A Study About the Life Cycle of Code Anomalies

Wallace Ribeiro, Vanessa Braganholo, Leonardo Murta

Universidade Federal Fluminense
Niterói, RJ, Brazil
Email: wribeiro@id.uff.br, {vanessa, leomurta}@ic.uff.br

***Resumo*—Software projects usually follow key architectural principles, such as modularity, high cohesion, low coupling, etc. However, during software evolution, sometimes developers introduce changes that are not compatible with these principles, called code anomalies. Some approaches aim at detecting and fixing code anomalies, but they miss a deeper discussion about the introduction patterns and life cycle of such code anomalies. In this paper we analyze several projects to understand the incidence of code anomalies and their life cycle. In addition, we correlate code anomalies with some project characteristics and with other code anomalies. For example, we could observe that *Shotgun Surgery* and *God Method* anomalies never happen together. Moreover, we observed that *Shotgun Surgery*, *God Package*, and *Misplaced Class* anomaly life cycles are closely correlated.**

I. Introdução

Arquitetura de software pode ser definida como a organização dos componentes integrantes de um software e a comunicação destes entre si [1]. Esta definição é a mais aceita entre vários autores [2], [3], [4], com algumas variações. A arquitetura de software possui grande importância por ser uma representação relativamente concisa e intelectualmente compreensível de como o sistema é estruturado e como seus componentes trabalham em conjunto [5]. Além disso, a arquitetura de software segue princípios chave como modularidade, alta coesão, baixo acoplamento, etc., que visam atender a atributos de qualidade como manutenibilidade, reusabilidade, inteligibilidade, entre outros.

Apesar da importância da arquitetura para a construção de software, cabe ressaltar que nem sempre sua evolução ocorre da maneira esperada [6]. Às vezes os desenvolvedores introduzem mudanças que violam os princípios arquiteturais mencionados anteriormente, levando ao surgimento de anomalias de código. Essas anomalias podem gerar dificuldades na manutenção das funcionalidades existentes e na implementação de novas funcionalidades, aumentando assim o custo de desenvolvimento [6].

Vários autores definem catálogos que listam possíveis anomalias e indicam como essas anomalias poderiam ser detectadas e removidas do código-fonte. Por exemplo, Fowler et al. [3] listam 21 anomalias e indicam possíveis formas de remover essas anomalias por meio de refatorações. Além disso, outros trabalhos [7], [8] complementam o catálogo de Fowler et al. [3] com equações matemáticas que permitem detectar objetivamente cada anomalia no código-fonte.

Apesar da existência de diversos trabalhos visando catalogar, detectar e remover anomalias, há pouco conhecimento sobre as circunstâncias em que anomalias surgem em projetos de software. Aspectos como o ciclo de vida de anomalias, padrões relacionados a esses ciclos de vida e a frequência de surgimento de anomalias ainda estão pouco claros na literatura. Desta forma, o objetivo deste trabalho consiste em fornecer um entendimento mais profundo sobre esses aspectos por meio da resposta às seguintes questões de pesquisa:

Q1: Qual é a incidência de anomalias em projetos de software? Anomalias podem ser raras em alguns projetos e frequentes em outros. Assim, essa questão visa observar se o número de anomalias varia entre projetos, se existem anomalias que são mais frequentes do que outras ou se a frequência das anomalias depende de características do projeto.

Q2: Como se caracteriza o ciclo de vida das anomalias? Anomalias podem surgir no momento da criação dos construtos estruturais do projeto (i.e., pacotes, classes ou métodos) ou serem adquiridas com o passar do tempo. Além disso, algumas dessas anomalias podem ser corrigidas em algum momento subsequente. Caso sejam corrigidas, elas podem reaparecer em algum momento futuro na história do construto. Com essa questão de pesquisa pretendemos saber se diferentes anomalias seguem ciclos de vida diferentes, e se essas variações são uniformes em diferentes projetos.

Q3: Há padrões relacionados ao surgimento de anomalias? Anomalias podem estar relacionadas a características do projeto onde ocorrem, como, por exemplo, o estilo arquitetural adotado. Desta forma, essa questão visa observar se determinadas métricas de produto, padrões arquiteturais ou escolhas dos desenvolvedores correlacionam com o surgimento de determinadas anomalias.

Q4: Há correlação entre os ciclos de vida de diferentes anomalias? Algumas anomalias podem ter seus ciclos de vida correlacionados com os ciclos de vida de outras anomalias. Por exemplo, algumas anomalias podem estar sempre emergindo em conjunto. Por outro lado, algumas anomalias pode surgir concomitantemente com a correção de outras anomalias. A resposta a essa questão de pesquisa pode ajudar a entender de forma mais profunda como anomalias se relacionam.

Para responder a essas questões, foram analisados 9.508 revisões de 11 projetos diferentes. Essa análise observou o ciclo de vida de 5 anomalias (*Feature Envy*, *God Method*, *God Package*, *Misplaced Class*, *Shotgun Surgery*) sobre os construtos estruturais desses projetos (pacotes, classes e métodos). Resultados dessa análise mostram, por exemplo, que as

978-1-5090-5086-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SBCARS.2016.22

anomalias *God Method* e *Shotgun Surgery* nunca atacam concomitantemente os mesmos métodos. Além disso, evidenciam uma correlação entre as anomalias *God Package*, *Misplaced Class* e *Shotgun Surgery*.

O restante desse artigo está organizado em cinco seções além desta introdução. A Seção II apresenta os materiais e métodos usados neste estudo. A Seção III apresenta os resultados de análise de cada uma das anomalias estudadas neste trabalho. A Seção IV apresenta as ameaças à validade desse estudo. A Seção V discute os trabalhos relacionados. Finalmente, a Seção VI responde as questões de pesquisa e resume as contribuições desse trabalho.

II. Materiais e Métodos

Para responder as questões de pesquisa enumeradas na Seção I, selecionamos cinco anomalias amplamente conhecidas e citadas na literatura para serem estudadas [3]:

***God Method*:** Anomalia que indica que o método possui um número excessivo de funcionalidades, comprometendo assim a sua compreensão, reutilização e manutenção. Frequentemente, o método tende a centralizar as funcionalidades da classe na qual está inserido.

***Shotgun Surgery*:** Anomalia que indica que uma modificação no método pode desencadear modificações em várias outras partes do software. Essa anomalia influencia negativamente na manutenibilidade do código devido ao efeito cascata durante modificações.

***Feature Envy*:** Anomalia que indica que o método utiliza mais atributos de outras classes do que os atributos da classe a que pertence. Esta anomalia usualmente é consequência da adição de um método em uma classe inadequada.

***God Package*:** Anomalia que indica que o pacote se tornou muito grande, de forma análoga a *God Method*, comprometendo assim sua compreensão, reutilização e manutenção.

***Misplaced Class*:** Anomalia que indica que a classe utiliza muito mais classes de determinados pacotes do que classes internas ao seu pacote. De forma análoga a *Feature Envy*, essa anomalia usualmente é consequência da adição da classe em um pacote inadequado.

