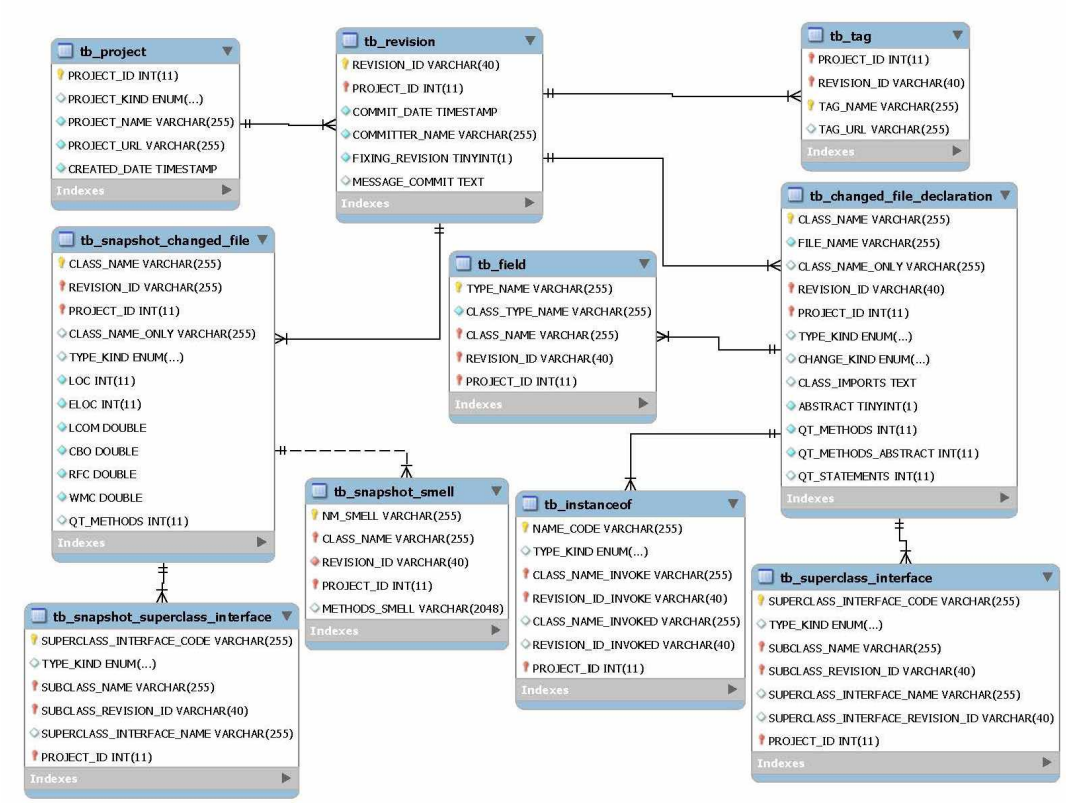


Figura 10 – Diagrama do banco de dados que armazena os dados a serem analisados

As tabelas deste banco representam os dados dos sistemas obtidos das ferramentas *BOA, Code Smell Analyser* e *frameworks VCS*. As informações obtidas do *BOA* populam as tabelas *TB_PROJECT, TB_REVISION, TB_CHANGED_FILE_DECLARATION, TB_FIELD, TB_SUPERCLASS_INTERFACE* e *TB_INSTANCEOF*. As informações obtidas do *Code Smell Analyser* populam as tabelas *TB_SNAPSHOT_CHANGED_FILE, TB_SNAPSHOT_SUPERCLASS_INTERFACE* e *TB_SNAPSHOT_SMELL*. E por fim, as informações obtidas dos *frameworks VCS* populam a tabela *TB_TAG*. Além de persis-

tir os dados, o sistema mencionado também executa outros tipos de atividades que serão destacadas a seguir:

a) Verificações de consistência nos dados importados - remoção de dados duplicados, garantia de integridade entre os dados.

b) Realizar ligações entre as entidades - o *BOA* retorna as dependências de cada classe, como atributos, referências por *instanceof*, superclasses e interfaces. Contudo, o retorno consiste no código escrito pelo desenvolvedor da classe. Portanto, é necessário verificar se estas dependências são de outras classes internas do sistema, e em caso afirmativo, efetuar as ligações entre as entidades do banco de dados. Para efetuar esta verificação, primeiramente é verificado se a superclasse se encontra no mesmo pacote que a sua dependência, e em caso negativo, os *imports* da subclasse são avaliados para verificar se a superclasse se encontra em algum deles.

```
1  package net.minecraft.src;
2
3  import java.io.FileInputStream;
4  import java.io.FileNotFoundException;
5  import java.io.FileOutputStream;
6  import java.io.IOException;
7  import java.util.List;
8  import java.util.Properties;
9  import java.util.Random;
10 import net.minecraft.client.Minecraft;
11 import org.lwjgl.input.Keyboard;
12 import org.lwjgl.input.Mouse;
13 import sun.util.logging.resources.logging;
14
15 public class ThxEntityHelicopter extends ThxEntity
16 {
17     static int instanceCount = 0;
18
19     // controls and options
20     // set from mod thx.properties
```

Figura 11 – Exemplo de busca do nome das classes

A Figura 11 ilustra um exemplo para a verificação mencionada. A classe NET.MINECRAFT.SRC.THXENTITYHELICOPTER estende de THXENTITY. O *BOA* irá retornar apenas o nome THXENTITY, informando que esta é superclasse de THXENTITYHELICOPTER. Para descobrir se THXENTITY é uma classe interna do sistema, primeiramente é verificado se está no mesmo pacote que a subclasse, onde neste caso é NET.MINECRAFT.SRC. Em caso negativo, é feita uma verificação em cada *import* da subclasse (linhas 3 até 13), verificando se a superclasse se encontra em algum destes diretórios. Caso não seja encontrado, a superclasse será considerada externa, e portanto será descartada da análise.

c) Descarte de informações desnecessárias - são descartadas informações que não possuem utilidade para responder às perguntas de pesquisa. Um exemplo consiste em heranças e implementações de interfaces externas, atributos e verificações de *instanceof* que não sejam para classes internas do sistema. Estes descartes são necessários porque ao armazenar informações de 1.656 sistemas, algumas tabelas

tornam-se excessivamente grandes, elevando o tempo de consulta dos dados. Portanto, é preciso manter no banco de dados apenas informações pertinentes à esta pesquisa.

d) Geração dos arquivos de saída - o sistema efetua uma compilação dos dados necessários para responder às perguntas de pesquisa, gerando os arquivos para serem analisados.

3.4 Análise dos Dados

Esta seção apresenta a metodologia empregada para responder cada pergunta de pesquisa proposta no início deste Capítulo. Com isso, para cada pergunta, serão descritos o conjunto de dados necessários para serem extraídos dos sistemas estudados neste trabalho, o método estatístico empregado para análise e a sua devida justificativa.

3.4.1 A Influência da Idade no Uso de Herança e Interfaces

Para responder a pergunta de pesquisa *RQ #1*, são avaliadas a versão inicial de cada sistema Java, para medir o uso de herança e interface. O esperado é que a versão inicial represente como os desenvolvedores pensaram sobre a arquitetura do sistema, assim como na construção de suas abstrações. Em seguida, é avaliado se a idade do sistema (em meses) influencia o número de classes em herança ou implementando interfaces. Para realizar esta avaliação, foi utilizado um modelo de regressão binomial negativa (*Negative Binomial Regression (NBR)*), porque a variável de resposta (para cada projeto, o número de classes em herança ou implementando interface) é uma contagem. O modelo *NBR* também pode lidar com sobre-dispersão, ou seja, quando a variância da resposta é maior que a média (COHEN; COHEN, 1975). Desta forma, o modelo *NBR* proposto é:

$$\ln_{(INHER)} = INTERCEPT + \beta_1(CLASSES) + \beta_2(MONTHS\_RELEASE) \quad (1)$$

$$\ln_{(INTER)} = INTERCEPT + \beta_1(CLASSES) + \beta_2(MONTHS\_RELEASE) \quad (2)$$

As variáveis de resposta são o *log* de *INHER* (número de classes em herança) e *INTER* (número de classes implementando interface). O modelo é controlado pelo número de classes (variável preditora *CLASSES*, e a variável analisada foi o preditor *MONTHS_RELEASE*, que consiste no número de meses entre a data da versão inicial do projeto e a data da versão inicial do projeto mais recente (julho/2015).

O coeficiente *NBR* é interpretado da seguinte forma: para uma unidade modificada em uma variável preditora, a mudança de uma unidade no coeficiente β_i produz uma resposta esperada em e_i^{β}, dado que outras variáveis preditoras no modelo são mantidas constantes.

Em primeiro lugar, é verificado se a variável preditiva *MONTHS_RELEASE* é estatisticamente significativa para a resposta. Desta forma, é realizado o teste estatístico *ANOVA* com dois modelos *NBR*, um com a variável preditor *MONTHS_RELEASE* e outro sem esta. Em seguida, é realizado o teste estatístico de razão de verossimilhança para verificar se o modelo *NBR* é o mais apropriado para ser usado com os dados produzidos nesta pergunta de pesquisa. Este teste compara o *NBR* com o modelo de *Poisson*, que não possui parâmetro de dispersão para lidar com amostras super-dispersas.

Por fim, é efetuada uma comparação entre o uso de classes com herança e interface. Para isso, são obtidos a proporção de classes com herança e interface em dois intervalos. O primeiro intervalo considera sistemas com a primeira versão mais antigos que a mediana. E o segundo considera os sistemas mais recentes. As variáveis de proporção são descritas a seguir:

$$FATOR\_INHERITANCE = \frac{INHER}{CLASSES} \tag{3}$$

$$FATOR\_INTERFACE = \frac{INTER}{CLASSES} \tag{4}$$

3.4.2 A Influência da Idade nas Quebras de Encapsulamento por *instanceof*

Para responder a pergunta de pesquisa *RQ #2*, são contadas as classes distintas que estendem superclasses ou implementam interfaces, e que ocorrem como parâmetro do operador *instanceof*. Para obter os operadores *instanceof* realizados, é necessário construir um *script* para o *BOA*, filtrando por *expressions* do tipo *ExpressionKind.TYPECOMPARE*. Para isso, são considerados todos os *commits*, isto é, se uma classe ocorreu como parâmetro de *instanceof* em qualquer uma das revisões, é contada uma vez. Assim, é identificado se a idade do sistema (em meses) possui alguma influência sobre o número de classes que estendem superclasses ou implementam interfaces referenciadas pelo operador *instanceof*. Para isso, foi proposto o seguinte modelo *NBR*:

$$\ln_{(INHER)} = INTERCEPT + \beta_1(EXTENDS) + \beta_2(MONTHS\_CREATION) \tag{5}$$

$$\ln_{(INTER)} = INTERCEPT + \beta_1(IMPLEMENTS) + \beta_2(MONTHS\_CREATION) \tag{6}$$

As variáveis preditoras de resposta (*INHER e INTER*) representam o número de classes que estendem as superclasses ou implementam interfaces que ocorrem como parâmetro do operador *instanceof*. A variável preditora *EXTENDS* é o número de classes que estendem as superclasses, e *IMPLEMENTS* é o número de classes que implementam interfaces.

A última variável preditora é *MONTHS_CREATION*, que é o número de meses entre a data de criação do projeto e a data de criação do projeto mais recente (outubro/2013).

Para comparar o uso do operador *instanceof* entre classes com herança e interface, foram adicionadas duas novas variáveis, que representam o fator de uso do operador *instanceof* para herança e interface em cada sistema.

$$FATOR\_INHERITANCE = \frac{INHER}{EXTENDS} \tag{7}$$

$$FATOR\_INTERFACE = \frac{INTER}{IMPLEMENTS} \tag{8}$$

3.4.3 A Influência da Idade sobre Alterações Corretivas

Para responder a pergunta de pesquisa *RQ #3*, os *commits* do tipo *fixing revision* são identificados usando uma função do *BOA* chamado *isFixingRevision()*. Em seguida, para cada sistema, são contados a quantidade de *commits* do tipo *fixing revision* que modificaram classes com e sem herança ou implementação de interface. Estes dois grupos são analisados porque é necessário saber se apenas classes em herança ou implementando interfaces são influenciadas ou se todas as classes são influenciadas em conjunto. Para isso, como nas perguntas de pesquisa *RQ #1* e *RQ #2*, o modelo *NBR* é proposto:

$$\ln_{(INHER)} = INTERCEPT + \beta_1(FIXING) + \beta_2(MONTHS\_CREATION) \tag{9}$$

$$\ln_{(NOT\_INHER)} = INTERCEPT + \beta_1(FIXING) + \beta_2(MONTHS\_CREATION) \tag{10}$$

$$\ln_{(INTER)} = INTERCEPT + \beta_1(FIXING) + \beta_2(MONTHS\_CREATION) \tag{11}$$

$$\ln_{(NOT\_INTER)} = INTERCEPT + \beta_1(FIXING) + \beta_2(MONTHS\_CREATION) \tag{12}$$

As variáveis de resposta (*INHER e INTER*) representam, para cada projeto, o número de classes modificadas com herança ou implementação de interfaces em um *commits* do tipo *fixing revision*. As variáveis de resposta (*NOT_INHER e NOT_INTER*) representam o número de classes modificadas sem herança ou implementação de interfaces em um *commits* do tipo *fixing revision*. A variável preditora *FIXING* é o total de *commits* do tipo *fixing revision* para cada projeto Java. A última variável preditora é *MONTHS_CREATION*, que é o número de meses entre a data de criação do projeto e a data de criação do projeto mais recente (outubro/2013).

Para comparar a quantidade de *commits* do tipo *fixing revision* entre classes com herança e interface, foram adicionadas duas novas variáveis, que mostram a representatividade de destes *commits* em cada sistema.

$$FATOR\_INHERITANCE = \frac{INHER}{FIXING} \tag{13}$$

$$FATOR\_INTERFACE = \frac{INTER}{FIXING} \tag{14}$$

3.4.4 A Influência dos Indicadores de Coesão e Acoplamento

Para responder a pergunta de pesquisa *RQ #4*, foram obtidos as versões inicial e final de cada sistema escrito em Java, e a ferramenta *Code Smell Analyser* foi utilizada para extrair os valores das seguintes métricas estruturais: *CBO,ELOC,LCOM,NOM,RFC e WMC.*

Com isso, é verificado se existe uma diferença significativa entre os valores das métricas citadas, entre classes em herança ou implementação de interfaces, e classes sem herança ou implementação de interface. Para efetuar tal verificação, é aplicado o teste de *Mann Whitney U.* Esta comparação ocorre duas vezes, usando duas versões distintas dos sistemas (inicial e final).

3.4.5 A Influência dos *Code Smells*

Para responder a pergunta de pesquisa *RQ #5*, foram selecionados as versões inicial e final de cada sistema escrito em Java, e a ferramenta *Code Smell Analyser* foi utilizada no código-fonte, para extrair as classes infectadas pelos seguintes *Code Smells*: *God Class, Class Data Should Be Private, Functional Decomposition, Long Method, Complex Class, Lazy Class* e *Spaghetti Code.* A única exceção consiste no *Code Smell Refused Parent Bequest*, que não será analisado, pois este não é detectado pela ferramenta *Code Smell Analyser.*

Para identificar se algum *Code Smell* está ocorrendo com maior frequência em classes com herança ou interface, foi construída uma tabela de contingência, aplicando testes de *Fisher* e X^2, para verificar se a hipótese se confirma. Neste caso, cada *Code Smell* possui uma tabela de contingência própria.

3.4.6 Frequência e Motivação para Adição e Remoção de Herança e Interface

Para responder a pergunta de pesquisa *RQ #6*, foram analisados o histórico de *commits* das classes. Ao término da análise, cada classe do sistema foi separada em quatro grupos:

a) *Sempre* - classes que foram construídas com herança ou interface, e assim permaneceram até a sua exclusão ou último *commit* do sistema.

b) *Perdeu* - classes que foram construídas com herança ou interface, e este comportamento foi removido em algum *commit* do sistema.

c) *Depois* - classes que foram construídas sem herança ou interface, e adquiriram este comportamento após algum *commit* do sistema.

d) *Nunca* - classes que nunca possuíram herança ou interface, e assim permaneceram até a sua exclusão ou último *commit* do sistema.