Essa anomalias foram escolhidas por serem bastante citadas na literatura, apesar do ciclo de vida de várias delas não ter sido estudado ainda. Além disso, para quantificar de forma objetiva o aparecimento e desaparecimento dessas anomalias nos construtos estruturais do projeto com o decorrer do tempo, lançamos mão das fórmulas definidas por Marinescu [7]. A Tabela I apresenta as métricas que são utilizadas pelas fórmulas para os construtos de pacote, classe e método. A Tabela II apresenta as fórmulas propriamente ditas. Na Tabela II, a função *Top(M, P%)* indica se o construto em análise tem o valor da métrica *M* entre os *P%* maiores valores dessa métrica dentro do construto em que está contido. Por exemplo, a expressão *Top(LOC, 20%)* indica se o método em análise está entre os 20% maiores em termos de números de linhas de código (LOC) dentro da classe a que ele pertence. Vale notar que as fórmulas de Marinescu têm acurácia variando de 50% (para *God Package*) a 75% (para *Misplaced Class* e *God Method*) [9].

Tabela I
Métricas de Pacote, Classe e Método

Construto	Nome da Métrica	Sigla
Pacote	*Number of Client Classes*	NOCC
Pacote	*Number of Client Packages*	NOCP
Pacote	*Package Cohesion*	PC
Pacote	*Package Size*	PS
Classe	*Class Locality*	CL
Classe	*Dependency Dispersion*	DD
Classe	*Number of External Dependencies*	NOED
Método	*Access of Import-Data*	AID
Método	*Access of Local Data*	ALD
Método	*Changing Classes*	CC
Método	*Changing Methods*	CM
Método	*Cyclomatic Complexity*	CYC
Método	*Lines of Code*	LOC
Método	*Number of Import Classes*	NIC
Método	*Number of Local Variables*	NOLV
Método	*Number of Parameters*	NOP

Tabela II
Fórmulas para detecção de anomalias [7]

Anomalia	Fórmula
Feature Envy	AID $>$ 4 e Top(AID, 10%) e ALD $<$ 3 e NIC $<$ 3
God Method	Top(LOC, 20%) e LOC $\geq$ 70 e (NOP $>$ 4 ou NOLV $>$ 4) e CYC $>$ 4
God Package	PS $>$ 20 e Top(PS, 25%) e NOCC $>$ 20 e NOCP $>$ 3
Misplaced Class	CL $<$ 0.33 e Top(NOED, 25%) e NOED $>$ 6 e DD $<$ 3
Shotgun Surgery	CC $>$ 5 e CM $>$ 7

Cada anomalia estudada foi classificada em três categorias ortogonais:

Época do nascimento: Uma anomalia pode ser classificada como congênita, quando nasce juntamente com o construto, ou como adquirida, quando surge após a criação do construto.

Estado atual da anomalia: Uma anomalia ser classificada como corrigida, quando não está mais presente no construto, ou como não corrigida, caso contrário.

Padrão de correção da anomalia: Uma anomalia corrigida no passado pode voltar a afetar o mesmo construto após um certo período de tempo. Este padrão de correção e retorno é classificado de três formas distintas: simples, duplo e recorrente. O padrão simples ocorre quando a anomalia nunca foi corrigida ou se foi corrigida não reapareceu subsequentemente no mesmo construto. O padrão duplo ocorre quando após ser corrigida, o mesmo tipo de anomalia retorna apenas uma vez com o passar do tempo. Por último, o padrão recorrente ocorre quando a anomalia é corrigida e outra anomalia do mesmo tipo retorna mais de uma vez no mesmo construto.

Nesse trabalho, analisamos 9.508 revisões de 11 projetos de Código Aberto (*Open Source*) diferentes visando entender o ciclo de vida das anomalias. Para viabilizar a coleta das métricas necessárias às análises realizadas nesse trabalho, selecionamos apenas projetos escritos na linguagem Java e que estivessem disponibilizados no sistema de controle de versões Git. Os projetos foram selecionados buscando por diversidade

Tabela III
CARACTERIZAÇÃO DOS PROJETOS EM TERMOS DE TAMANHO E DURAÇÃO

	Tamanho				Duração	
Projeto	LOC	Pacotes	Classes	Métodos	Revisões	Período
Commons Email	5K a 5K	3	17 a 18	207 a 217	754	11/2004 a 06/2015
Google APIs	2K a 17K	15 a 34	24 a 106	93 a 528	594	07/2010 a 05/2015
Guava	68K a 130K	7 a 16	204 a 391	2.577 a 5.137	2.707	06/2009 a 02/2015
ImgLib2	57K a 65K	47 a 49	300 a 328	3.488 a 3.707	1.943	11/2009 a 07/2015
JFreeChart	217K a 217K	37	484	7.740 a 7.760	18	05/2011 a 10/2011
JUnit	9K a 16K	24 a 30	103 a 149	755 a 1.146	345	12/2000 a 07/2015
Log4j	32K a 43K	15 a 21	143 a 195	1367 a 1.888	1.268	11/2000 a 06/2015
MapDB	5K a 27K	1	11 a 36	177 a 742	914	08/2012 a 08/2014
SPMF	80K a 86K	111	340 a 341	2.884 a 2.891	10	07/2014 a 08/2014
Storm	101K a 101K	58	429 a 430	4.419 a 4.428	162	09/2011 a 06/2015
Titan	21K a 34K	45 a 47	173 a 239	1.579 a 2.211	793	02/2012 a 07/2014

em termos de tamanho (de 2k a 217k LOC), duração (10 a 2.707 revisões) e tipo (bibliotecas, sistemas de banco de dados, frameworks, etc.). São eles: Commons Email[1], Google APIs[2], Guava[3], ImgLib2[4], JFreeChart[5], JUnit[6], Log4J[7], MapDB[8], SPMF[9], Storm[10] e Titan[11]. A Tabela III caracteriza esses projetos em termos de tamanho e de duração. É sabido que o tamanho de um projeto varia ao longo do tempo, o que é caracterizado aqui pelas diferentes revisões do projeto, então são apresentados os valores mínimos e máximos para cada construto. A Tabela III também apresenta o número de revisões analisadas de cada projeto e o período de desenvolvimento analisado. Foram analisadas revisões consecutivas do histórico completo dos projetos no período avaliado. É possível observar que os projetos possuem características bastante diversas segundo esses aspectos.

III. RESULTADOS E DISCUSSÃO

Nesta seção apresentamos os resultados obtidos pelas análises. Esses resultados estão agrupados por anomalia nas subseções a seguir. Em seguida, fazemos uma análise conjunta das anomalias.

A. God Method

God Method é um dos tipos de anomalia mais comuns, frequentemente encontrado nos projetos de software analisados nesse trabalho. Esta anomalia foi identificada em quase todos os projetos, estando ausente apenas nos projetos Guava, ImgLib2 e JUnit. A Figura 1 mostra a distribuição da anomalia *God Method* quanto a sua época de nascimento, estado atual, e padrão de surgimento e correção nos projetos analisados. Para cada projeto, a primeira barra (em tons de verde) mostra a quantidade de métodos que apresentam a anomalia desde a primeira revisão (anomalia congênita, mostrada em verde claro), e a quantidade de métodos que apresenta a anomalia em alguma revisão futura (anomalia adquirida, mostrada em verde escuro). A segunda barra (em tons de azul) mostra a quantidade de métodos nos quais a anomalia foi corrigida (azul claro), e a quantidade de métodos nos quais a anomalia não foi corrigida (azul escuro). Finalmente, a terceira barra (em tons de vermelho) mostra a distribuição do padrão de correção da anomalia. Especificamente, é mostrada a quantidade de métodos nos quais a anomalia tem padrão simples (laranja), duplo (vermelho) e recorrente (vermelho escuro).

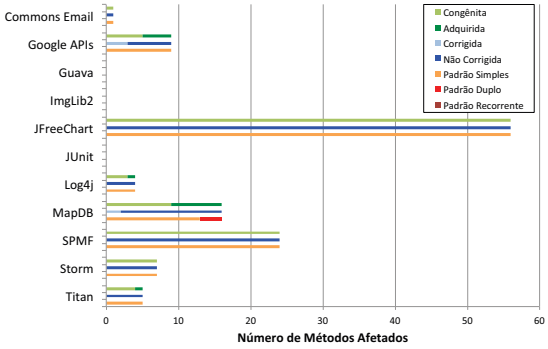

Figura 1. Distribuição da anomalia *God Method* quanto a sua época de nascimento (barra verde), estado atual (barra azul) e padrão de surgimento e correção (barra vermelha) nos projetos analisados

Como se pode notar pela Figura 1, há um número significativo de aparições congênitas, ou seja, a anomalia sempre esteve lá desde o principio. Os projetos Google APIs e MapDB são exceções, pois possuem número significativo de aparições adquiridas. Outro ponto a ser notado é quanto ao estado atual da anomalia. Pelo que pode ser observado, existe um predomínio de anomalias não corrigidas. Por último, pode-se verificar que existe um predomínio do padrão simples de surgimento e correção, sendo que somente o projeto MapDB possui alguns métodos com padrão duplo de correção.

Sendo as anomalias congênitas mais comuns que as adquiridas, pode-se supor que problemas arquiteturais são fruto de mau planejamento de um artefato desde seu nascimento. Um método nasce complexo ao invés de se tornar complexo após algumas revisões. Esses métodos também dificilmente são corrigidos, ou seja, convivem com a anomalia durante toda a duração do projeto.

[1]https://commons.apache.org/proper/commons-email/. Acesso: 07/08/2015.
[2]https://developers.google.com/api-client-library/java/. Acesso: 07/08/2015.
[3]https://github.com/google/guava. Acesso: 07/08/2015.
[4]http://fiji.sc/ImgLib2. Acesso: 07/08/2015.
[5]http://www.jfree.org/jfreechart/. Acesso: 07/08/2015.
[6]http://junit.org/. Acesso: 07/08/2015.
[7]http://logging.apache.org/log4j/2.x/. Acesso: 07/08/2015.
[8]http://www.mapdb.org/. Acesso: 07/08/2015.
[9]http://www.philippe-fournier-viger.com/spmf/. Acesso: 07/08/2015.
[10]http://storm.apache.org/. Acesso: 07/08/2015.
[11]http://thinkaurelius.github.io/titan/. Acesso: 07/08/2015.

Adicionalmente, verificou-se que os métodos afetados pela anomalia *God Method* possuem valores extremamente baixos para as métricas *Changing Classes* e *Chaning Methods*. Uma hipótese é que métodos com a anomalia *God Method* não são utilizados em muitos contextos diferentes. Uma evidência inicial que sustenta essa hipótese consiste em não termos encontrado nenhum método nos projetos analisados com as anomalias *God Method* e *Shotgun Surgery* concomitantemente. Vale notar que foram encontrados 122 métodos com a anomalia *God Method* e 132 métodos com a anomalia *Shotgun Surgery*. Uma possível explicação para esse fato é a de que esses métodos sejam criados com propósitos bem definidos desde o início. Como suas anomalias são majoritariamente congênitas, não há evidências de que mudanças durante a evolução do software causaram essas anomalias, e como raramente desaparecem, é pouco provável que sejam enxergadas como problemáticas pelos seus projetistas.

De fato, analisando cada um dos métodos que apresentaram essa anomalia, foi possível perceber que existem padrões muito similares em todos. Uma análise caso a caso mostra que 31% dos métodos afetados por *God Method* se comportam como controladores de estado, 24% como construtores de objetos, 7% como analisadores, 7% como resolvedores de protocolo, e os demais 31% têm outras funções diversas. O principal padrão observado é que esses métodos costumam realizar tarefas extensas, na maior parte das vezes de baixa complexidade, que consistem em coleções de verificações simples de vários objetos, variáveis, números, etc. Esta quantidade de verificações ocasiona o aumento da complexidade ciclomática e o aumento do número de linhas do método. O aumento nos valores dessas duas métricas é necessário para o surgimento de um *God Method*. Nos projetos estudados foram encontrados vários casos com esse padrão, desde *parsers* que checam *tokens*, métodos que constroem objetos através de coleções de verificações, métodos de conversão de informações e métodos que controlam máquinas de estados.

B. Shotgun Surgery

A anomalia *Shotgun Surgery* também é bastante comum nos projetos analisados, estando ausente apenas nos projetos Commons Email, JUnit e Titan. A Figura 2 mostra a classificação da anomalia quanto a sua época de nascimento, estado atual, e padrão de surgimento e correção. Como se pode notar pela figura, sua classificação é muito parecida com a classificação da anomalia *God Method*. Há um número significativo de aparições congênitas, ou seja, a anomalia sempre existiu desde a criação do construto. O projeto Guava é a exceção, pois possui uma quantidade significativa de aparições adquiridas.

Outro ponto a ser notado é quanto ao estado atual dos métodos com a anomalia. Pelo que pode ser observado, existe um predomínio de anomalias não corrigidas. Os projetos Google APIs e MapDB são exceções por possuírem respectivamente a totalidade e a maior parte de suas anomalias corrigidas. Por último, pode-se verificar que existe um predomínio do padrão simples de surgimento e correção de anomalias.

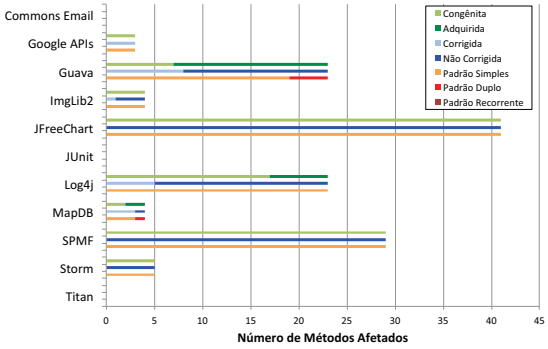

Figura 2. Distribuição da anomalia *Shotgun Surgery* quanto a sua época de nascimento (barra verde), estado atual (barra azul) e padrão de surgimento e correção (barra vermelha) nos projetos analisados

As métricas utilizadas para detecção da anomalia *Shotgun Surgery* são *Changing Classes* e *Changing Methods*. Contudo, também investigamos como outras métricas se comportam em métodos afetados por *Shotgun Surgery*. De maneira geral os métodos afetados podem ter complexidade ciclomática alta, entretanto costumam ter baixo número de parâmetros e de variáveis locais, e número pequeno de linhas (bem abaixo das 70 que caracterizaria uma das condições de *God Method*). Dos dados coletados, o maior método afetado por *Shotgun Surgery* tinha 52 linhas de código. Isso reforça a percepção anteriormente discutida de que métodos com *Shotgun Surgery* tendem a não ter a anomalia *God Method*. Algumas hipóteses podem ser levantadas a partir dessa análise. A primeira é de que métodos são criados desde o início com essas características, ou seja, desde o início foram pensados para serem usados em diferentes contextos. Outra hipótese é que esses métodos possuem poucas linhas de código para que sejam suficientemente genéricos para servir a vários contextos diferentes.