O primeiro e quarto grupos tiveram a mesma estrutura desde a sua criação até o fim. No entanto, as classes no segundo e terceiro grupos são alterados, ou seja, adicionando ou removendo herança e implementação de interface. Para uma melhor compreensão, é realizada uma análise qualitativa (semelhante à análise temática (CRUZES; DYBA, 2011)), em uma amostragem destas classes para descobrir a motivação para as mudanças. Essas etapas foram realizadas com 40 mudanças em 40 projetos distintos, para evitar que a mesma equipe realizasse mais de uma mudança. Para cada alteração, foram gerados dois diagramas de classe UML (RUMBAUGH; JACOBSON; BOOCH, 2004). O primeiro diagrama de classe consiste no estado anterior da classe alterada com suas dependências. E o segundo diagrama contém a modificação estrutural após o *commit*. Cada mudança foi codificada, gerando um tema. E então, uma fusão destes temas foi realizada, definindo o nome e temas finais. Segue uma visão detalhada dos passos empregados:

a) Categorização das amostras - Antes de selecionar quais classes foram empregadas para análise, foram estabelecidos quatro conjuntos de amostras para serem coletadas: classes que perderam herança, classes que perderam interface, classes que adicionaram herança e classes que adicionaram interface.

b) Seleção das amostras - Para cada conjunto citado, foram selecionadas dez classes de sistemas diferentes, para garantir que as operações analisadas fossem empregadas por desenvolvedores e arquiteturas de sistema distintos. Com isso, foram analisados um total de quarenta classes de quarenta sistemas distintos.

c) Geração dos dados - para cada classe analisada, foram gerados dois diagramas de classe da *UML* (RUMBAUGH; JACOBSON; BOOCH, 2004). O primeiro diagrama consiste no estado anterior da classe alterada com as suas dependências. E o segundo diagrama contém a classe alterada e as suas dependências após o *commit* que efetuou a modificação na estrutura da classe. Os diagramas contém as relações de herança, implementação de interfaces e composição entre as classes. Também foram extraídas as mensagens de cada *commit*, que contém um texto escrito pelo desenvolvedor descrevendo as alterações. E por fim, foram extraídos os nomes dos métodos adicionados, modificados ou removidos da classe alterada e as dependências que também sofreram alteração no *commit*.

d) Geração e nomeação dos segmentos - para cada classe alterada, foram identificadas e nomeadas todas as atividades ocorridas. Os diagramas fornecem uma visualização do ocorrido, contudo também são gerados segmentos a partir das mensagens do *commit* e dos métodos adicionados e removidos na classe alterada. Por exemplo, houve um caso onde a herança foi removida e a mensagem do *commit* relata que a alteração ocorreu para melhorar a performance do sistema. Então este segmento foi nomeado como *herança removida visando performance* .

e) Geração dos temas - os segmentos foram comparados entre as classes para verificar semelhanças. Desta forma, são agrupadas as classes que resultaram em mudanças com motivação semelhantes. Por exemplo, algumas classes perderam herança com uma classe e começaram a implementar alguma interface. Contudo, em alguns casos, além da implementação, uma nova herança foi criada. Em outros, foi realizada uma composição. Nestes casos, embora existam diferenças no *design* das classes alteradas, foi necessário avaliar se a semântica da alteração é semelhante.

f) Junção entre grupos e definição de nomes - os temas gerados para cada grupo foram analisados para verificar oportunidades de efetuar junção. Possivelmente um *commit* que envolve perda de herança pode adicionar uma interface, logo grupos distintos podem possuir temas em comum.

Capítulo 4

Resultados e Discussão

Neste capítulo, serão apresentados e discutidos os resultados para as perguntas de pesquisa formuladas no capítulo anterior.

4.1 RQ #1 - A Influência da Idade no Uso de Herança e Interfaces

A Tabela 2 apresenta o modelo *NBR* para influência da idade da versão inicial dos sistemas sobre a quantidade de classes com herança e implementando interfaces. A coluna *Estimate* contém os coeficientes de regressão para todas as variáveis preditoras. Para ambos os casos, a variável *MONTHS_RELEASE* é positiva. Com isso, a tendência é que quanto mais antiga seja a versão do sistema, maior será a quantidade de classes com herança e interface, mas sendo uma pequena associação. Os coeficientes são interpretados da seguinte forma: em classes com herança, a variável *CLASSES* possui coeficiente 0,0032. Isso significa que para cada classe adicionada no sistema, o *log* da variável de resposta *INHER* é somado em 0,0032, ou seja, será somado em 1,003 classes ($= e^{0.0032}$). O mesmo vale para a variável preditora *MONTHS_RELEASE*, para cada mês adicionado na idade do sistema, o *log* da quantidade de classes com herança incrementa em 0,002, ou seja, a quantidade de classes com herança aumenta em 1,002 ($= e^{0.002}$).

Tabela 2 – Modelo *NBR* para classes em hierarquia de herança e implementando interfaces

	Estimate	Std. Error	z value	Pr(> |z|)
Herança				
(Intercept)	3,0155966	0,0512405	58.852	< 2e-16 ***
CLASSES	0,0032857	0,0000574	57.241	< 2e-16 ***
MONTHS_RELEASE	0,0020224	0,0005005	4.041	5.32e-05 ***
Interface				
(Intercept)	2,704e+00	5,602e-02	48.266	< 2e-16 ***
CLASSES	2,880e-03	5,553e-05	51.857	< 2e-16 ***
MONTHS_RELEASE	2,240e-03	5,472e-04	4.094	4.25e-05 ***

Após obter os coeficientes, foi verificado se a variável *MONTHS_RELEASE* é estatisticamente relevante para o modelo. Ao executar o teste estatístico *ANOVA*, obteve-se que o valor da probabilidade de *Qui Quadrado* $Pr > X^2 = 6.462469e^{-}05$ para classes com herança e $Pr > X^2 = 6.326497e^{-}05$ para classes implementando interfaces. Sendo assim, aceita-se a suposição que os coeficientes da variável preditora *MONTHS_RELEASE* são maiores que 0. Por fim, foi executado o teste da razão de verossimilhanças, para verificar se as suposições do modelo *NBR* são atendidas. Para o modelo de classes com herança, o *log* de verossimilhança retornou *195.195,1 (df=4)*. Para classes com interface, o valor obtido foi *154.311,3 (df=4)*. O teste de *pchisq* retornou 0 para ambos os casos. Isto sugere que o modelo *NBR* é mais apropriado que o modelo de *Poisson*.

Para visualizar graficamente a influência que cada variável preditora exerce sobre a resposta, as Figuras 12 e 13 apresentam gráficos do tipo *scatter3d* evidenciando os resultados da modelagem *NBR* para classes com herança e interface. Os sistemas foram particionados pelo número de classes para se obter uma melhor visualização. Os resultados mostram que tanto para classes com herança quanto interface, sistemas mais novos com até 500 classes têm apresentado um menor uso de herança e interface. Para sistemas com mais de 500 classes, para herança ainda existe uma redução no uso, contudo com menor inclinação. E para interface, sistemas com mais de 1000 classes apresentaram aumento no seu uso. Sistemas maiores tendem a possuir arquiteturas mais complexas, o que indica a necessidade de maiores abstrações.

Por fim, para comparar o uso de herança e interface, foi obtida a mediana da idade em meses da primeira versão dos sistemas. A mediana possui a data Março-2009. E com isso, os sistemas foram separados em dois grupos, que são anteriores e posteriores à data mencionada. A Figura 14 destaca a proporção do uso de classes com herança e interfaces nos sistemas. Nota-se que até Março-2009, a herança possuía maior intensidade, entre os fatores 0,3 e 0,4, isto é, a maior parte dos sistemas possuem entre 30% e 40% das suas classes com herança. Para interfaces, o fator varia entre 20% e 30%. Contudo, após Março-2009 houve uma queda no uso de herança, e uma ligeira queda no uso de interfaces. Foi executado o teste de *Mann-Whitney U* com o intuito de verificar se as diferenças são significativas entre os sistemas de épocas distintas. Para classes com herança, foi obtido $\alpha = 0.0003814$, e para interface $\alpha = 2.761e^{-}06$. Com isso, para herança os resultados são discretos. Para interface, não existe relevância na redução.

Conclusão RQ #1: Foi confirmada a hipótese de que classes com herança vem sendo menos utilizada nos sistemas mais recentes. Contudo, para sistemas de pequeno e médio porte, a também houve uma redução no uso de interface, o que não era esperado, já que a literatura recomenda o uso de composição com interface. A exceção ocorre em sistemas de grande porte, onde houve crescimento no seu uso.

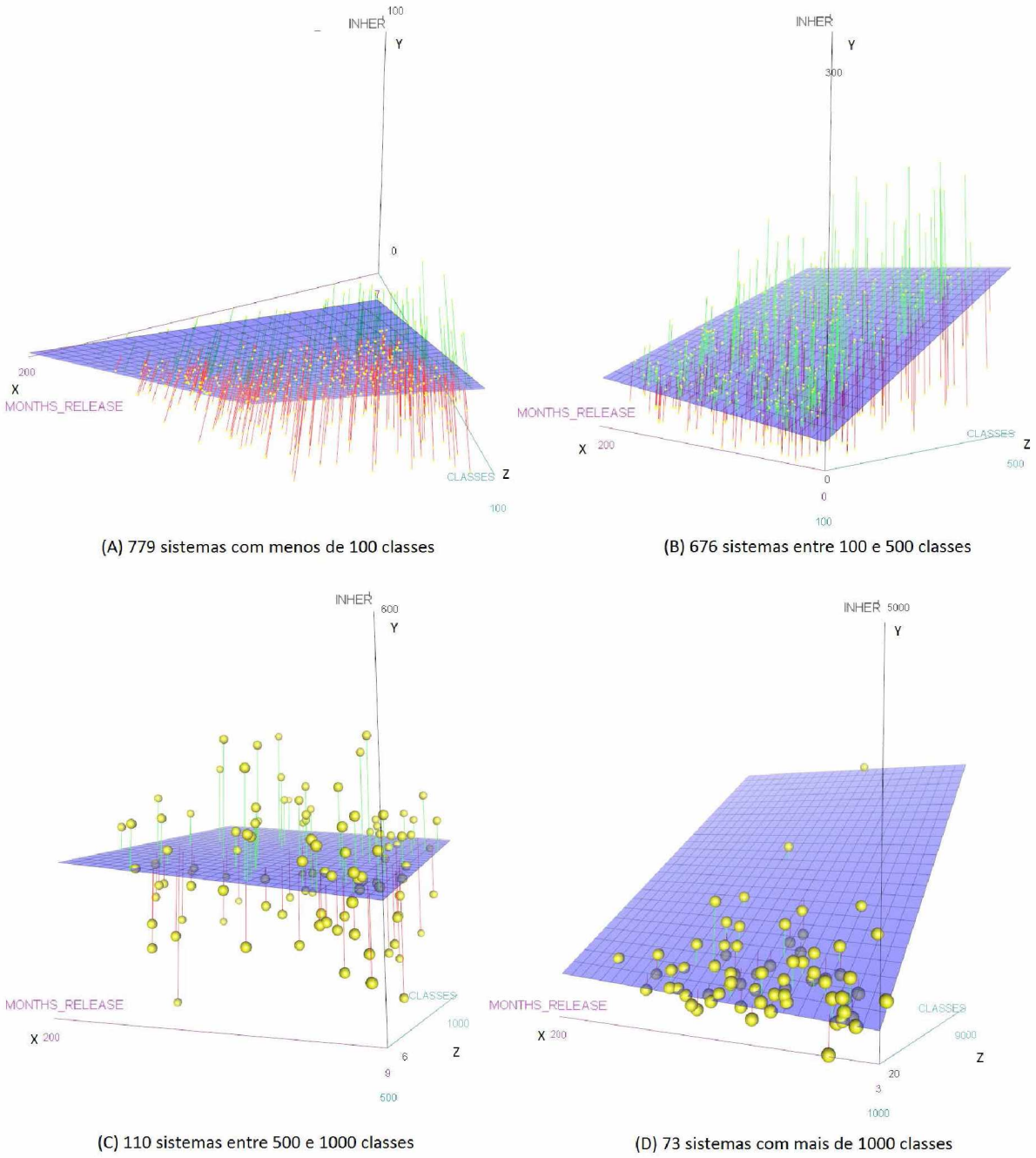


Figura 12 – *Scatter3d* com a influência das variáveis preditoras sobre a quantidade de classes com herança

4.2 RQ #2 - A Influência da Idade nas Quebras de Encapsulamento por *InstanceOf*

A Tabela 3 apresenta o modelo *NBR* para a influência da idade dos sistemas sobre o uso do operador *instanceof* em classes com herança e implementações de interfaces. A coluna *Estimate* mostra que a variável *MONTHS_CREATION* possui coeficiente positivo

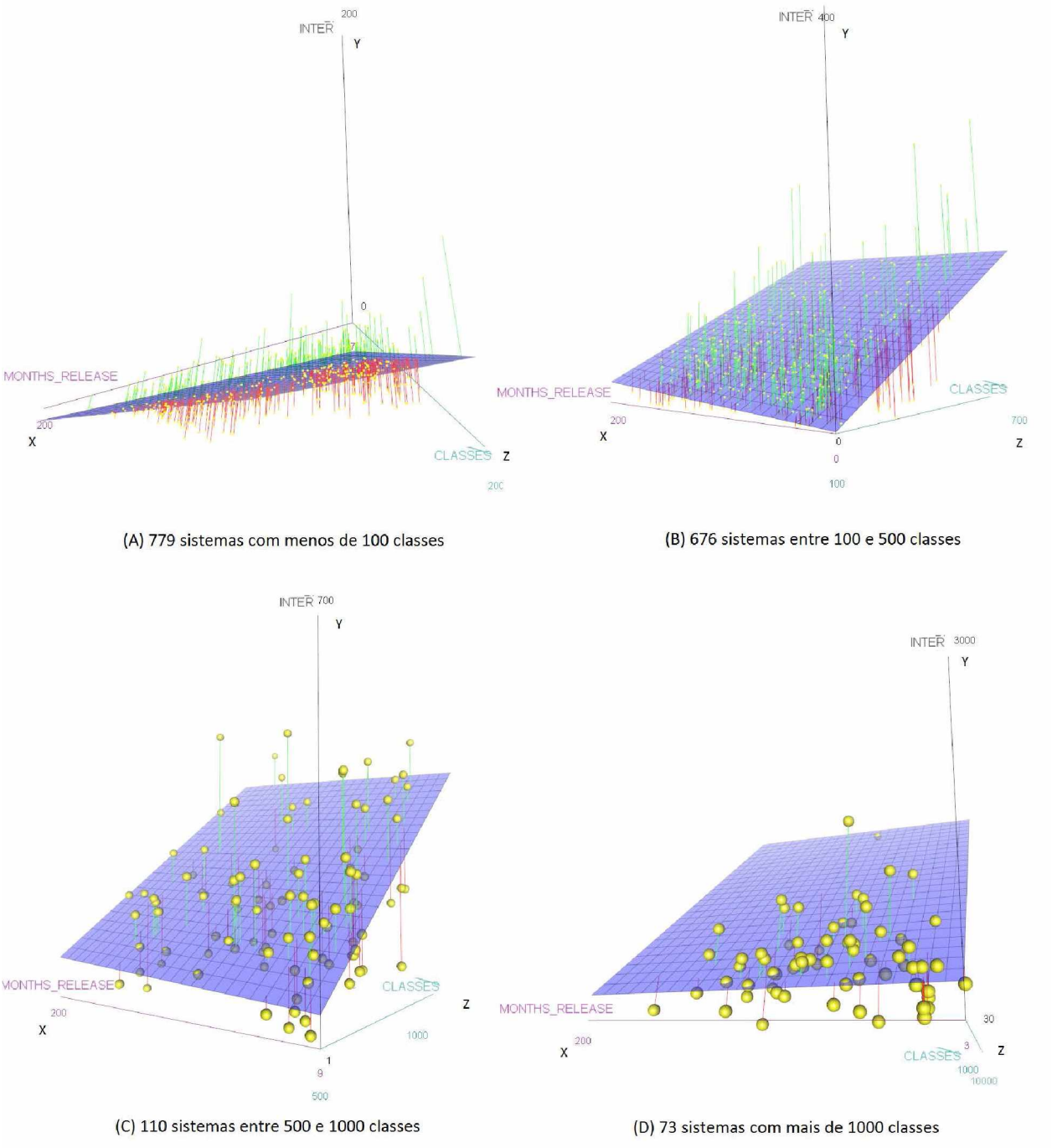
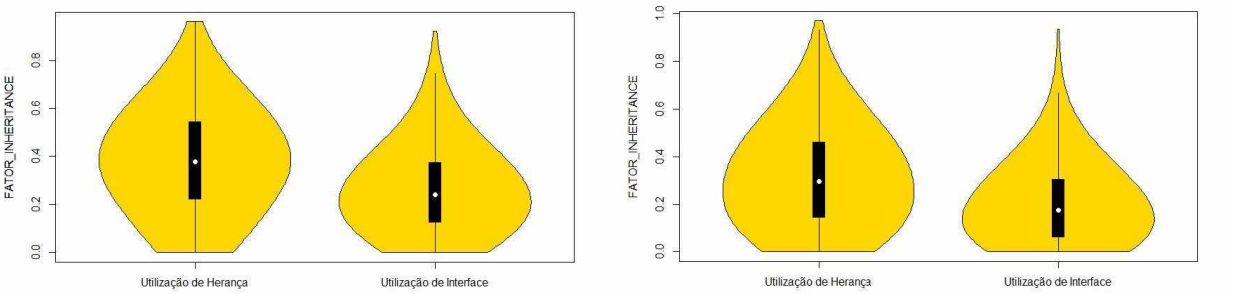