Em seguida, iniciamos uma investigação sobre a razão de alguns projetos possuírem um grande número de métodos afetados por *Shotgun Surgery* e outros um número pequeno. Percebemos que há uma correlação com a organização do projeto. Um método com *Shotgun Surgery* está sendo utilizado em muitos contextos diferentes, ou seja, seu propósito é genérico e diferentes classes e métodos são seus clientes. Em projetos Orientados a Objetos, há um foco no projeto de classes com baixo acoplamento. Por outro lado, na programação procedural existe uma clara separação entre métodos e dados a serem processados [10] – um método é utilizado para resolver problemas de outras classes. Quando se verifica uma grande quantidade de métodos sendo utilizados desta maneira, pode-se supor que de certa forma uma característica da programação procedural está sendo utilizada dentro do contexto da orientação a objetos.

A Figura 3 ilustra esse ponto. Ela apresenta dois projetos, Titan e Log4j, respectivamente com e sem a anomalia *Shotgun Surgery*. Cada círculo vermelho representa uma classe e cada seta representa uma dependência para um método de outra classe. A orientação da seta aponta para a classe cujo método

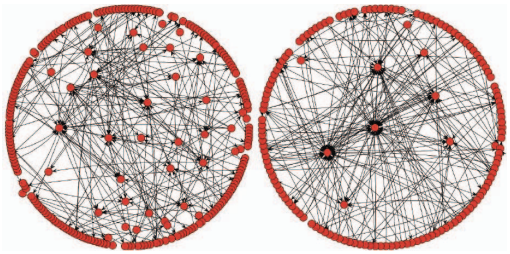

Figura 3. Distribuição do número de classes clientes de cada classe do projeto Titan (esquerda, com *Shotgun Surgery*) e Log4j (direita, sem *Shotgun Surgery*)

é invocado. Pode-se verificar no grafo da esquerda que não existem classes com muito mais clientes do que outras. Já quando um projeto se comporta de forma mais procedural, como exibido à direita, tem-se um grupo de classes que fornecem serviços através de métodos e outro que apenas utiliza esses métodos. Isso faz com que algumas classes tenham muitos clientes, enquanto outras tenham poucos. Para investigar essa hipótese, analisamos os métodos que tinham a anomalia *Shotgun Surgery* e observamos que eles residiam em classes funcionais, com nomes Helpers ou Utils. Essas classes não pertencem ao domínio da aplicação, mas servem para realizar funcionalidades genéricas no projeto.

O cálculo do desvio padrão do número de classes clientes por classe mostra o quanto o projeto distribui uniformemente as responsabilidades, como é esperado na orientação a objetos, ou não. Um valor alto de desvio padrão indica que algumas classes são utilizadas por muitas outras classes, enquanto outras por poucas. A Tabela IV indica o desvio padrão do número de classes clientes das classes (coluna σ clientes) e o número de anomalias *Shotgun Surgery* para cada projeto (coluna # *Shotgun Surgery*). O desvio padrão foi coletado no momento em que cada um dos projetos possuía o maior número de anomalias.

Essa percepção pôde ser comprovada ao calcularmos a correlação de Spearman [11] entre ambas as métricas. O valor obtido é de 0,85, o que denota uma forte correlação positiva. Desta forma, há indícios de que o paradigma procedural, comum em arquiteturas orientadas a serviços, tende a propiciar um surgimento maior de anomalias *Shotgun Surgery*.

Por fim, em dois projetos pudemos observar um alto índice de correção das anomalias: MapDB e Google APIs (Figura 2). Quando o projeto MapDB corrigiu três das quatro anomalias que possuía, o desvio padrão do número de clientes por classe caiu de 4,3 para 2,17. Já quando o projeto Google APIs teve suas 3 anomalias corrigidas, o desvio padrão do número de clientes por classe caiu de 2,99 para 1,02.

Esta característica pressupõe que para se corrigir as anomalias *Shotgun Surgery* é necessário modificar a estrutura do projeto, priorizando as colaborações entre objetos ao uso extensivo de Helpers e Utils.

C. Feature Envy

A anomalia *Feature Envy* não foi tão comumente encontrada nos projetos analisados quanto as anomalias analisadas nas

Tabela IV
DESVIO PADRÃO DO NÚMERO DE CLASSES CLIENTES POR CLASSE E NÚMERO DE ANOMALIAS *Shotgun Surgery*

Projeto	σ clientes	# *Shotgun Surgery*
Commons Email	3,32	0
Google APIs	2,99	3
Guava	8,41	23
Imglib2	5,85	4
JFreeChart	9,31	41
JUnit	2,98	0
Log4j	9,07	23
MapDb	4,3	4
SPMF	4,4	29
Storm	4,49	5
Titan	2,95	0

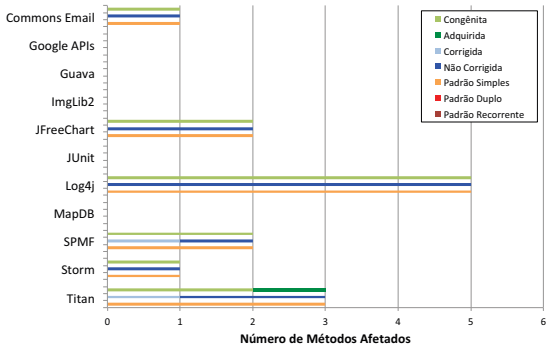

Figura 4. Distribuição da anomalia *Feature Envy* quanto a sua época de nascimento (barra verde), estado atual (barra azul) e padrão de surgimento e correção (barra vermelha) nos projetos analisados

seções anteriores. Ela foi encontrada nos projetos Commons Email, JFreeChart, Log4j, SPMF, Storm e Titan, mas não apareceu nos projetos Guava, ImgLib2, JUnit, MapDB e Google APIs. A Figura 4 mostra a classificação da anomalia quanto a sua época de nascimento, estado atual, e padrão de surgimento e correção. É possível notar que, mesmo quando presente, o número de ocorrências é muito inferior ao das anomalias anteriores.

Similarmente às anomalias anteriores, a maioria dos casos de aparições é congênita. Outro ponto a ser notado é que existe um predomínio de anomalias não corrigidas. Por último, pode-se verificar que o padrão simples de surgimento e correção ocorre na totalidade dos projetos.