Figura 13 – *Scatter3d* com a influência das variáveis preditoras sobre sobre a quantidade de classes com interface

para ambos os casos, entretanto a associação é pequena. Sendo assim, sistemas mais antigos possuem tendência a estarem discretamente mais associados com o uso do operador *instanceof* para referenciar classes com herança e interface.

Assim como na *RQ #1*, foram executados testes estatísticos para verificar se a variável *MONTHS_CREATION* é estatisticamente relevante para o modelo. Obteve-se *Qui Quadrado* $Pr > X^2 = 2.220446e^{-}16$ para classes com herança e $Pr > X^2 = 0$ para classes



(A) Utilização de herança e interface em sistemas com versão inicial anteriores à mediana

(B) Utilização de herança e interface em sistemas com versão inicial posteriores à mediana

Figura 14 – Gráfico violino que destaca uso de herança e interface em sistemas anteriores e posteriores à mediana

Tabela 3 – Modelo *NBR* para a influência de classes em hierarquia de herança e implementação de interface no uso do operador `instanceof`

	Estimate	Std. Error	z value	Pr(> |z|)
Herança				
(Intercept)	1,059e+00	7,283e-02	14.543	<2e-16 ***
EXTENDS	2,685e-03	6,663e-05	40.304	<2e-16 ***
MONTHS_CREATION	7,282e-03	8,518e-04	8.549	<2e-16 ***
Interface				
(Intercept)	5,139e-01	7,177e-02	7.161	8,02e-13 ***
IMPLEMENTS	4,421e-03	9,317e-05	47.453	< 2e-16 ***
MONTHS_CREATION	8,888e-03	8,278e-04	10.737	< 2e-16 ***

implementando interface, aceitando a suposição que os coeficientes da variável preditora *MONTHS_CREATION* são maiores que 0. Por fim, foi executado o teste da razão de verossimilhanças, para verificar se as suposições do modelo *NBR* são atendidas. Para o modelo de classes com herança, o *log* de verossimilhança retornou *56.230,2 (df=4)*. Para classes implementando interfaces, o valor obtido foi *45.660,21 (df=4)*. O teste de *pchisq* retornou 0 para ambos os casos. Estes sugerem o modelo *NBR* é mais apropriado que o modelo de *Poisson*.

Para visualizar graficamente a influência que cada variável preditora exerce sobre a resposta, foram gerados os gráficos do tipo *scatter3d*, como mostram as Figuras 15 e 16. Em todas as partições, foi observado que sistemas mais recentes tendem a diminuir o uso do operador *instanceof*. Contudo, em classes com interfaces foi observada uma maior inclinação nessa direção, principalmente em sistemas com mais classes.

Por fim, para comparar o uso do operador *instanceof* em classes com herança e interface, foi obtida a mediana da idade em meses referente à data de criação dos sistemas. A mediana possui a data Julho/2008. E com isso, os sistemas foram separados em dois grupos, que são anteriores *(A)* e posteriores *(B)* à data mencionada, como mostra a Figura 17. É possível visualizar que após Julho/2008, houve uma queda no volume de classes

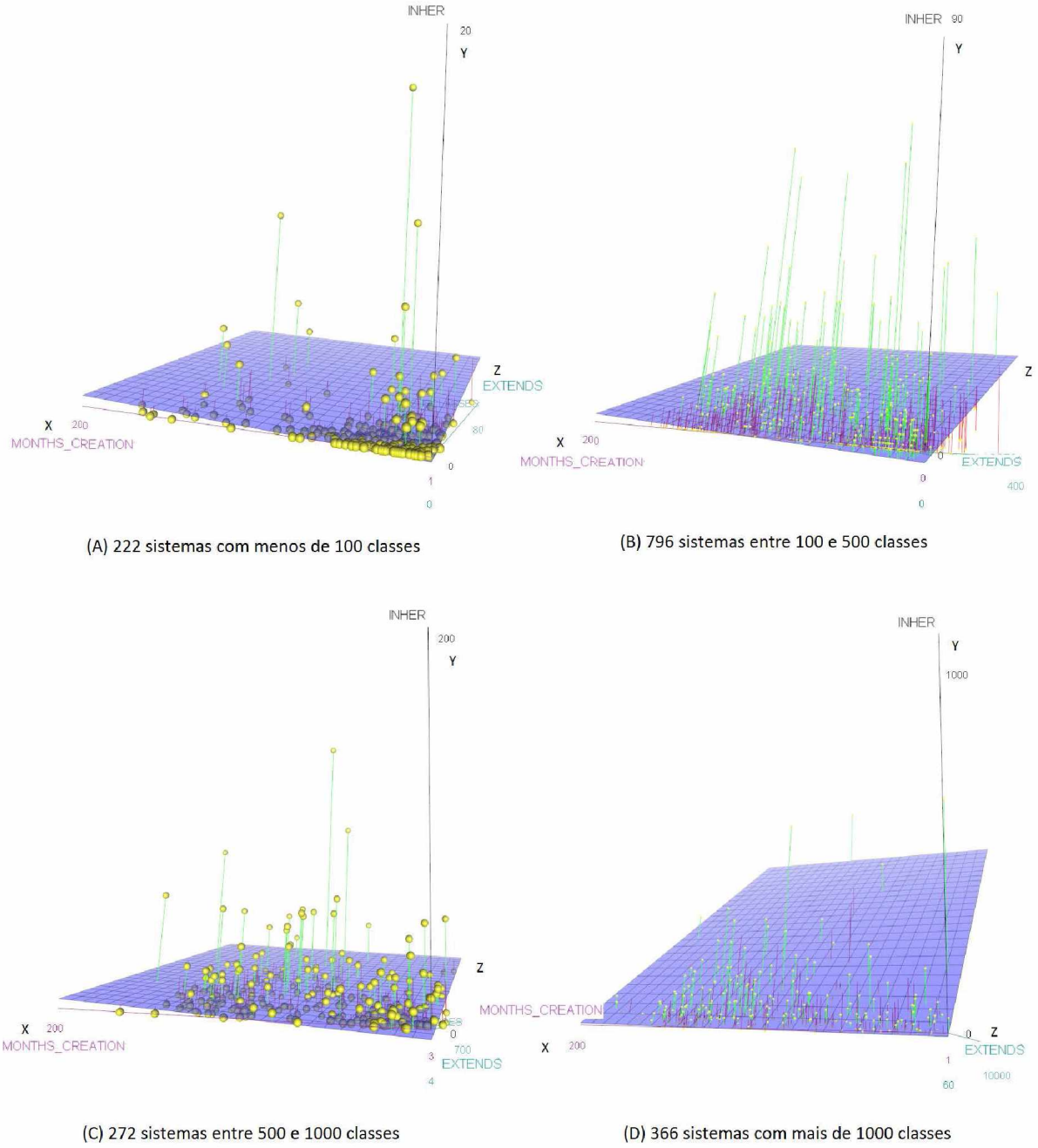


Figura 15 – *Scatter3d* com a influência das variáveis preditoras sobre o uso de *instanceof* em classes com herança

com interface que usam o operador *instanceof*. O teste de *Mann-Whitney U* aponta que existem diferenças significativas, retornando $\alpha < 2.2e-16$ para herança e interface.

Conclusão RQ #2: Foi observado um discreto decréscimo no uso do operador *instanceof* em classes com herança e principalmente implementando interface, que obteve uma decréscimo menos discreto comparado com herança. Também foi constatado que classes que implementam interfaces possuem um pouco mais de quebras de encapsulamento que

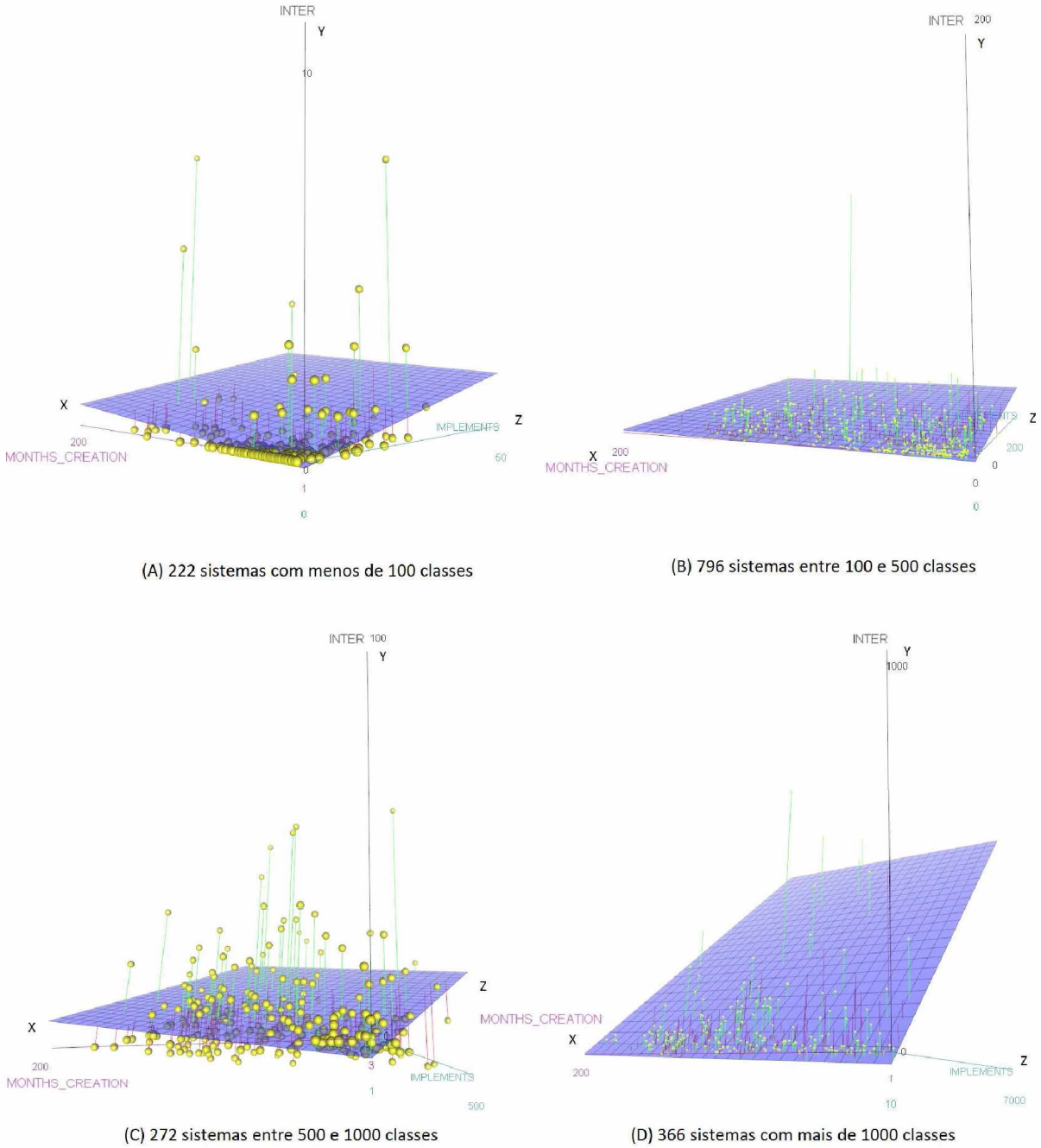
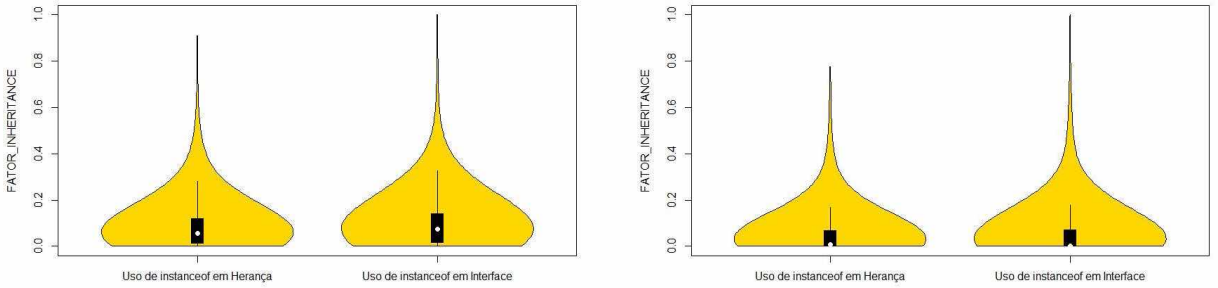


Figura 16 – *Scatter3d* com a influência das variáveis preditoras sobre o uso de *instanceof* em classes com interface

classes com herança.



(A) Uso do operador instanceof em sistemas com data de criação anteriores à mediana

(B) Uso do operador instanceof em sistemas com data de criação posteriores à mediana

Figura 17 – Gráfico violino com o fator de uso do operador *instanceof* em classes com herança e interface

4.3 RQ #3 - A Influência da Idade sobre Alterações Corretivas

A Tabela 4 apresenta o modelo *NBR* para a quantidade de alterações corretivas em classes com herança e implementações de interfaces. A coluna *Estimate* mostra que a variável *MONTHS_CREATION* possui valor positivo para ambos os casos. Para verificar se sistemas mais antigos possuem tendência de possuir mais mudanças corretivas apenas em classes com herança e implementações de interfaces, o mesmo modelo NBR foi aplicado em classes que não estão possuem herança ou implementam interface. O resultado mostra que a variável *MONTHS_CREATION* possui pouca influência e valor negativo para ambos os casos. Isto mostra que sistemas mais antigos possuem uma pequena tendência de ter mais mudanças corretivas nas classes com herança e implementação de interfaces, e sistemas mais novos tendem a possuir mais mudanças corretivas em classes sem herança ou implementação de interfaces.