Para aprofundar o entendimento sobre o ciclo de vida desta anomalia, analisamos os casos específicos em que a anomalia foi observada. Em 71% dos casos (10 dos 14 métodos que sofrem dessa anomalia) havia a construção de objetos de uma determinada classe a partir de outra classe. Isso ocorre, por exemplo, no projeto Commons Email, onde o método *attach(EmailAttachment)* da classe *MultiPartEmail* recebe um objeto da classe *EmailAttachment* e o transforma em um objeto da classe *MultiPartEmail*. Para tal, ele acessa vários atributos da classe *EmailAttachment*, o que faz com que a anomalia *Feature Envy* se manifeste. Outros métodos em outros projetos seguem o mesmo modelo de utilizar os atributos de uma classe para gerar outra. Por exemplo, a classe *VertexCentricQueryOptimizer* do projeto Titan possui a anomalia no método

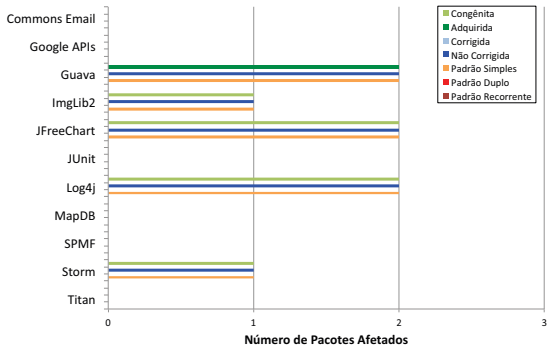

Figura 5. Distribuição da anomalia *God Package* quanto a sua época de nascimento (barra verde), estado atual (barra azul) e padrão de surgimento e correção (barra vermelha) nos projetos analisados

optimize(VertexCentricQuery). Este método utiliza um objeto *VertexCentricQuery*, otimiza algumas de suas características e cria uma lista de *VertexCentricQuery* como resultado.

D. God Package

Algumas poucas ocorrências da anomalia *God Package* foram observadas nos projetos Guava, ImgLib2, JFreeChart, Log4j e Storm, porém não observamos nenhuma ocorrência nos projetos Google APIs, Commons Email, MapDB, SPMF, Titan e JUnit. A Figura 5 mostra a classificação da anomalia quanto a sua época de nascimento, estado atual, e padrão de surgimento e correção. De acordo com a Figura, todos os projetos afetados, com exceção do Guava, possuem aparições congênitas. No projeto Guava, todas as anomalias *God Package* foram adquiridas com o passar do tempo. Além disso, jamais um pacote com a anomalia *God Package* foi corrigido. Por último, pode-se verificar que o padrão simples de surgimento e correção predomina na totalidade dos projetos, o que é esperado visto que nunca houve correção da anomalia.

Vale notar que MapDB é um projeto de apenas um pacote, portanto jamais sofrerá de *God Package*, já que uma das condições para um pacote ser afetado por *God Package* é possuir classes clientes. Como MapDB não possui classes externas ao seu único pacote, ele não foi considerado nesta análise.

Seguindo um raciocínio análogo ao usado na análise da anomalia *Shotgun Surgery*, é possível supor que uma distribuição não uniforme das classes clientes de cada pacote possa levar à anomalia *God Package*. Por outro lado, também é possível supor que uma distribuição não uniforme dos pacotes clientes de cada pacote possa levar a essa anomalia. A Tabela V exibe, para cada projeto, o desvio padrão do número de classes (coluna σ classes) e pacotes (coluna σ pacotes) clientes por pacote, e o número de pacotes com a anomalia *God Package* (coluna # *God Package*). O desvio padrão foi coletado no momento em que cada um dos projetos possuía o maior número de anomalias.

Essa percepção novamente pôde ser comprovada ao calcularmos a correlação de Spearman [11] entre as métricas. A correlação entre o desvio padrão do número de classes clientes

Tabela V
DESVIO PADRÃO DO NÚMERO DE CLASSES E PACOTES CLIENTES POR PACOTE E NÚMERO DE ANOMALIAS *God Package*

Projeto	σ classes	σ pacotes	#*God Package*
Commons Email	0,58	0,58	0
Google APIs	1,44	1,18	0
Guava	8,31	2,85	2
ImgLib2	9,56	3,33	1
JFreeChart	25,21	4,23	2
JUnit	7,2	3,41	0
Log4j	17,52	4,29	2
SPMF	6,33	4,25	0
Storm	8,65	4,39	1
Titan	4,69	2,67	0

e a contagem de anomalias *God Package* foi de 0,85, indicando uma forte correlação positiva. Por outro lado, o desvio padrão do número de pacotes clientes tem uma correlação de 0,44 com a contagem de anomalias *God Package*, indicando uma correlação positiva moderada. Temos então alguns indícios de que uma atribuição de responsabilidades pouco uniforme correlaciona com o surgimento de anomalias *God Package*, como também foi observado com a anomalia *Shotgun Surgery*.

Como ambas as anomalias *Shotgun Surgery* e *God Package* correlacionam com o número de classes clientes, iniciamos uma investigação sobre a relação entre essas duas anomalias. Uma hipótese é que programação orientada a serviços geraria anomalias *Shotgun Surgery*, e a presença dessas anomalias dentro de um pacote geraria anomalias *God Package*. Para avaliar essa hipótese, apresentamos na Tabela VI os pacotes que possuem a anomalia *God Package* ou que possuem métodos com *Shotgun Surgery*. Essa tabela mostra o nome do pacote, o número de métodos com a anomalia *Shotgun Surgery* (coluna #SS), se o pacote tem a anomalia *God Package* (coluna GP), o número de classes e pacotes clientes, o número de classes do pacote e a coesão do pacote. Além disso, a tabela também mostra a coesão média de cada projeto (marcada com μ na tabela).

Analisando essa tabela, percebe-se que todos os pacotes com a anomalia *God Package* possuem no mínimo dois métodos com a anomalia *Shotgun Surgery*. A única exceção é o pacote *net.imglib2* do projeto ImgLib2. Pode-se verificar que pacotes que possuem no mínimo dois métodos com *Shotgun Surgery* possuem alto número de classes e pacotes clientes. Esses números fazem com que duas das três métricas caracterizadoras de *God Package* sejam satisfeitas. Em resumo, para grande parte dos pacotes que possuem mais de dois métodos com *Shotgun Surgery*, a única característica que os faz não se tornarem um *God Package* é a sua quantidade de classes ser inferior a 21.

Algumas exceções são verificadas nos pacotes que possuem mais de 2 anomalias *Shotgun Surgery* e mesmo assim possuem números de pacotes e classes clientes baixos: *org.jfree.chart.block* e *org.jfree.chart.renderer.category* do projeto JFreeChart. Nestes dois casos têm-se duas anomalias *Shotgun Surgery* e um número baixo de classes (11 e 4, respectivamente) e de pacotes clientes (2 e 3, respectivamente). Contudo, pudemos observar que os métodos afetados por *Shotgun Surgery* são utilizados por classes internas, ou seja,

Tabela VI
CARACTERIZAÇÃO DOS PACOTES QUE POSSUEM MÉTODOS COM A ANOMALIA *Shotgun Surgery* (COLUNA #SS MOSTRA O NÚMERO DE MÉTODOS AFETADOS POR *Shotgun Surgery*) E DOS PACOTES COM A ANOMALIA *God Package* (COLUNA GP)