Ao executar o teste estatístico *ANOVA*, obteve-se que o valor $Pr > X^2 = 0.0016$ para classes com herança e $Pr > X^2 = 5.009351e^{-}06$ para classes implementando interfaces. E $Pr > X^2 = 0.04222498$ for classes sem herança e $Pr > X^2 = 0.03607978$ para classes que não implementam interface. Isso indica que a suposição é aceita que os coeficientes da variável *MONTHS_CREATION* são maiores que 0. Por fim, foi executado o teste da razão de verossimilhanças, para verificar se as suposições do modelo *NBR* são atendidas. Para o modelo de classes com herança, o *log* de verossimilhança retornou *119.697,6 (df=4)*. Para classes com interface, o valor obtido foi *89.185,02 (df=4)*. Estes valores de *chi-quadrado* sugerem o modelo *NBR* é mais apropriado que o modelo de *Poisson*. Para classes sem herança e interface, resultados similares foram obtidos.

As Figuras 18 e 19 apresentam os gráficos do tipo *scatter3d* no mesmo formato das perguntas de pesquisa anteriores. Nota-se em praticamente todas as partições uma leve redução na quantidade de mudanças corretivas, exceto em sistemas de pequeno porte para

Tabela 4 – Modelo *NBR* para a influência de herança e implementação de interfaces em mudanças corretivas

	Estimate	Std. Error	z value	Pr(> |z|)
Herança				
(Intercept)	2.737e+00	4.883e-02	56,048	< 2e-16 ***
FIXING	4.844e-03	8.149e-05	59,450	< 2e-16 ***
MONTHS_CREATION	1.763e-03	5.750e-04	3,067	0.00216 **
Sem Herança				
(Intercept)	3.168e+00	4.086e-02	77.528	<2e-16 ***
FIXING	4.018e-03	6.817e-05	58.933	<2e-16 ***
MONTHS_CREATION	-9.758e-04	4.823e-04	-2.023	0.0431 *
Interface				
(Intercept)	2.357e+00	5.360e-02	43,970	<2e-16 ***
FIXING	4.606e-03	8.917e-05	51,660	<2e-16 ***
MONTHS_CREATION	2.832e-03	6.303e-04	4,494	7e-06 ***
Without Interface				
(Intercept)	3.352e+00	3.971e-02	84.416	<2e-16 ***
FIXING	4.317e-03	6.639e-05	65.032	<2e-16 ***
MONTHS_CREATION	-9.671e-04	4.687e-04	-2.064	0.0391 *

classes com interface, onde a mudança foi mais significativa.

Por fim, para comparar a quantidade de alterações corretivas em classes com herança e interface, foi obtida a mediana já mencionada na *RQ #2*. Na Figura 20, é possível observar que houve uma redução em alterações corretivas, seja para classes com herança ou implementando interface. O teste de *Mann-Whitney U* aponta que existem diferenças significativas retornando $\alpha < 2.2e-16$ para interface e $\alpha = 1.867e-09$ em herança.

Conclusão RQ #3: Em linhas gerais, a quantidade de mudanças têm obtido um leve decréscimo tanto para classes com herança como interface. Isto indica que estão sendo mais fáceis de manter, levando em consideração que menos *bugs* estão surgindo nestas classes.

4.4 RQ #4 - A Influência dos Indicadores de Coesão e Acoplamento

O teste de *Mann-Whitney U* foi aplicado para verificar se existe alguma diferença significativa $\alpha = 0,05$ foi encontrada entre as classes com e sem herança, e que implementam ou não interfaces. Com isso, apenas a métrica *WMC* para classes com e sem herança não retornou diferença significativa, ou seja, $\alpha = 0,1741$.

Além disso, doze gráficos do tipo violino foram gerados, como mostram as Figuras 21 e 22. Estes gráficos auxiliam no entendimento do tamanho do efeito de cada variável. Cada gráfico representa uma métrica, na versão inicial ou final, e os quatro violinos são:

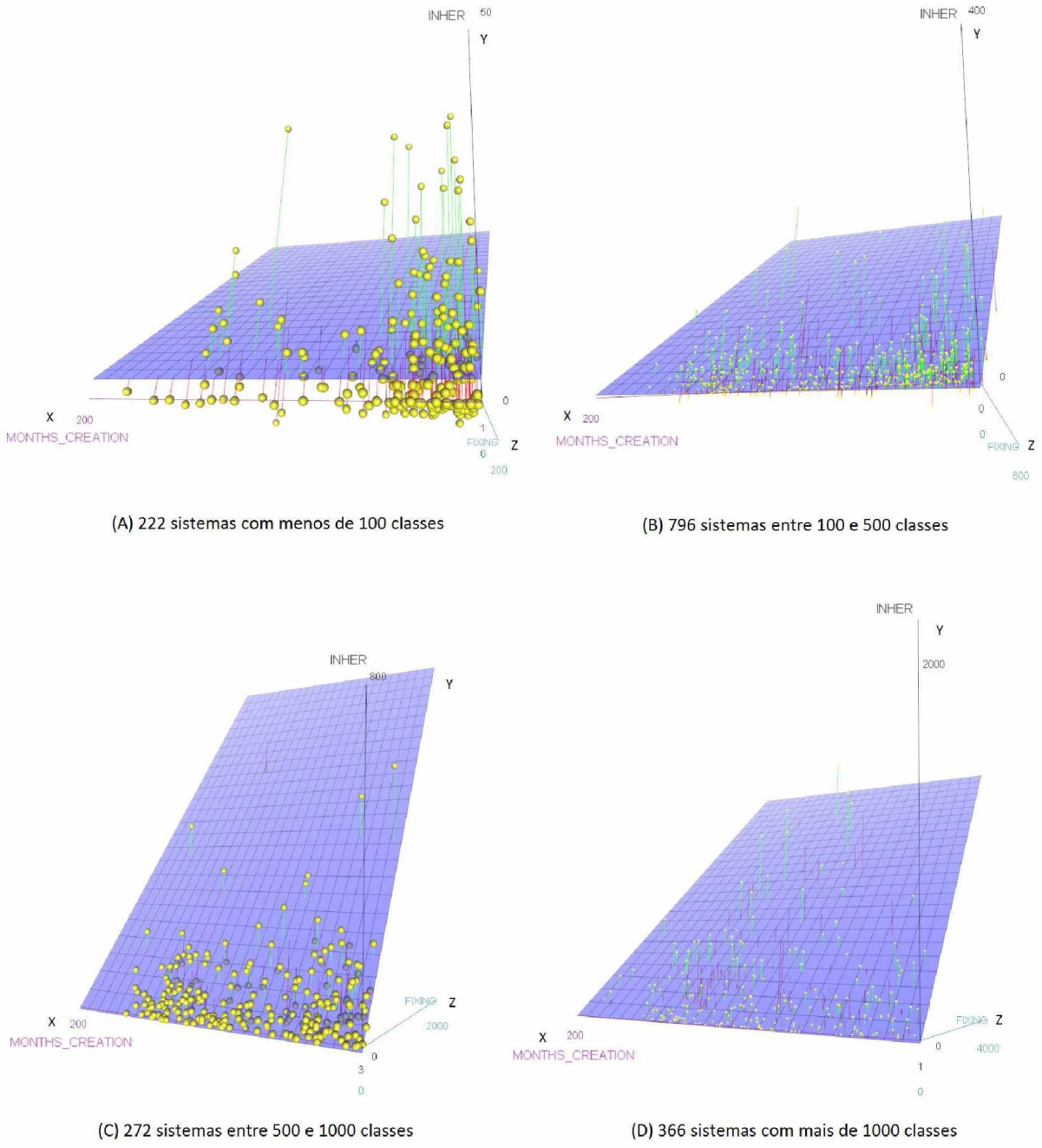


Figura 18 – *Scatter3d* com a influência das variáveis preditoras sobre a quantidade de alterações corretivas em classes com herança

classes com herança, classes sem herança, classes que implementam interfaces e classes que não implementam interfaces.

Para classes com herança, os gráficos violino mostraram alta influência na métrica *LCOM (E),(F)*, ou seja, classes com herança tendem a possuir menor coesão que as demais. Além disso, tendem a ser menores *ELOC (C),(D)*, e possuir menor acoplamento *RFC (I),(J)*.

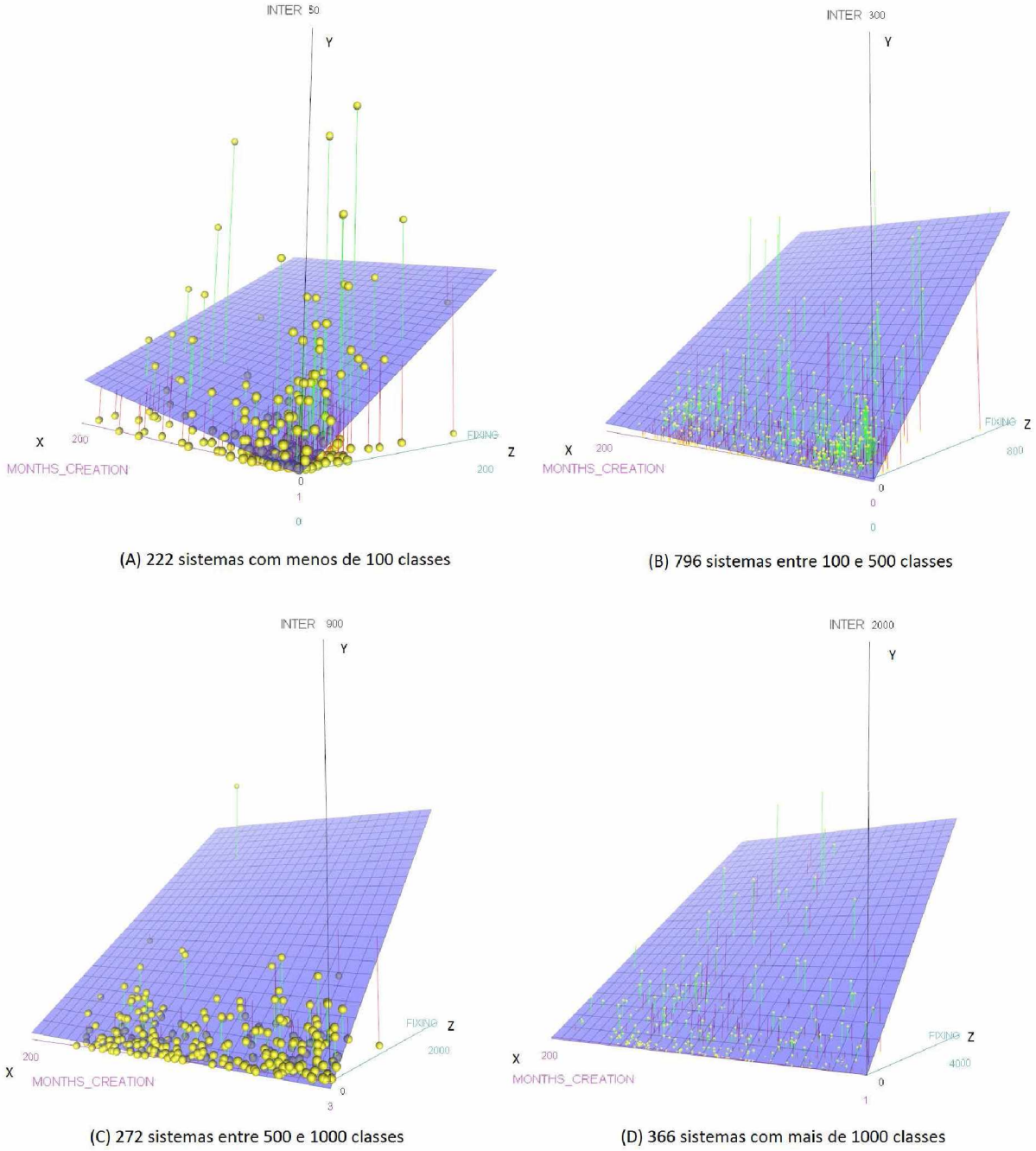


Figura 19 – *Scatter3d* com a influência das variáveis preditoras sobre a quantidade de alterações corretivas em classes com interface

Para classes que implementam interfaces, os gráficos violino mostraram alta coesão se comparado com as demais *LCOM (E),(F)*. Além disso, um menor indicador para *RFC (I),(J)* foi observado, principalmente para classes em versões finais.

Conclusão RQ #4: Existem indícios que classes em herança tendem a possuir mais responsabilidades devido à menor coesão, apesar de possuírem menos linhas de código. Para classes implementando interfaces, as métricas mostraram melhores valores em versões

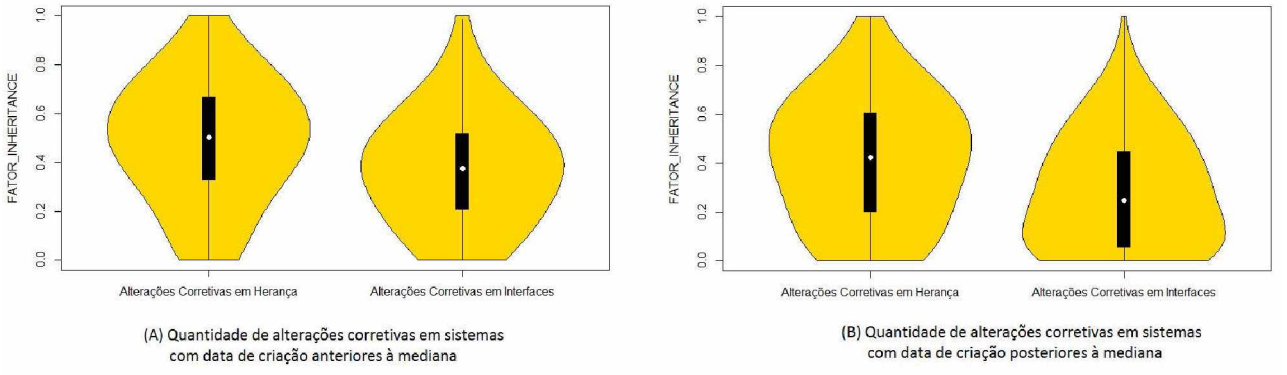


Figura 20 – Gráfico violino com o fator de mudanças corretivas nos sistemas, em classes com herança e interface

finais, indicando que mudanças nestas classes não tendem a alterar seu comportamento.

4.5 RQ #5 - A Influência dos *Code Smells*

As Tabelas 5 e 6 apresentam os resultados encontrados para classes com herança e implementando interfaces nas versões inicial e final de cada sistema. Para $X^2 < 0.05$, existe evidência que os *Code Smells* ocorrem com mais frequência em um cenário do que outro (com herança ou sem herança, com interface ou sem interface). Caso contrário, os *Code Smells* ocorrem na mesma proporção em ambos os cenários. Assim, para classes com herança, os *Code Smells Lazy Class e Complex Class* não possuem efeito na versão inicial, mas na final existe uma evidência de que esses *Code Smells* ocorrem mais nas classes com herança. Esse resultado indica que as classes com herança tendem a absorver mais funcionalidades durante seu ciclo de vida, e algumas dessas classes não têm muitas funções. Os resultados em classes que implementam interfaces mostram a mesma proporção na versão inicial e final para todos os *Code Smells*.

Conclusão RQ #5: Para as classes com herança, há uma evidência de que os *Code Smells Lazy Class e Complex Class* ocorrem durante as mudanças das classes. Mas para as classes que implementam interfaces, não foram encontradas alterações na predominância de *Code Smells* encontrados. Isso indica que pode haver uma relação fraca entre a interface e suas implementações de classes.