Pacote	# SS	GP	# Classes clientes	# Pacotes clientes	# Classes	Coesão
Projeto Google APIs						**0,59** (μ)
com.google.api.client.util	3	Não	35	12	13	0,220
Projeto Guava						**0,29** (μ)
com.google.common.collect	17	Sim	31	7	187	0,017
com.google.common.base	4	Sim	70	9	35	0,029
com.google.common.primitives	1	Não	18	6	16	0,058
com.google.common.io	1	Não	1	1	25	0,053
Projeto ImgLib2						**0,23** (μ)
net.imglib2	0	Sim	41	26	23	0,090
net.imglib2.util	3	Não	52	16	13	0,026
net.imglib2.view	1	Não	6	6	18	0,033
Projeto JFreeChart						**0,23** (μ)
org.jfree.chart.plot	18	Sim	80	11	46	0,068
org.jfree.chart.axis	8	Sim	81	9	45	0,068
org.jfree.chart	2	Não	60	16	20	0,053
org.jfree.chart.block	2	Não	11	2	16	0,130
org.jfree.chart.plot.dial	1	Não	0	0	12	0,260
org.jfree.chart.renderer	2	Não	56	4	12	0,060
org.jfree.chart.renderer.category	2	Não	4	3	25	0,190
org.jfree.data	4	Não	50	13	16	0,042
org.jfree.data.general	2	Não	79	13	18	0,065
Projeto Log4j						**0,31** (μ)
org.apache.log4j.helpers	8	Sim	49	12	25	0,047
org.apache.log4j	8	Sim	41	11	32	0,077
org.apache.log4j.spi	7	Não	38	9	11	0,018
Projeto SPMF						**0,60** (μ)
ca.pfv.spmf.algorithms.sequentialpatterns.BIDE_and_prefixspan	9	Não	2	2	13	0,330
ca.pfv.spmf.algorithms.sequentialpatterns.fournier2008_seqdim	1	Não	7	3	14	0,410
ca.pfv.spmf.algorithms.sequentialpatterns.spade_spam_AGP	1	Não	3	3	7	0,290
ca.pfv.spmf.algorithms.sequentialpatterns.spam	2	Não	1	1	13	0,190
ca.pfv.spmf.patterns	2	Não	4	4	2	1,000
ca.pfv.spmf.patterns.itemset_array_integers_with_count	4	Não	18	16	2	1,000
ca.pfv.spmf.patterns.itemset_array_integers_with_tids	5	Não	10	7	2	1,000
ca.pfv.spmf.patterns.itemset_list_integers_without_support	1	Não	14	2	1	1,000
ca.pfv.spmf.tools	4	Não	60	39	1	1,000
Projeto Storm						**0,28** (μ)
backtype.storm.utils	2	Sim	46	22	36	0,013
backtype.storm.task	2	Não	32	19	5	0,500
backtype.storm.tuple	1	Não	13	10	4	0,170

os clientes desses métodos são classes do próprio pacote. Isso pode ocasionar um aumento na coesão do pacote por causa de um alto acoplamento entre as classes do pacote. De fato, os pacotes *org.jfree.chart.block* e *org.jfree.chart.renderer.category* possuem valores de coesão de pacote 0,13 e 0,19 respectivamente, enquanto todos os pacotes que sofrem da anomalia *God Package* possuem coesão menor do que 0,1.

Como resultado dessa análise, pode-se assumir que algumas características influenciam o surgimento de *God Package*: (i) o pacote possui métodos com *Shotgun Surgery*; (ii) o pacote possui um número significativo de classes; e (iii) o pacote possui baixa coesão. O fato de um pacote possuir métodos com *Shotgun Surgery* frequentemente leva à anomalia *God Package*, com exceção de quando o pacote é muito pequeno ou possui coesão alta. Dessa maneira, um projeto que possua um valor alto para a coesão média de pacotes pode possuir poucos pacotes afetados por *God Package* mesmo possuindo muitos métodos com *Shotgun Surgery*.

O projeto SPMF é um dos exemplos onde existem muitas anomalias *Shotgun Surgery* sem que estas tenham ocasionado o surgimento de pacotes com *God Package*. Verificando de forma mais detalhada, pode-se notar que os métodos com *Shotgun Surgery* possuem *Changing Methods* internos ao próprio pacote. Isso significa que o projeto foi organizado de forma que exista muito uso de classes dos próprios pacotes, restringindo as trocas de informações dentro dos mesmos. Dessa maneira, uma classe pode ser utilizada por muitas classes do mesmo pacote, o que geraria *Shotgun Surgery* sem que aquele pacote possuísse classes clientes. Pode-se notar na Tabela VI que todos os pacotes do projeto SPMF que possuem *Shotgun Surgery* possuem coesão de pacote acima de 0,1. Pode-se notar também na Tabela VI que o projeto SPMF é o que possui a maior coesão média entre os pacotes.

E. Misplaced Class

A anomalia *Misplaced Class* indica que uma classe utiliza muito mais classes de outro pacote que do seu próprio pacote, estando portanto deslocada. Esse é o tipo de anomalia mais raramente encontrado nesta análise, tendo afetado apenas uma classe dos projetos Guava e Log4j.

No projeto Guava, a classe afetada, *MediaType*, do pacote *com.google.common.net*, possui anomalia adquirida, não corri-

gida e de padrão simples. Verifica-se que esta classe dependia, inicialmente, de classes de três pacotes. Após algumas revisões a classe foi simplificada, passando a usar apenas as classes de dois pacotes, assim adquirindo a anomalia *Misplaced Class*. No projeto Log4j, a classe afetada, *LoggingEvent*, pertence ao *pacote org.apache.log4j.spi* e modela a representação interna de um evento de *logging*. A anomalia é congênita, não corrigida e de padrão simples. Essa classe possui como dependência as classes de dois pacotes.

Como essa anomalia afeta poucas classes, pode-se levantar algumas hipóteses apenas por intuição. Apesar de classes com *Misplaced Class* terem suas partes pertencerem a outros pacotes, a união desses pedaços faz emergir um novo tipo de dado, com semântica bastante diferente das classes originais. A classe *MediaType*, por exemplo, utiliza os pacotes *com.google.common.collect* e *com.google.common.base*, pacotes que guardam respectivamente estruturas de coleções (listas, mapas, etc.) e estruturas de base (funções, *strings*, etc.). A classe *MediaType* usa classes desses dois pacotes para formar suas estruturas internas e, como consequência, está mais relacionada semanticamente ao pacote a que pertence (*com.google.common.net*) do que aos pacotes das classes que a constituem.

Pode-se notar também que classes afetadas pela anomalia *Misplaced Class* possuem dependências somente a pacotes afetados por *God Package*. Isso pode ser um efeito colateral de pacotes com *God Package*, já que concentram muitas classes que servem a muitos pacotes e outras classes. É possível que classes mais simples possuam dentro de um *God Package* todas as classes necessárias para a sua construção. Assim, o uso de dados externos fica limitado a poucos pacotes, causando desta forma a anomalia *Misplaced Class*. Contudo, como o número de ocorrências da anomalia *Misplaced Class* é muito baixo, não há evidências suficientes dessa relação.

Por fim, se a hipótese de que classes que possuem a anomalia *Misplaced Class* dependem de pacotes com *God Package* se confirmar, a correção da classe afetada não poderia ser a realocação da mesma no pacote que sofre de *God Package*. Adicionar uma classe a um *God Package* faz com que ele fique ainda maior e pode fazer com que esse pacote ganhe ainda mais clientes. Desta forma, a diminuição de *Shotgun Surgery*, que é o principal causador de *God package*, ajudaria a tratar a anomalia *Misplaced Class*.

F. Análise Conjunta das Anomalias

Nesta seção estudamos as características comuns a várias anomalias. A primeira delas está relacionada ao do ciclo de vida das anomalias. Nas anomalias *God Method*, *Shotgun Surgery*, *Feature Envy* e *God Package* existe um predomínio para o surgimento congênito. Esta característica pode indicar que muitos artefatos são anômalos desde sua concepção. O fato de que grande parte dessas anomalias jamais é corrigida pode significar que alguns projetistas, além de projetarem artefatos de forma anômala, não consideram que isso seja nocivo e por isso não se preocupam em corrigi-los.