4.6 RQ #6 - Frequência e Motivação para Adição e Remoção de Herança e Interface

A Tabela 7 apresenta os resultados para a quantidade de classes em que houve adição ou remoção de herança e implementação de interface, separados em quatro grupos. Cada

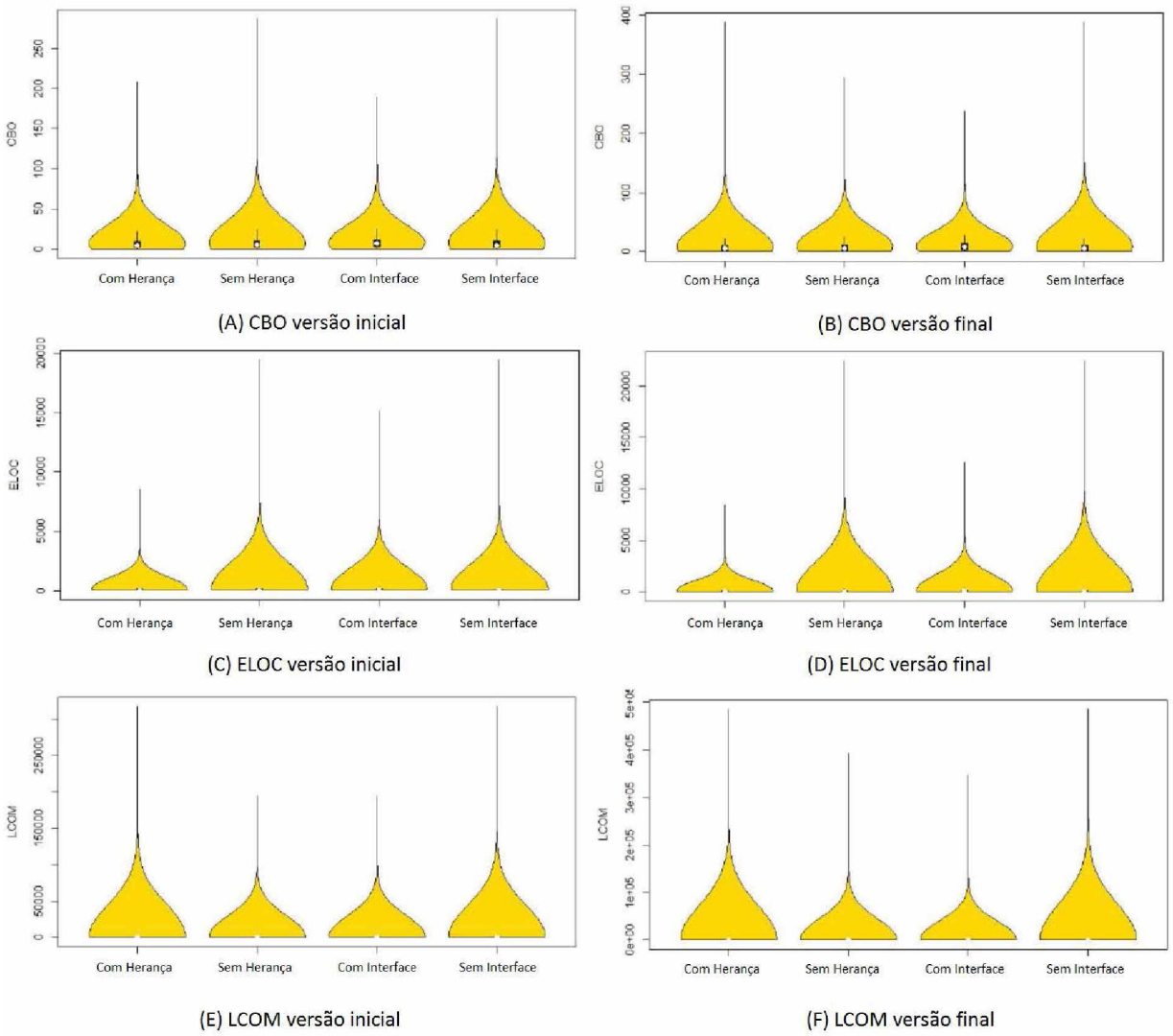


Figura 21 – Gráfico violino para as métricas *CBO,ELOC* e *LCOM* de coesão e acoplamento coletadas nas versões inicial e final dos sistemas

grupo possui duas colunas, onde a coluna *Ult* representa a quantidade e o percentual de classes que persistiram até a versão final do sistema, e a coluna *Del* que representa as classes que foram excluídas em algum *commit* do sistema. O resultado mostra que 11,33% das classes foram excluídas do sistema em algum *commit*. O grupo *Sempre* mostra a quantidade de classes que foram construídas com herança ou interface, e assim permaneceram até a versão final, ou até serem excluídas do sistema. Os resultados mostram que 37,11% das classes foram criadas usando herança e assim permaneceram até a versão final do sistema. Para implementação, são 17,91% das classes. Também é mostrado que cerca de 12% das classes que sempre possuíram herança foram excluídas dos sistemas. Este índice está coerente com a média geral. Isso indica que classes construídas com herança ou implementação de interface não possuem maior tendência de exclusão. O grupo *nunca* mostra a quantidade de classes que nunca adquiriram herança ou implementação de in-

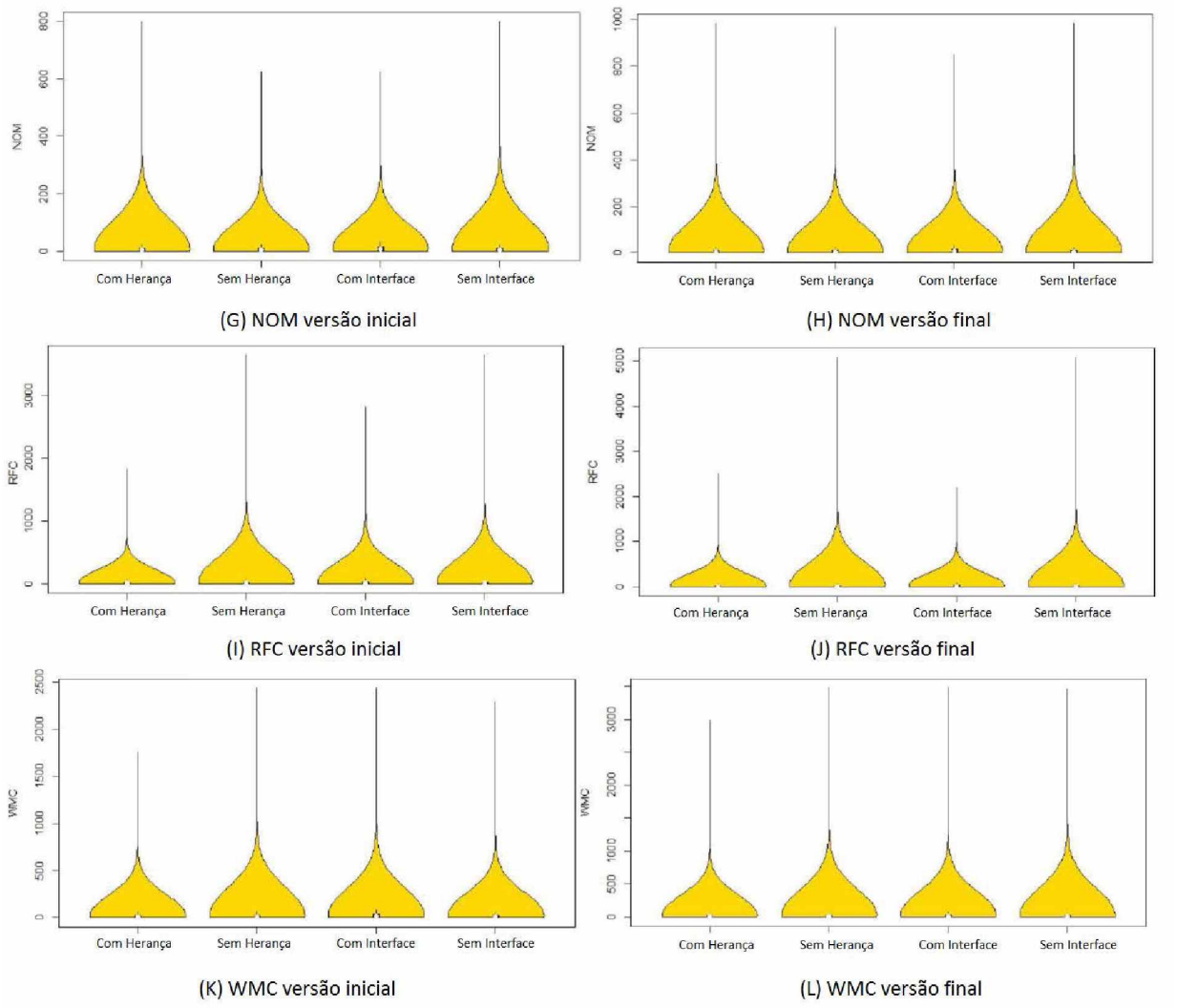


Figura 22 – Gráfico violino para as métricas *NOM,RFC* e *WMC* de coesão e acoplamento coletadas nas versões inicial e final dos sistemas

terfaces até a sua exclusão, ou até a versão final do sistema. Destaca-se que 60,24% das classes existentes na versão final dos sistemas nunca possuiram herança, e 79,68% nunca implementaram interface.

Os resultados dos grupos mencionados mostram que, durante o ciclo de vida das classes, não ocorreram operações de adição ou remoção de herança e implementação em classes já existentes. Com isso, a maioria permaneceu até a sua exclusão ou versão final dos sistemas dentro da arquitetura que foram construídas. As poucas classes que foram modificadas se encontram nos grupos *perdeu* e *depois*. O grupo *perdeu* destaca as classes que possuíam herança ou implementavam interfaces, mas perderam em algum *commit*. Com isso, 0,5% das classes possuíam herança e perderam até a versão final, e 1,09% para implementação. E menos de 10% classes que perderam herança ou implementação são excluídas. Isso indica que a maior parte destas classes ainda possuem funcionalidade para

Tabela 5 – Tabela de contingência para a versão inicial dos sistemas, com cada code smell para classes com hierarquia de herança ou implementando interfaces

	Com C.D.S.Private	Sem C.D.S.Private	X^2 p-value
Com Herança	1,051	105,749	< 2.2e-16
Sem Herança	2,581	141,707	
Com Interface	408	45,592	< 2.2e-16
Sem Interface	3,222	201,740	
	Com Complex Class	Sem Complex Class	X^2 p-value
Com Herança	813	106,447	0.1346
Sem Herança	1,022	143,754	
Com Interface	803	45,781	< 2.2e-16
Sem Interface	1,033	204,298	
	Com Functional D.	Sem Functional D.	X^2 p-value
Com Herança	300	106,398	0.5331
Sem Herança	385	143,695	
Com Interface	85	45,777	8.09e-05
Sem Interface	600	200,194	
	Com God Class	Sem God Class	X^2 p-value
Com Herança	3,067	105,834	3.557e-12
Sem Herança	4,867	142,703	
Com Interface	2,672	42,259	< 2.2e-16
Sem Interface	5,252	203,164	
	Com Lazy Class	Sem Lazy Class	X^2 p-value
Com Herança	17,809	89,071	0.8616
Sem Herança	24,001	120,274	
Com Interface	4,788	41,094	< 2.2e-16
Sem Interface	36,999	168,168	
	Com Long Method	Sem Long Method	X^2 p-value
Com Herança	978	105,968	1.826e-12
Sem Herança	1,750	142,927	
Com Interface	626	45,564	6.058e-10
Sem Interface	2,102	203,209	
	Com Spaghetti Code	Sem Spaghetti Code	X^2 p-value
Com Herança	2,294	106,444	1.54e-09
Sem Herança	3,645	143,692	
Com Interface	2,053	45,776	< 2.2e-16
Sem Interface	3,879	204,238	

os sistemas. Outro fator interessante é que, somando apenas as classes que adquiriram herança, apenas 1,45% destas perderam a herança sem exclusão imediata (no mesmo *commit*). O grupo *depois* apresenta classes que não foram construídas com herança ou implementação de interfaces, mas em algum momento adquiriram tal funcionalidade. O percentual mostrado é de 2,07% para herança e 1,31% para implementação de interface. Este índice é superior ao percentual das classes que perderam herança ou implementação de interfaces. Quando um comportamento é removido, as dependências podem ser afetadas, possibilitando o aumentando a quantidade de classes ou interfaces alteradas em um único *commit*. Contudo, ao adicionar um comportamento, novas dependências são criadas ao longo dos *commits*. Logo, a tendência é que adicionar herança ou implementação de interfaces seja uma atividade menos complexa do que a sua remoção. Isso justifica a diferença no percentual.

Tabela 6 – Tabela de contingência para a versão final dos sistemas, com cada code smell para classes com hierarquia de herança ou implementando interfaces

	Com C.D.S.Private	Sem C.D.S.Private	X^2 p-value
Com Herança	1,567	164,130	< 2.2e-16
Sem Herança	3,720	209,206	
Com Interface	696	70,114	< 2.2e-16
Sem Interface	4,649	303,391	
	Com Complex Class	Sem Complex Class	X^2 p-value
Com Herança	1,297	165,150	5.592e-08
Sem Herança	1,346	211,960	
Com Interface	1,067	70,449	< 2.2e-16
Sem Interface	1,599	307,064	
	Com Functional D.	Sem Functional D.	X^2 p-value
Com Herança	460	165,091	0.5603
Sem Herança	568	211,864	
Com Interface	137	70,442	1.364e-05
Sem Interface	891	306,917	
	Com God Class	Sem God Class	X^2 p-value
Com Herança	4,975	164,106	0.0009353
Sem Herança	6,792	210,388	
Com Interface	3,754	69,663	< 2.2e-16
Sem Interface	8,062	305,228	
	Com Lazy Class	Sem Lazy Class	X^2 p-value
Com Herança	27,664	138,152	0.04209
Sem Herança	34,957	177,734	
Com Interface	6,402	64,208	< 2.2e-16
Sem Interface	56,189	252,121	
	Com Long Method	Sem Long Method	X^2 p-value
Com Herança	1,720	164,293	3.757e-13
Sem Herança	2,760	210,633	
Com Interface	1,116	70,039	< 2.2e-16
Sem Interface	3,367	305,292	
	Com Spaghetti Code	Sem Spaghetti Code	X^2 p-value
Com Herança	3,692	165,125	0.005518
Sem Herança	5,035	211,855	
Com Interface	1,116	70,039	< 2.2e-16
Sem Interface	3,367	305,292	

Tabela 7 – Quantidade de adições e remoções de herança e interface nas classes dos sistemas analisados

Tipo	**Sempre**		**Perdeu**		**Depois**		**Nunca**		**Total**	
	Ult	**Del**	**Ult**	**Del**	**Ult**	**Del**	**Ult**	**Del**	**Ult**	**Del**
Herança	384.933	46.950	5.875	480	21.510	1827	624.860	83.371	1.037.178	132.628
	(37,11%)	(35,39%)	(0,5%)	(0,3%)	(2,07%)	(1,37%)	(60,24%)	(62,86%)	(88,66%)	(11,33%)
Interface	185.763	23.226	11.372	814	13.610	1.043	826.433	107.545	1.037.178	132.628
	(17,91%)	(17,51%)	(1,09%)	(0,61%)	(1,31%)	(0,78%)	(79,68%)	(81,08%)	(88,66%)	(11,33%)

Ao concluir a análise da Tabela 7, dois questionamentos ainda persistem: o primeiro consiste em compreender as razões pelas quais uma classe perde herança ou implementação de interfaces, principalmente em situações onde a classe não é excluída do sistema. Presume-se que quando a classe é excluída, tal funcionalidade foi substituída ou não possui mais utilidade. Contudo sem a exclusão da classe, a funcionalidade continua servindo ao sistema, mas de outra forma. O segundo questionamento consiste em entender os mo-

tivos pelos quais a classe é acrescentada posteriormente com herança ou implementação de interfaces. Para isso, foi realizada uma análise temática sobre as classes, e cada tema encontrado será apresentado como uma subseção.

4.6.1 Abstrações Incertas

Este tema foi detectado em 15 classes (37,5%), e apresenta situações onde a superclasse ou interface foram construídos inicialmente com poucas funcionalidades sendo oferecidas às subclasses. Contudo, foi observado que ao longo das alterações que ocorreram no sistema, estas superclasses ou interfaces não obtiveram o incremento de funcionalidades que justificasse sua existência, e foram excluídas ou permaneceram sem subclasses.

Como exemplo, o sistema JSourcePad [1] possui a classe kkckkc.syntaxpane.model.Scope que herda de kkckkc.syntaxpane.model.Interval. Contudo, a subclasse necessitava apenas de dois atributos da superclasse, sendo que os demais comportamentos não são necessários. Sendo assim, em um determinado *commit*, a subclasse foi desacoplada criando os atributos já existentes na classe Interval, contudo com apenas um método replicado da superclasse. A mensagem do *commit* menciona *reduction of memory usage*, mostrando que neste caso foi melhor evitar criar mais objetos desnecessariamente em tempo de execução, em detrimento do reaproveitamento de código proporcionado pela herança. Outro exemplo com herança ocorreu no sistema tiled [2], onde a classe tiled.mapeditor.undo.MapLayerStateEdit herdava funcionalidades de MapLayerEdit, que herda de javax.swing.undo.AbstractUndoableEdit. Contudo, foi observado que a classe MapLayerStateEdit depende mais das funcionalidades de AbstractUndoableEdit do que MapLayerEdit, logo a modificação foi realizada.

Em alguns exemplos, a superclasse é excluída, logo a subclasse recebe os comportamentos da classe excluída, e consequentemente a herança é desfeita. Por exemplo, no sistema Forge Essentials [3], a superclasse Com.ForgeEssentials.ConsoleInfo foi excluída, e a subclasse PlayerInfo copiou suas funcionalidades. Neste caso, não havia necessidade de existirem duas classes.

Para interfaces, este cenário ocorreu por exemplo no sistema gobandroid [4], como mostra a Figura 23. A interface org.ligi.gobandroid.logic.GoDefinitions.java possuía constantes que eram usadas pelas subclasses GoGame, GoMove e GoBoard, todas do mesmo pacote (A). Contudo, foi necessário adicionar o método getHandicapArray() que estava definido na classe GoGame. Como as classes não dependiam de funcionalidades da interface, o método foi movido para a interface, que foi convertida em classe (B). Contudo, a nova classe GoDefinitions adquiriu o *Code Smell Lazy Class*.

[1] https://github.com/kkckkc/jsourcepad/commit/a20e2f9115625c89a11e1472f235766f1f10d976
[2] https://github.com/bjorn/tiled-java/commit/093afd9107edad64c85bc90244586fd46626db61
[3] https://github.com/ForgeEssentials/ForgeEssentials/commit/b530da9d5e11322596818eef1bb40c233a8fedc9
[4] https://github.com/ligi/gobandroid/commit/6b2eb530bbf854e71c1f0b8186667d561456d076

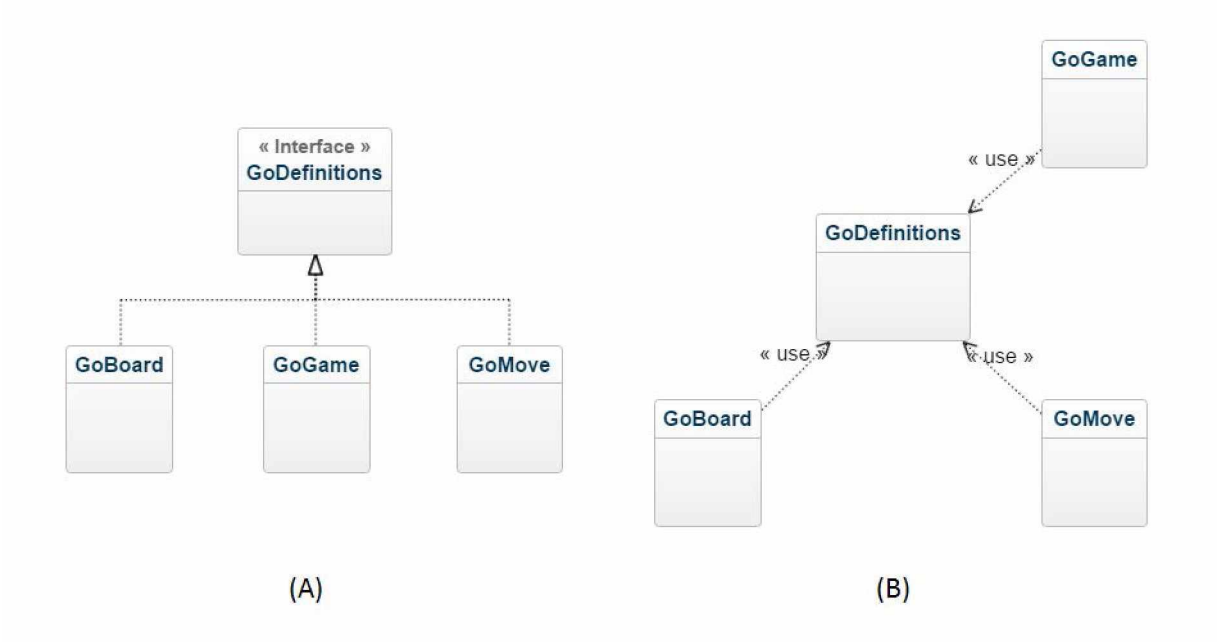


Figura 23 – Diagrama de classes ilustrando uma modificação ocorrida no sistema GOBANDROID

4.6.2 Comportamento Padrão para as Interfaces

Este tema foi detectado em 9 classes (22,5%). Ocorre quando as classes que implementam uma interface necessitam de um comportamento que seja comum a estas. Desta forma, para evitar duplicidade de código-fonte, em todos os casos foi construída uma superclasse com esta funcionalidade. A nova superclasse implementa a interface que necessita do comportamento, e as classes que antes implementavam a interface, começam a estender da nova superclasse.

Por exemplo, a Figura 24 apresenta uma alteração realizada no sistema PLUGIN GRADLE FOR NETBEANS [5]. A parte *(A)* evidencia que as classes MEMPROJECTPROPERTIES e PROJECTPROPERTIESPROXY, ambas do pacote ORG.NETBEANS.GRADLE.PROJECT.PROPERTIES implementavam a interface PROJECTPROPERTIES. Contudo, foi necessário adicionar um comportamento que é comum às duas classes, por isso as subclasses deixaram de implementar diretamente a interface para estender a nova classe ABSTRACTPROJECTPROPERTIES, como é mostrado em *(B)*. E a superclasse passou a estender a interface PROJECTPROPERTIES. Desta forma, as subclasses continuam implementando os métodos definidos pela interface, mas o seu comportamento comum é colocado em uma superclasse. Este comportamento também foi observado no sistema DCACHE [6], onde três interfaces adquiriram um comportamento padrão com a construção da superclasse ABSTRACTCELLCOMPONENT.

[5] https://github.com/kelemen/netbeans-gradle-project/commit/de69efbffbf5d30ae8649a111f851ece52695a8a
[6] https://github.com/dCache/dcache/commit/c30369df7d3eb33135f21157a1eaaea320caf3eb

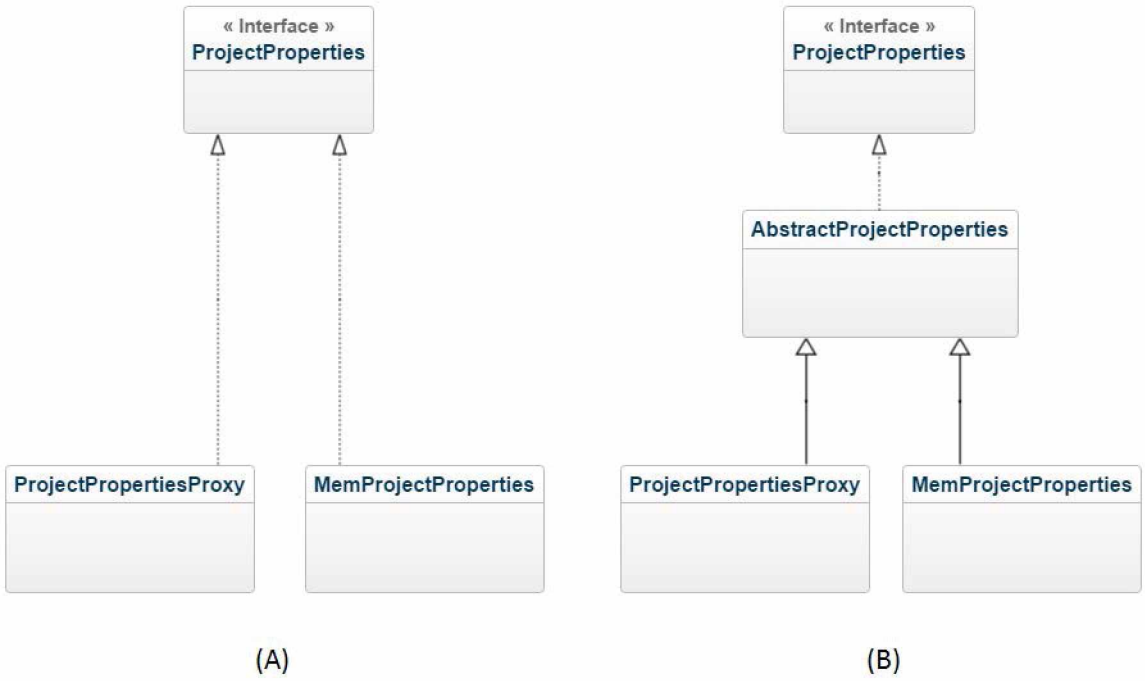


Figura 24 – Diagrama de classes ilustrando uma modificação ocorrida no sistema PLUGIN GRADLE FOR NETBEANS

Outras variações deste comportamento foram detectadas, como é o caso do sistema ECLIPSE BPEL [7]. As classes WSDLIMPORTRESOLVER e XSDIMPORTRESOLVER, ambas do pacote ORG.ECLIPSE.BPEL.MODEL.UTIL estendiam a interface IMPORTRESOLVER. Contudo, a classe WSDLIMPORTRESOLVER possui duplicidades com XSDIMPORTRESOLVER, logo foi constatado que a primeira classe apenas especializava o comportamento da segunda. Com isso, XSDIMPORTRESOLVER foi promovida a superclasse de WSDLIMPORTRESOLVER, e a sua implementação se manteve, ao contrário da classe WSDLIMPORTRESOLVER.

Uma observação é que a partir da versão 8, a sintaxe de Java permite que as interfaces possuam um método *default* [8]. Desta forma, tais superclasses podem ser substituídas.

4.6.3 Novas Funcionalidades com Adoção de Boas Práticas

Em 7 casos observados (17,5%), foi constatado que a alteração de um requisito foi acompanhada de um incremento na qualidade do *design* das classes, observando boas práticas recomendadas na literatura. Em todos os casos observados, o *commit* não foi realizado com o objetivo de implementar boas práticas. A motivação consistiu na alteração

[7] https://github.com/eclipse/bpel/commit/90f3bb67b9cff68f70fa486ff814a8374b21f913
[8] https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

nos requisitos do sistema, contudo foram implementados de forma extensível, observando boas práticas de *POO*.

Como primeiro exemplo, foi detectada uma alteração de herança para composição. A Figura 25 mostra o diagrama de classes com a alteração realizada no sistema jDTO-Binder [9]. Neste caso, a classe com.juancavallotti.jdto.spring.SpringDTOBinder herdava de DTOBinderBean, que implementa a interface DTOBinder, como é mostrado em *(A)*. Para evitar perder as funcionalidades de DTOBinderBean sem quebrar o encapsulamento da classe, esta passou a possuir uma instância de DTOBinderBean, implementando também a interface DTOBinder, como mostra em *(B)*.

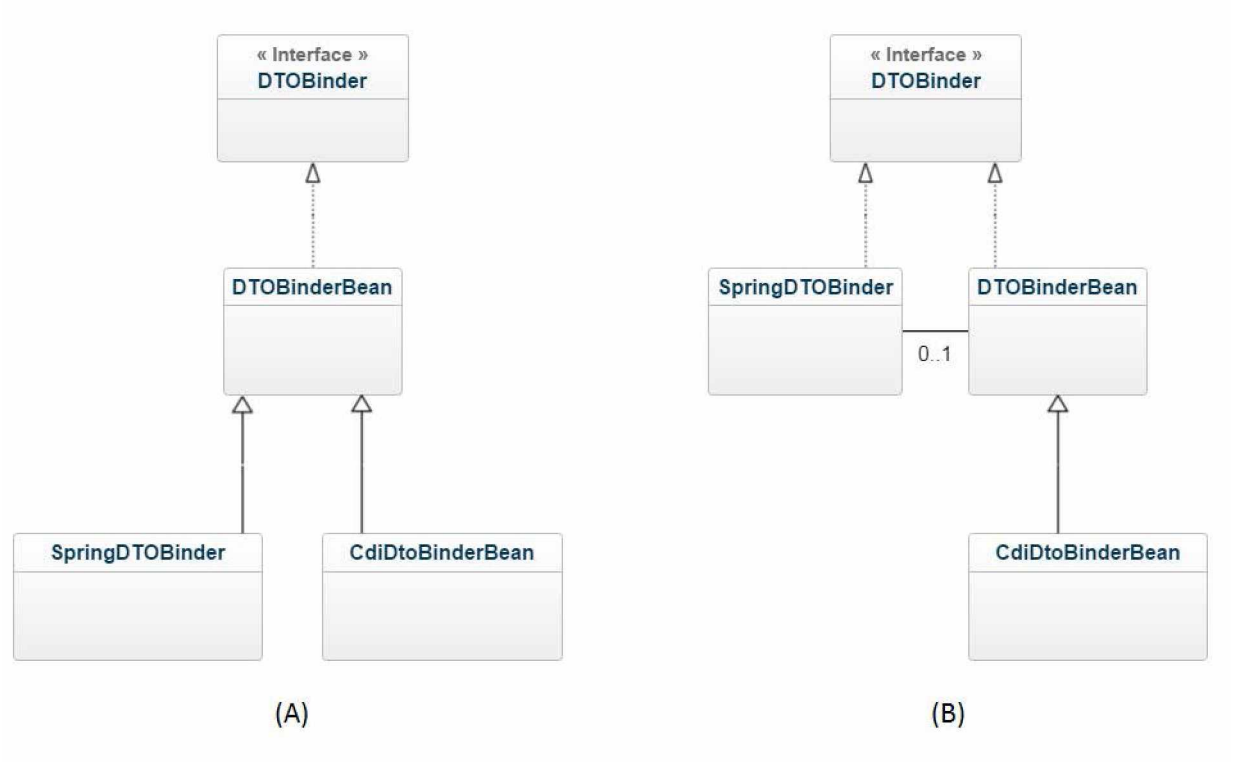


Figura 25 – Diagrama de classes alteradas no sistema jDTO-Binder para favorecer o uso de composição

Outro exemplo ocorreu no sistema Eclipse Platform Team [10] onde houve uma modificação para obter maior polimorfismo e menor acoplamento. O sistema necessitou que alguns comandos fossem interpretados por classes já existentes no sistema, logo foi criada uma interface org.eclipse.team.tests.ccvs.core.ICVSClient, e as classes CommandLineCVSClient e EclipseCVSClient começaram a implementá-la. E a classe SameResultEnv, que antes possuía acoplamento direto com as classes para consumir seus serviços, foi modificada para se acoplar com a nova interface, implementando o princípio *Programe para Interface e não para Implementação*. Diante disso, foi

[9] https://github.com/jDTOBinder/jDTO-Binder/commit/8f54c719a8f31fdb5597ee4cffbe38839613dd89
[10] https://github.com/eclipse/eclipse.platform.team/commit/0d58b5e568e6c3f3e96506ec6d5f4431885acb04

definido o método *execute(ICVSCLient)* obtendo polimorficamente uma subclasse desta interface. Outro cenário praticamente idêntico ao citado ocorreu no sistema POLLY [11], onde a classe POLLY.RX.ENTITIES.SCOREBOARDENTRY não foi prevista para exportar seu conteúdo para *CSV*. Como tal funcionalidade se tornou necessária, esta passou a implementar a interface DE.SKUZZLE.POLLY.SDK.CSVEXPORTABLE e chamada polimorficamente na classe CSVEXPORTER.