Há também fortes indícios de correlação entre as anomalias estudadas. A correlação entre as anomalias *Shotgun Surgery*, *God Package* e *Misplaced Class* levanta a hipótese de que em casos extremos pode ser difícil corrigir apenas uma das anomalias sem corrigir as outras. Uma vez que existam alguns métodos com *Shotgun Surgey* dentro de um pacote, existe uma probabilidade razoável de que ele se torne *God Package*. Além disso, pacotes com *God Package* podem gerar, em classes fora deles, a anomalia *Misplaced Class*. Dessa forma, pode-se perguntar se as anomalias *Misplaced Class* são causas ou sintomas de outras anomalias: o problema é o fato de uma classe utilizar muitas classes pertencentes a poucos pacotes, ou essas muitas classes é que deveriam estar mais espalhadas em diferentes pacotes, já que estão em um pacote com *God Package*? Essa questão permanece em aberto.

IV. Ameaças à Validade

Ameaças à validade de Construção. Este estudo se baseia no estudo de anomalias de Marinescu [7], e usa suas fórmulas para definir quando ocorre ou não uma anomalia. Desta maneira, os resultados deste estudo dependem da precisão das definições e fórmulas do estudo de Marinescu. Cabe ressaltar que as definições e fórmulas de Marinescu são bem aceitas pela literatura, o que atenua essa ameaça.

Ameaças à Validade Interna. As classificações do comportamento de métodos que possuem características parecidas utilizadas para estudar a anomalia *God Method* foram feitas pelos autores desse trabalho e estão sujeitas a falhas de interpretação. A definição de construtor de objetos usada no estudo da anomalia *Feature Envy* e sua respectiva classificação também sofrem do mesmo problema. Contudo, essas classificações foram realizadas cuidadosamente, analisando-se o código fonte dos artefatos bem como seus comentários de forma a entender suas funcionalidades e classificá-los de forma apropriada.

Outra ameaça à validade interna é a heurística utilizada para verificar mudanças no nome de construtos, necessária para rastrear os mesmos construtos ao longo de revisões diferentes. A heurística utilizada consiste em analisar a semelhança entre dois construtos. Para tal, é verificado se um construto recém criado poderia ser um construto que acaba de ser removido. Essa comparação considera a estrutura interna do construto e usa um limiar de 90%, calibrado empiricamente, para aceitar os construtos como similares. Os autores verificaram os construtos apontados pela heurística como correspondentes e concluíram que ela funcionou adequadamente. Entretanto, não há garantias de que, no entendimento do desenvolvedor do projeto, dois construtos semelhantes não sejam completamente distintos no que se refere ao seus significados dentro da arquitetura do projeto.

Por fim, como o estudo foi *post-hoc*, não é possível saber se o desenvolvedor sabia se a anomalia existia ou não. Anomalias consistem em características nocivas do código, que dificultam a manutenção, reutilização, etc. De acordo com os resultados observados, essas consequências negativas não foram severas a ponto de motivar a correção por parte dos desenvolvedores. Em alguns casos, houve tal motivação, mas a correção não

foi profunda a ponto de evitar a recorrência da anomalia. Vale notar que enviamos e-mails para os desenvolvedores, porém não recebemos retorno.

Ameaças à Validade Externa. O número de projetos analisados nesse estudo pode ser considerado baixo. Foram utilizados apenas 11 projetos para estudar os padrões. Dessa forma, não se pode generalizar os resultados obtidos. Para amenizar essa ameaça, selecionamos projetos com diferentes tamanhos e durações, somando 9.508 revisões no total, fazendo com que a diversidade de nossa análise aumentasse. Outra ameaça à validade externa é em relação ao tipo de projeto analisado. Todos os projetos foram implementados em Java e são de código aberto, hospedados no repositório público GitHub. Novamente, não é possível afirmar que a análise de projetos proprietários escritos, por exemplo, em C++, apresentaria os mesmos resultados. Tentamos amenizar essa ameaça selecionando projetos de domínios diferentes, desenvolvidos por equipes diferentes.

Ameças de Conclusão. Para calcular a dispersão de clientes entre as classes de um projeto, utilizamos desvio padrão. Outras técnicas estatísticas poderiam ser utilizadas para esse propósito. Além disso, todas as análises de correlação foram feitas usando o coeficiente de correlação de postos de Spearman, observando a relação monotônica entre as variáveis. Desta forma, não é possível afirmar se as variáveis se relacionam linearmente ou não.

V. Trabalhos Relacionados

Yamashita e Moonen [12] estudam a aglomeração de anomalias e como essa aglomeração influencia na degradação do *design* de software. Esse estudo agrupa anomalias com características comuns (e.g., *God Method* e *Feature Envy*) e conclui que certas anomalias em conjunto estão relacionadas com problemas de manutenção dos sistemas de software.

Tufano *et al.* [13] analisam como surgem as anomalias de código sob o viés do fluxo de desenvolvimento de software. Algumas das descobertas feitas por esse estudo indicam que geralmente anomalias são introduzidas por desenvolvedores com alta carga de trabalho e que anomalias costumam surgir mais comumente associadas ao desenvolvimento de novas funcionalidades do que à correção de defeitos.

Nosso trabalho confirma resultados dos estudos [13], [12] para anomalias específicas. O estudo [13] aponta que anomalias surgem por decisões arquiteturais, normalmente ligadas a novas funcionalidades. Já [12] mostra que *God Method* e *Feature Envy* são relacionadas a funcionalidades de alta complexidade e elementos que cresceram desproporcionalmente. Essas conclusões, em alinhamento com o nosso trabalho, indicam que tanto *God Method* quanto *Feature Envy* são possivelmente resultantes de escolhas de soluções de *design* para tentar resolver problemas complexos.

Yamashita *et al.* [14] mostram como anomalias podem surgir em conjunto e como o surgimento de certas anomalias pode influenciar no surgimento de outras. Este estudo apresenta, entre outros resultados, a inter-relação entre *Feature Envy* e *God Class*, sendo que esta relação permanece verdadeira mesmo considerando diferentes projetos de software.

Fontana *et al.* [15] apresentam algumas relações entre anomalias e a qualidade da arquitetura de um software. Nesse estudo também foram descobertas algumas relações entre o surgimento de anomalias em diferentes partes do sistema. Um exemplo de resultado apresentado é que na maior parte dos casos os métodos afetados pela anomalia *Shotgun Surgery* são chamados por classes ou métodos que possuem algum outro tipo de anomalia. Tanto [15] quanto o nosso trabalho apontam proporções semelhantes de ocorrência de projetos (66% contra 72%, respectivamente) e de métodos (0,12% contra 0,4%, respectivamente) com *Shotgun Surgery*.

Por fim, Oizumi *et al.* [16] estudam a aglomeração de anomalias e como essa aglomeração influencia na degradação do *design* de software. Os autores definem aglomeração como sendo um conjunto de anomalias que estão relacionadas no código, como, por exemplo, duas anomalias que atuam sobre um mesmo método, ou duas anomalias que atuam sobre classes em uma mesma hierarquia de herança. Eles concluem que a aglomeração de anomalias indica problemas de *design* e que uma parte significativa desses problemas é congênita, ou seja, surge em estágios iniciais do desenvolvimento do software.