Por fim, também ocorreram casos onde, ao adicionar um comportamento em uma superclasse pouco coesa, uma operação de *refactoring move method* foi realizada para separar os comportamentos da superclasse em novas classes. Um exemplo ocorre no sistema DMDIRC [12], onde a classe COM.DMDIRC.SERVER.JAVA teve acúmulo de responsabilidades, logo suas implementações de interfaces foram transferidas para uma nova classe chamada COM.DMDIRC.SERVEREVENTHANDLER

4.6.4 Outros Temas

Ocorreram 3 casos de perda de herança de superclasses internas, para estender bibliotecas externas. Em outros 2 casos foram detectados uma operação de *refactoring move package*, que não é prevista de detecção neste trabalho, e citada no final deste capítulo como ameaça à validade. E por fim, 4 casos onde houve uma quantidade muito grande de classes alteradas, com diversas operações sendo realizadas. Nestes casos, não foi identificado uma causa específica.

Conclusão RQ #6: Poucas classes adicionaram ou removeram herança após a sua construção. Quando ocorre, é motivada pelos novos requisitos que são adicionados no sistema. A herança é desfeita principalmente porque a superclasse foi projetada para atender possíveis novas funcionalidades que nunca surgiram no sistema, logo estas superclasses são criadas como *Lazy Class* e assim permanecem até a herança ser removida. Não foram observadas heranças que foram desfeitas para efetuar *refactoring* em classes do tipo *God Class*. Hierarquias de interface são desfeitas principalmente para evitar duplicidade de código-fonte entre as classes que as implementam. A modificação para adição de herança também predominou para evitar duplicidade, reaproveitando código-fonte. E por fim, a adição de interface foi observada principalmente em razão de se obter polimorfismo, usando de boas práticas.

4.7 Ameaças à Validade

Uma ameaça à validade externa ocorre porque os resultados encontrados não podem ser generalizados para outras linguagens orientadas a objetos, porque todos os projetos foram escritos em Java. Para generalizar este estudo, é necessário avaliar projetos de

[11] https://github.com/skuzzle/polly/commit/6402e3b89c8b234de89d15ffe93171dc27adb8bf
[12] https://github.com/DMDirc/DMDirc/commit/8378135048c2b9c330d62e030d300376be397eda

outras linguagens orientadas a objetos. Além disso, apenas os projetos open source foram considerados. No entanto, o conjunto de dados de grande escala fornece uma amostra representativa de projetos Java open source.

Outra ameaça à validade deste trabalho diz respeito às *tags*. Embora o trabalho mostre claramente que houve um processo de higienização das *tags*, descartando nomes que não condizem com versões do sistema, não existem garantias que a *tag* selecionada realmente represente uma versão do sistema, ou inclusive que seja de fato a versão inicial ou final. Alguns sistemas possuem *tags* com nomes que remetem a uma versão do sistema. Por exemplo, a primeira *tag* do sistema *nsuml* [13] possui nome *release0_0_1*, que induz a uma versão do sistema. Contudo, a primera *tag* do sistema *gzigzag* [14] possui nome *snapshot-2000-07-11T12_26_00Z*, que não diz claramente se é uma versão ou um marco do sistema. Com isso, foram descartadas apenas *tags* que claramente possuem nomes incompatíveis com versões do sistema.

Outro aspecto sobre as *tags* pode ser visto nas tabelas 7 e 6. A primeira tabela citada obtém do *BOA* todas as classes, totalizando 1.037.178 classes que permaneceram até o último *commit* dos sistemas. Na segunda tabela citada, é mostrado o total de classes da versão final dos sistemas, que totaliza *378.632* classes, um número consideravelmente inferior ao da primeira tabela citada. Contudo, ao receber um conjunto de *tags* de um sistema, não existem garantias que as *tags* de fato contemplem todas as classes do sistema, ou que exista *tag* marcada para a versão final em ambiente de produção. De qualquer forma, foram computadas 259.565 classes para a versão inicial dos sistemas, ou seja, ao comparar as duas versões, existe um crescimento de funcionalidades nos sistemas em geral.

Uma outra ameaça à validade é que este trabalho desconsiderou operações de *refactoring* como *move package* ou *rename class*. Caso ocorra alguma das operações, a análise irá considerar como uma nova classe. O motivo de tal operação se deve ao fato de que o *BOA* retorna que a classe anterior foi excluída e a nova criada. Contudo, caso ocorra uma operação de *refactoring* como *move folder* externo ao pacote, o *BOA* irá retornar o mesmo nome da classe, logo não existirão efeitos colaterais.

A detecção de *Code Smells* é outra ameaça à validade. Este não é um processo completamente preciso. Para mitigar esta ameaça, como outros trabalhos, confiamos nos resultados das implementações DECOR, que é considerada uma ferramenta estado da arte.

Por fim, não há informações precisas sobre a confiabilidade da função *isFixingRevision()* presente na infra-estrutura *BOA*. Embora seja mencionado na API que *uma mensagem de commit é considerada como uma correção de bug se ele corresponde a um conjunto de expressões regulares*, não há nenhuma informação sobre o que são essas expressões regulares. No entanto, o conjunto de dados *BOA* está sendo continuamente investigado por

[13] http://sourceforge.net/projects/nsuml
[14] http://sourceforge.net/projects/gzigzag

outras pesquisas.

Capítulo 5

Trabalhos Relacionados

Neste capítulo serão apresentados os principais trabalhos relacionados com este. Com isso, tais trabalhos foram divididos em três grupos: análise do uso de herança, uso de métricas e detecção de Code Smells e por fim, modelos de regressão e análise temática em pesquisas relacionadas com a área de Engenharia de Software.

5.1 Análise sobre Herança e interfaces

Diversas pesquisas tem sido empregados para avaliar o uso de herança. Muitas dessas são anteriores à própria criação do Java. Embora diversos dos seus conceitos possam ser empregados até os dias atuais, não emprega estudos de caso em Java, especialmente sobre o uso de interfaces. Com isso, serão destacados apenas os trabalhos que possuem maior relação com o propósito deste trabalho.

Um estudo (TEMPERO; NOBLE; MELTON, 2008) propôs avaliar o quanto os recursos de herança e interface são utilizados em sistemas Java. Para isso, considerou a análise de herança e interface de forma separada, assim como a herança ocorrida com bibliotecas externas e superclasses definidas internamente no sistema. Para isso, foi criado um conjunto de métricas, analisando 93 sistemas totalizando 100.000 tipos definidos pelo usuário. Na análise longitudinal realizada, observou-se que os sistemas adquirem muitos novos tipos de herança ao longo do tempo. Além disso, foi constatado que 3 em cada 4 tipos utilizam alguma forma de herança ou implementação de interfaces em pelo menos metade das aplicações avaliadas. Tal utilização se refere tanto a superclasses e interfaces (internas ou externas). Embora esta pesquisa possua um propósito quantitativo em avaliar o uso de herança, não faz medições sobre efeitos negativos do seu uso, ao contrário deste trabalho, que efetua medições sobre efeitos negativos que o uso da herança podem proporcionar ao programador, alertando-o sobre como programas Java são construídos no mundo real. É por este motivo que, para analisar qualidade, as superclasses externas são descartadas, pois não foram definidas internamente no sistema. Contudo, o trabalho citado analisa indicadores desconsiderados neste trabalho, como por exemplo a métrica

estrutural *DIT*.

Outro estudo que propõe avaliar o uso da herança é (TEMPERO; YANG; NOBLE, 2013). Neste caso, a proposta consiste em avaliar por que um programador escolhe utilizar herança, e até que ponto tal escolha é realmente necessária. Com isso, seriam identificados situações onde a herança foi empregada de forma desnecessária. Além da avaliação sobre as definições das classes, os métodos também foram avaliados. Foi constatado que 34% das subclasses utilizam recursos das superclasses, contudo as subclasses são constantemente usadas polimorficamente pela sua superclasse (dois terços dos casos). Também foi constatado que não existem muitas oportunidades de efetuar operações de *refactoring* para composição. A principal diferença deste estudo para o trabalho proposto é que este estudo avalia se os recursos propostos pela herança foram empregados, enquanto que o trabalho proposto avalia os efeitos colaterais que o emprego da herança pode gerar no código-fonte.

No que tange à avaliação sobre os efeitos negativos do seu uso, diversos trabalhos tem se baseado na medição de métricas estruturais específicas de herança, como *Depth of Inheritance (DIT)* ou *Number of Children (NOC)*. (DALY et al., 1996) por exemplo construiu um conjunto de atividades para serem executadas em classes com herança em diferentes níveis de *DIT*, e replicadas em classes sem herança. Concluiu que a atividade com herança obteve uma quantidade maior de linhas de código alteradas, e quanto maior for o valor de *DIT*, menor será a manutenibilidade do sistema. Tal estudo foi replicado por (CARTWRIGHT, 1998) que provou justamente o contrário, isto é, classes de herança são mais fáceis de manter. (HARRISON; COUNSELL; NITHI, 2000) confirmou que herança afeta a manutenibilidade do sistema. Contudo, destacou que sistemas grandes geralmente são difíceis de entender com ou sem o uso de herança. Tal informação motivou o particionamento dos sistemas nas perguntas de pesquisa.

5.2 Métricas e Detecção de *Code Smells*

Diversas abordagens têm sido propostas para detectar e recomendar a remoção de *Code Smells*. As abordagens se baseiam nos Code Smells catalogados de livros conhecidos na literatura (FOWLER et al., 1999)(SURYANARAYANA; SAMARTHYAM; SHARMA, 2015). Algumas destas abordagens são baseadas em métricas, e outras efetuam análise sobre o histórico de versões do código-fonte.

Dentro das abordagens baseadas em métricas, (MOHA et al., 2010) construíram uma ferramenta chamada *DECOR*, utilizando uma linguagem específica de domínio (*DSL*) para especificar *Code Smells*. Com isso, estes podem ser definidos em um vocabulário como entrada do sistema. Desta forma, diversos *Code Smells* podem ser identificados, desde que as entradas sejam parametrizadas corretamente. Por exemplo, um *Code Smell* do tipo *long method* pode ser parametrizado para a quantidade de linhas que o usuário

acredita ser conveniente, para classificar a classe com o *Code Smell* mencionado. A ferramenta *Code Smell Analyser* empregada neste trabalho foi construída por (TUFANO et al., 2015), utilizando desta abordagem de extração de métricas estruturais, baseando-se nas condições propostas originalmente pelo *DECOR*.

Além da abordagem baseada em métricas, outros trabalhos tem buscado detectar *Code Smells* utilizando técnicas de mineração de dados e inteligência artificial. Já (JIANG et al., 2014) fizeram uma proposta de remoção do *Code Smell Divergent Change* utilizando o algoritmo *K-means*, com valor dinâmico para a variável *K*. Por fim, (BOUSSAA et al., 2013) construíram uma abordagem baseada em algoritmos genéticos, com uma população de 100.000 soluções candidatas para detecção de três *Code Smells*. O maior problema desta abordagem é o tempo gasto para detectar, que é superior a 1 hora, o que torna inviável para uso em um *plug-in*.

Ao invés de se basearem unicamente na análise estática do código-fonte usando métricas, (RATIU et al., 2004) propôs uma abordagem para detectar *Code Smells* baseado em todo o histórico de alterações da classe afetada. Contudo a pesquisa se limitou a apenas dois *Code Smells*: *God Class* e *Data Class*. Foram construídas métricas para avaliar o incremento das classes ao longo das versões, como quantidade de linhas, métodos, atributos públicos, etc. Já (PALOMBA; BAVOTA; PENTA, 2013) observaram que muitos *Code Smells* obtém melhor acurácia de detecção caso sejam avaliadas as mudanças no código-fonte ao longo dos *commits* realizados. Com isso, construíram uma ferramenta chamada *HIST* que detecta os *Code Smells Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob Class* e *Feature Envy* através do histórico de *commits* de cada classe.

Sobre operações de *refactoring* visando remover *Code Smells*, (TSANTALIS; CHATZIGEORGIOU, 2009) construíram uma ferramenta chamada *JDeodorant*, visando remover o *Code Smell Feature Envy*, utilizando a operação de *refactoring Move Method*. (BAVOTA et al., 2014) propuseram a utilização da operação do refactoring *Extract Class* para remover o *Code Smell Blob Class*, utilizando teoria dos jogos, onde cada jogador representa uma subclasse a ser extraída da *Blob Class*. Também foi utilizada a técnica *LDA* para identificar a quantidade de jogadores. Contudo, esta abordagem ainda é manual e carece de uma ferramenta automatizada.

5.3 Uso de Modelos de Regressão e Análise Temática

Diversos trabalhos tem utilizado modelos de regressão para avaliar melhor as relações entre determinadas variáveis. (TUFANO et al., 2015) por exemplo utilizou um modelo de regressão linear para identificar se um determinado *commit* é do tipo *smell-introducing*, ou seja, se os valores coletados das métricas estruturais para uma classe em determinado *commit* apontam para que esta seja infectada por determinado Code Smell.

Além da regressão linear, também existem trabalhos que já utilizaram o NBR.(SILVA;

VALENTE; MAIA, 2015) identificou diferentes situações onde ocorrem *co-change commits*, e os agrupou em clusters. Para detectar quais destes *clusters* exercem influência sobre a quantidade de *co-change commits*, utilizou-se o modelo *NBR*. Além dos diferentes tipos de *clusters*, foram utilizados outras variáveis preditoras com a quantidade de arquivos que contém código-fonte e a quantidade de meses em que o sistema teve commits.

Sobre a análise temática, (SILVA; TSANTALIS; VALENTE, 2016) catalogou um conjunto de motivos pelos quais os desenvolvedores efetuam operações de *refactoring*. Tais motivos foram agrupados pela operação de *refactoring*. Com isso, durante 2 meses, foram identificadas diversas operações de *refactoring* em 124 sistemas. Para cada operação realizada, um email era disparado para o programador que realizou tal *commit*, questionando as razões pelas quais tais alterações foram realizadas nas classes. As respostas foram analisadas utilizando análise temática, e os padrões foram identificados e catalogados como os motivos para tais operações de *refactoring*.

Capítulo 6

Conclusão

Neste trabalho, foram formuladas e analisadas as respostas das seis perguntas de pesquisa envolvendo o uso de herança e interface nos sistemas escritos em Java. As três primeiras perguntas analisaram os sistemas levando em consideração a sua época de criação, as duas últimas avaliaram a evolução dos sistemas no decorrer do tempo, e por fim, a quarta pergunta obteve ambas as análises. Cada pergunta ofereceu uma conclusão. A seguir, as conclusões de cada pergunta de pesquisa irão fornecer uma análise final sobre o resultado deste trabalho.

6.1 Lições Aprendidas

As lições aprendidas mostram uma compilação das respostas de todas as perguntas de pesquisa, fornecendo um conjunto útil de observações e recomendações.

Lição #1 - *Sistemas mais recentes têm projetado melhor o recurso de herança, mas o crescimento destas classes ocorre mais rapidamente* - Algumas melhorias foram encontradas no uso de herança: sistemas mais recentes têm apresentado menos mudanças corretivas, a diminuição do uso de herança pode indicar que sua utilização esteja ocorrendo em situações mais apropriadas. Além disso, o operador *instanceof* têm sido menos usado para obter funcionalidades específicas de subclasses, e principalmente, ao comparar com classes sem herança, este recurso vem sendo projetado sem a tendência de obter valores consideravelmente altos de métricas de coesão e acoplamento, não se convertendo em *Code Smells*. Contudo, a última versão dos sistemas mostrou que classes com herança acumulam funcionalidades com mais facilidade que as classes sem esta hierarquia, acumulando funcionalidades, e consequentemente gerando *Code Smells* como *God Class* e *Complex Class*. O excesso de dependência entre as subclasses e superclasses têm mostrado na prática que estas hierarquias possuem uma tendência de crescimento acima da média em funcionalidades, possibilitando a geração de classes menos coesas(violação de *SRP*), e com vários motivos para ser modificada (violação de *OCP*).

Lição #2 - *Desenvolvedores ainda tendem a projetar herança visando primariamente*

o reaproveitamento de código - Embora este trabalho tenha mostrado melhorias no uso de herança em sistemas mais recentes,a motivação primária para o seu uso têm permanecido o mesmo desde quando este recurso foi concebido, que é o reaproveitamento de código. A análise temática realizada sobre as adições e remoções de herança mostraram que este recurso é o mais utilizado pelos desenvolvedores para evitar duplicidade no código-fonte, em detrimento ao uso de composição. Foi detectado que na última versão dos sistemas existe uma tendência de acréscimo no uso do *Code Smell Lazy Class*, assim como *Complex Class*. Estes dois *Code Smells* tendem a ser antagônicos, contudo ocorrem ao mesmo tempo em classes com herança. Foi observado que *Lazy Classes* podem ser adicionadas como superclasses apenas para evitar duplicidade de código-fonte, assim como *Complex Classes* podem ser adicionadas com diversas funcionalidades para subclasses que podem ser *Lazy Classes*, com poucos métodos complementando o comportamento da superclasse. Com isso, a motivação por parte dos desenvolvedores de obter primariamente reaproveitamento de código pode conduzir hierarquias de herança a se tornarem extremamente complexas ao longo do tempo, aumentando a complexidade de uma eventual operação de *refactoring*. Como foi mostrado, apenas 1,45% das classes que estavam em hierarquia de herança sofreram operações de *refactoring*, sendo que na análise temática não foi detectado nenhum *God Class* sendo extraído. Isso indica que estudos temporais precisam ser realizados para um melhor entendimento sobre o crescimento destas classes, para que sejam produzidas novas recomendações evitando tal situação.

Lição #3 - *Existem alguns indícios de que interfaces ainda são subutilizadas*

Neste trabalho, foram observados alguns aspectos que indicam pouco uso das interfaces. O primeiro que merece destaque consiste nos resultados da análise temática, onde foi mostrado que as interfaces foram construídas com um propósito de reduzir o acoplamento entre classes, encapsulando as funcionalidades definidas em seu contrato para outras classes. Estas reduções de acoplamento podem não ser vistas como algo necessário pelos desenvolvedores ao projetar sistemas com arquiteturas mais simples. Isso é reforçado porque em sistemas de pequeno e médio porte houve um decréscimo no uso de interfaces. Contudo em sistemas maiores, que naturalmente tendem a demandar arquiteturas mais complexas, o seu uso cresceu. Outro ponto que merece atenção é o fato que interface é um recurso bem menos utilizado do que herança, com cerca 20% de uso sobre as classes dos sistemas, contra 40% de herança. É necessário observar que em herança, diversas classes estendem de superclasses externas, impossibilitando de estender também de superclasses internas. Com interface, uma classe pode implementar quantas interfaces assim desejar, logo não existe tal restrição. E por fim, foi realizada uma consulta sobre a quantidade de métodos nas interfaces, e foi detectado que 15,33% das interfaces construídas não possuem métodos. Uma interface sem métodos não define comportamento, logo seu uso está incoerente com o propósito.

Lição #4 - *Interfaces possuem tendência de possuir pouca relação com as classes que as*

implementam - Neste trabalho, foi observado que, ao contrário da herança, as classes que implementam as interfaces possuem outras funcionalidades além dos métodos implementados que foram definidos pelas interfaces. Isto indica que as classes que as implementam não existem em função das interfaces. As evidências sobre esta afirmação consistem três observações. A primeira consiste nas métricas estruturais, onde foi observado que os indicadores das interfaces sobre as métricas são parecidos com as classes que não implementam interfaces e não estendem superclasses. Outro ponto consiste nas operações de *refactoring*, que ocorrem com mais frequência em classes que implementam interfaces se comparado com herança. E o terceiro ponto consiste na análise temática, onde foi observado que as classes que implementam interfaces possuem uma quantidade razoavelmente maior de métodos do que os definidos pelas interfaces. Confirmando esta tendência, conclui-se que não faz sentido comparar herança com interface. Em classes com herança, existe um alto nível de acoplamento e funcionalidades compartilhadas entre superclasses e subclasses. Em interfaces, esse acoplamento é mais leve e parcial. Desta forma, com base nos resultados obtidos nas métricas e *Code Smells* para interface, conclui-se que não existe um comportamento diferenciado para as classes que implementam interfaces. Estas classes podem ser qualquer classe do sistema que, em determinado momento, implementa uma interface com um único método para ser usada polimorficamente em algum ponto do sistema. A análise relevante sobre interface consiste em verificar eventuais quebras de encapsulamento pela violação do princípio *SOLID ISP*. Contudo esta violação está mais voltada para a própria definição da interface do que para as classes que as implementam.

6.2 Trabalhos Futuros

Existem diversas oportunidades para evoluir este trabalho. Uma primeira oportunidade consiste em analisar as causas pelas quais as quebras de encapsulamento têm ocorrido nas classes. E destas identificações detectadas, sugerir recomendações de operações de *refactoring* baseado no que tem realmente sido empregado nos sistemas. Para isso, é necessário efetuar um mapeamento completo de todas as situações nas quais as quebras de encapsulamento ocorrem. Este trabalho analisa o uso do *instanceof*, que contribui para a quebra. Mas existem diversas outras situações para se avaliar, como por exemplo, mapear as quebras de encapsulamento que ocorrem ao detectar o *Code Smell Class Data Should Be Private* (que como foi mostrado neste trabalho, não predomina sobre classes com herança). Novos *Code Smells* podem ser detectados, como o *Refused Parent Bequest* que ainda não é detectado pela ferramenta *Code Smell Analyser*. além de uma análise de situações onde os princípios *SOLID* são violados. Neste caso, será possível avaliar o quanto o uso de herança ou interfaces realmente contribuem para que tal quebra ocorra.

Outra oportunidade de evolução que utiliza de metodologia semelhante à este trabalho, consiste em avaliar quantitativamente a ocorrência dos *Code Smells* em função do tempo.

Por exemplo, o *Code Smell Spaghetti Code* viola princípios primitivos de desenvolvimento de sistemas, que incluem programação estruturada e separada em módulos. Desta forma, é possível que atualmente estas violações não ocorram mais com tanta frequência como ocorriam antes. Estes resultados irão possibilitar que o foco das futuras pesquisas seja em torno dos *Code Smells* que realmente têm ocorrido recentemente, propondo novas técnicas e recomendações mais próximas dos problemas vivenciados na indústria.

Outra oportunidade consiste em entender por que classes com herança e implementação de interfaces vêm sendo menos empregado em sistemas de pequeno e médio porte. Algumas hipóteses podem estar relacionadas ao maior uso de *frameworks* de terceiros em sistemas menores, ocasionando em uma não contabilização de classe com herança, já que este trabalho avalia apenas herança interna. Outra possibilidade pode estar relacionada com a quantidade de desenvolvedores do sistema, pois pode existir uma relação entre a quantidade de desenvolvedores do sistema com a quantidade de abstrações criadas com herança e interface. E por fim, outra possibilidade pode ser porque em sistemas menores a tendência é que os requisitos sejam menos complexos, consequentemente sem demandar hierarquias mais complexas.

Por fim, uma última oportunidade a ser citada consiste em efetuar uma análise temporal das classes com herança, avaliando o seu crescimento em funcionalidades, débitos técnicos adquiridos, e separando o comportamento de superclasses e subclasses em um modelo de herança. Neste trabalho, superclasses e subclasses foram tratadas da mesma maneira. Contudo, algumas métricas e *Code Smells* podem incidir de forma diferente em superclasses e subclasses. Por exemplo, uma hipótese é que o *Code Smell God Class* surja mais frequentemente em superclasses, em virtude da superclasse ser provedora de funcionalidades para as suas subclasses.

Referências

ABBES, M. et al. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: **Proc. of the 15^{th} Int. Conf. on Software Maintenance, Reengineering and Reverse Engineering (CSMR'2011)**. Washington, DC, USA: IEEE Computer Society, 2011. p. 181–190.

BAVOTA, G. et al. In medio stat virtus: Extract class refactoring through nash equilibria. **Proc. of the 18^{nd} Int. Conf. on Software Maintenance, Reengineering and Reverse Engineering (CSMR'2014)**, IEEE Computer Society, Los Alamitos, CA, USA, v. 00, n. undefined, p. 214–223, 2014.

BLOCH, J. **Effective Java (The Java Series)**. 2. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

BOUSSAA, M. et al. Competitive coevolutionary code-smells detection. In: **Proc. of the 5^{th} Int. Symp. on Search Based Software Engineering (SSBSE'2013)**. New York, NY, USA: Springer-Verlag New York, Inc., 2013. p. 50–65.

BUDD, T. **An Introduction to Object-oriented Programming**. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1991.

CARTWRIGHT, M. An empirical view of inheritance. **Information and Software Technology**, Butterworth-Heinemann, Newton, MA, USA, v. 40, n. 14, p. 795–799, dez. 1998.

CHEN, L. Continuous delivery: Overcoming adoption obstacles. In: **Proc. of the Int. Work. on Continuous Software Evolution and Delivery (CSED'2016)**. New York, NY, USA: ACM, 2016. p. 84–84.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. **IEEE Trans. on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 20, n. 6, p. 476–493, jun. 1994.

COHEN, J.; COHEN, P. **Applied multiple regression/correlation analysis for the behavioral sciences**. Hillsdale, NJ: Erlbaum, 1975.

CRUZES, D. S.; DYBA, T. Recommended steps for thematic synthesis in software engineering. In: **Proc. of the 5^{th} Int. Symp. on Empirical Software Engineering and Measurement (ESEM'2011)**. Washington, DC, USA: IEEE Computer Society, 2011. p. 275–284.

DALY, J. et al. Evaluating inheritance depth on the maintainability of object-oriented software. **Empirical Software Engineering**, v. 1, n. 2, p. 109–132, 1996.

DYER, R. et al. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: **Proc. of the 35^{th} Int. Conf. on Software Engineering (ICSE'2013)**. Piscataway, NJ, USA: IEEE Press, 2013. p. 422–431.

FOWLER, M. et al. **Refactoring: Improving the Design of Existing Code**. Boston, MA, USA: Addison-Wesley, 1999.

GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-oriented Software**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

HAMMER, C.; SCHAADE, R.; SNELTING, G. Static path conditions for Java. In: **Proc. of the 3^{rd} Work. on Programming Languages and Analysis for Security (PLAS'2008)**. New York, NY, USA: ACM, 2008. p. 57–66.

HARRISON, R.; COUNSELL, S.; NITHI, R. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. **Journal of Systems and Software**, Elsevier Science Inc., New York, NY, USA, v. 52, n. 2-3, p. 173–179, jun. 2000.

HOLUB, A. Why extends is evil: Improve your code by replacing concrete base classes with interfaces. JavaWorld.com, 2003.

JIANG, D. et al. Distance metric based divergent change bad smell detection and refactoring scheme analysis. In: **Proc. of the 26^{th} Int. Journal of Innovative Computing, Information and Control (ICIC'2014)**. [S.l.]: ICIC International, 2014. v. 10, n. 4.

KHOMH, F. et al. An exploratory study of the impact of antipatterns on class change- and fault-proneness. **Empirical Software Engineering**, v. 17, n. 3, p. 243–275, 2012.

KOC A., T. A. A survey of version control systems. **Proc. of the 2^{nd} Int. Conf. on Engineering and Meta-Engineering (ICEME'2011)**, 2011.

LAVALLÉE, M.; ROBILLARD, P. N. Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality. In: **Proc. of the 37^{th} Int. Conf. on Software Engineering (ICSE'2015)**. Piscataway, NJ, USA: IEEE Press, 2015. p. 677–687.

LISKOV, B. Keynote address - data abstraction and hierarchy. In: **Proc. of the 2^{nd} Conf. on Object-oriented Programming Systems, Languages and Applications (OOPSLA'1987)**. New York, NY, USA: ACM, 1987. p. 17–34.

MARTIN, R. C. **Agile Software Development: Principles, Patterns, and Practices**. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

MENS, T.; TOURWÉ, T. A survey of software refactoring. **IEEE Trans. on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 30, n. 2, p. 126–139, fev. 2004.

MEYER, B. Reusability: The case for object-oriented design. In: BIGGERSTAFF, T. J.; PERLIS, A. J. (Ed.). **Software Reusability**. New York: acm press, 1989. p. 1–33.

MOHA, N. et al. Decor: A method for the specification and detection of code and design smells. **IEEE Trans. on Software Engineering**, v. 36, n. 1, p. 20–36, 2010.

PALOMBA, F.; BAVOTA, G.; PENTA, M. D. Detecting bad smells in source code using change history information. In: **Proc. of the 28^{th} Int. Conf. on Automated Software Engineering (ASE'2013)**. Palo Alto, CA, USA: IEEE Computer Society, 2013. p. 268–278.

PALOMBA, F. et al. Do they really smell bad? A study on developers' perception of bad code smells. In: **Proc. of the 30^{th} Int. Conf. on Software Maintenance (ICSM'2014)**. Washington, DC, USA: IEEE Computer Society, 2014. p. 101–110.

RATIU, D. et al. Using history information to improve design flaws detection. In: **Proc. of the 8^{th} Int. Conf. on Software Maintenance, Reengineering and Reverse Engineering (CSMR'2004)**. Washington, DC, USA: IEEE Computer Society, 2004. p. 223–.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. **Unified Modeling Language Reference Manual**. 2. ed. Essex, UK, UK: Pearson Higher Education, 2004.

SILVA, D.; TSANTALIS, N.; VALENTE, M. T. Why we refactor? confessions of github contributors. In: **Proc. of the 24^{th} Int. Symp. on Foundations of Software Engineering (FSE'2016)**. New York, NY, USA: ACM, 2016. p. 858–870.

SILVA, L. L.; VALENTE, M. T.; MAIA, M. d. A. **Co-Change Clustering**. Tese (Doutorado) — Universidade Federal de Minas Gerais (UFMG), 2015.

SNYDER, A. Encapsulation and inheritance in object-oriented programming languages. In: **Proc. of the 1^{st} Conf. on Object-oriented Programming Systems, Languages and Applications (OOPSLA'1986)**. New York, NY, USA: ACM, 1986. p. 38–45.

SURYANARAYANA, G.; SAMARTHYAM, G.; SHARMA, T. **Refactoring for Software Design Smells. Managing Technical Debt**. San Francisco, CA, USA: Elsevier, 2015.

TAIVALSAARI, A. On the notion of inheritance. **ACM Computing Surveys (CSUR)**, ACM, New York, NY, USA, v. 28, n. 3, p. 438–479, set. 1996.

TEMPERO, E.; NOBLE, J.; MELTON, H. How do Java programs use inheritance? an empirical study of inheritance in Java software. In: **Proc. of the 22^{nd} European Conf. on Object-Oriented Programming (ECOOP'2008)**. Berlin, Heidelberg: Springer-Verlag, 2008. p. 667–691.

TEMPERO, E.; YANG, H. Y.; NOBLE, J. What programmers do with inheritance in Java. In: **Proc. of the 27^{th} European Conf. on Object-Oriented Programming (ECOOP'2013)**. Berlin, Heidelberg: Springer-Verlag, 2013. p. 577–601.

TSANTALIS, N.; CHATZIGEORGIOU, A. Identification of move method refactoring opportunities. **IEEE Trans. on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 35, n. 3, p. 347–367, maio 2009.

TUFANO, M. et al. When and why your code starts to smell bad. In: **Proc. of the** 37^{th} **Int. Conf. on Software Engineering (ICSE'2015)**. Piscataway, NJ, USA: IEEE Press, 2015. p. 403–414.

YAMASHITA, A. F.; MOONEN, L. Do code smells reflect important maintainability aspects? In: **Proc. of the** 28^{th} **Int. Conf. on Software Maintenance (ICSM'2012)**. Washington, DC, USA: IEEE Computer Society, 2012. p. 306–315.