Desta forma, podemos pontuar as contribuições do nosso trabalho à luz desses estudos:

- Em relação a incidência, nós analisamos as anomalias *Feature Envy*, *God Package* e *Misplaced Class*, ainda não tratadas por nenhum trabalho da literatura;
- Em relação ao ciclo de vida, nós definimos classificações que abrangem, além do estado atual, a época do nascimento e o padrão de correção das anomalias e confirmamos que, segundo essa classificação, a maioria das anomalias é congênita e não é corrigida;
- Em relação aos padrões de surgimento, nós confirmamos alguns padrões já observados na literatura e contribuímos com outros, relacionados à anomalia *God Package*; e
- Nós identificamos correlações entre a anomalia *God Package* e as anomalias *Shotgun Surgery* e *Misplaced Class*, ainda não discutidas na literatura.

VI. Conclusão

Para analisar a degradação arquitetural através da análise do ciclo de vida de anomalias de código, esse trabalho propôs quatro questões de pesquisa. Com base nas análises realizadas em projetos reais, essas questões podem ser respondidas conforme segue.

Q1: Qual é a incidência de anomalias em projetos de software? O número de anomalias que surgem em um projeto depende de características específicas do projeto. O número total de anomalias presentes por cada cem mil linhas de código varia entre 0 e 0,80, com uma média de 0,39. O número de anomalias não parece depender do tamanho do projeto, já que a correlação de Spearman entre o número de linhas do projeto e o número de anomalias por cada mil linhas é -0,073. As anomalias mais comuns são *Shotgun Surgery* (48,53%)

e *God Method* (43,01%), enquanto as demais possuem uma incidência menor: 5,14% de *Feature Envy*, 2,57% de *God Package*, sendo a mais rara de todas a *Misplaced Class* (0,74%).

Q2: Como se caracteriza o ciclo de vida das anomalias? Em sua grande maioria, as anomalias são congênitas, não corrigidas e de padrão de correção simples, como consequência de não terem sido corrigidas. Isso corrobora a hipótese de que certas anomalias sempre surgem juntamente com a criação dos construtos e possivelmente seus projetistas não as vêm como anomalias, mas sim, como uma característica inerente ou até mesmo desejável da construção do software. A avaliação dessa hipótese, entretanto, está fora do escopo deste trabalho.

Q3: Há padrões relacionados ao surgimento de anomalias? Sim. A anomalia *God Method* está relacionada a padrões de construção de métodos que priorizam agregar várias verificações e comparações em um único método. A anomalia *Shotgun Surgery* está relacionada a padrões de construção de classes que fornecem serviços a outras classes através de métodos que podem ser utilizados em vários contextos, por várias classes diferentes. A anomalia *Feature Envy* está relacionada a métodos que constroem objetos de uma determinada classe utilizando como insumo atributos de outra classe. A anomalia *God Package* está relacionada a pacotes com baixa coesão e que possuem classes com métodos afetados por *Shotgun Surgery*. Já a anomalia *Misplaced Class*, apesar de aparecer em apenas duas classes, parece ser um efeito colateral da anomalia *God Package*. É importante ressaltar, no entanto, que não é possível afirmar que a inserção dessas anomalias seja proposital, uma vez que isso demanda um estudo que inclua entrevistas aos desenvolvedores de cada projeto, também fora do escopo deste trabalho.

Q4: Há correlação entre os ciclos de vida de diferentes anomalias? Sim. As anomalias *God Method* e *Shotgun Surgery* aparentam se relacionar de forma oposta, ou seja, a existência de uma dessas anomalias em um método parece evitar o surgimento da outra. Já as anomalias *Shotgun Surgery*, *God Package* e *Misplaced Class* aparentemente estão relacionadas a uma mesma decisão arquitetural.

Com base nos dados coletados nesse estudo, podemos levantar várias hipóteses a serem avaliadas qualitativamente em trabalhos futuros por meio de entrevistas com membros da equipe dos projetos. A primeira delas é que as manifestações das anomalias estão relacionadas às características dos projetos. De fato, ocorrências de *God Method*, *Shotgun Surgery* e *Feature Envy* evidenciam que certas decisões de projeto influenciam no surgimento das anomalias. Além disso, o fato de algumas anomalias estarem correlacionadas pode indicar que correções não devem ocorrer individualmente. De fato, na análise que fizemos para a anomalia *Misplaced Class*, notamos que é necessário um estudo aprofundado para corrigir a anomalia, de modo a evitar que esta ação acarrete em uma piora ou no surgimento de outra anomalia. Idealmente, as anomalias devem ser corrigidas em conjunto.

Desta forma, este artigo contribui com o estado a arte nos seguintes aspectos: (i) definição de uma classificação para o ciclo de vida de anomalias; (ii) análise das circunstâncias em que as anomalias se manifestam; e (iii) análise conjunta de anomalias, cruzando com características dos projetos. Esperamos que este estudo sirva como motivação para o surgimento de novos estudos, visando responder algumas das hipóteses levantadas. Além disso, esperamos que esse estudo possa motivar o desenvolvimento de ferramentas para o tratamento conjunto de anomalias, evitando que a remoção de uma anomalia leve a proliferação de outras.

Agradecimentos. Os autores agradecem ao CNPq e FAPERJ pelo financiamento parcial desse trabalho.

REFERÊNCIAS

[1] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012.

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[4] C. Hofmeister, R. Nord, and D. Soni, *Applied Software Architecture*. Addison-Wesley Professional, 1999.

[5] R. Pressman and B. Maxim, *Software Engineering: A Practitioner's Approach*, 8th ed. New York, NY: McGraw-Hill Education, 2014.

[6] M. Baldassarre, A. Bianchi, D. Caivano, and C. Visaggio, "Full Reuse Maintenance Process for Reducing Software Degradation," in *European Conference on Software Maintenance and Reengineering (CSMR)*, 2003, pp. 289–298.

[7] R. Marinescu, "Measurement and Quality in Object-Oriented Design," Ph.D. dissertation, Department of Computer Science, Polytechnic University of Timisoara, Romania, 2002.

[8] M. Munro, "Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code," in *IEEE International Software Metrics Symposium (METRICS)*, 2005, pp. 15–15.

[9] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in *IEEE International Conference on Software Maintenance (ICSM)*, 2004, pp. 350–359.

[10] D. Jana, *Java and Object-oriented Programming Paradigm*. PHI Learning, 2009.

[11] G. D. Garson, *Correlation*. Statistical Associates, 2012.

[12] A. F. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: an empirical study," in *International Conference on Software Engineering (ICSE)*, 2013, pp. 682–691.

[13] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk, "When and Why Your Code Starts to Smell Bad," in *International Conference on Software Engineering (ICSE)*, 2015, pp. 403–414.

[14] A. Yamashita, M. Zanoni, F. A. Fontana, and B. Walter, "Inter-smell relations in industrial and open source systems: A replication and comparative analysis," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 121–130.

[15] F. A. Fontana, V. Ferme, and M. Zanoni, "Towards assessing software architecture quality by exploiting code smell relations," in *International Workshop on Software Architecture and Metrics (SAM)*, 2015, pp. 1–7.

[16] W. N. Oizumi, A. F. Garcia, L. da Silva Sousa, B. B. P. Cafeo, and Y. Zhao, "Code Anomalies Flock Together: exploring code anomaly agglomerations for locating design problems," in *International Conference on Software Engineering (ICSE)*, 2016, pp. 440–451.