UNIVERSIDADE FEDERAL DE SÃO CARLOS
PROGRAMA DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Daniel Lippi Castro de Faria

Mock objects: Um estudo de caso de uma aplicação na indústria

São Carlos, São Paulo

2022

Daniel Lippi Castro de Faria

Mock objects: Um estudo de caso de uma aplicação na indústria

Trabalho de Graduação do Curso de Graduação em Engenharia de Computação da Universidade Federal de São Carlos para a obtenção do título de bacharel em Engenharia de Computação.

Orientação Prof. Dr. Auri Marcelo Rizzo Vincenzi

São Carlos, São Paulo

2022

Resumo

Contexto: Aplicações modernas em Java utilizam amplamente a ideia de *mock objects*. Nesse cenário, diversas técnicas e abordagens de uso aparecem na literatura. Muitas dessas se ancoram em princípios básicos de design OO como a interface entre objetos. No entanto, a prática dos desenvolvedores parece não coincidir com tais propostas.

Objetivos: Este estudo de caso investiga como um sistema da indústria (em Java) utiliza *mock objects* e como a aplicação destes se relaciona com conceitos de qualidade de software e princípios de design OO.

Método: Utilizando o modelo de qualidade *Factor-Criteria-Metrics*, o estudo analisa as implicações do uso de *mock objects* em aspectos da qualidade interna do produto investigado. Para isso, todas as 101 ocorrências de mocks desse sistema foram examinadas sob diferentes perspectivas como distribuição nas classes de teste, funcionalidades utilizadas, refatoração exigida e outras.

Resultados: A implementação de *mock objects* na aplicação analisada mostra forte ligação com o design da mesma. O número de dependências de uma classe em produção correlaciona-se ao número de mocks da classe de teste correspondente, por exemplo. Como consequência, decisões pobres em design, prejudicam em muito ideias internas de qualidade como a manutenibilidade dos testes da aplicação.

Conclusão: O uso de *mock objects* no sistema avaliado ignora conceitos básicos de design OO existentes na literatura. Isso é um grande indício que os desenvolvedores utilizam a técnica de forma imatura ao desconsiderar importantes aspectos de qualidade de software associados à sua aplicação.

Palavras-chave: Mock Objects, POO, OO Design, Qualidade de Software.

Abstract

Context: Modern Java applications make extensive use of mock objects. In this context, many approaches to this topic are present in the literature. A great number of these, employ basic OO design principles, such as object interfaces. However, developers practice tend to disagree with such ideas.

Aims: This study case will assess how an industry application (in Java) make use of mock objects and how this practice is related to software quality traits and OO design principles.

Method: Relying on the Factor-Criteria-Metrics quality model, this study will analyze how mock objects usage correlates to the intrinsic quality factors of the investigated application. In this regard, all the 101 mock occurrences are examined under different perspectives such as mock distribution in test classes, functionalities used, refactoring required and others.

Results: In the analyzed application, mock objects implementations are highly related to the design of the same. The number of dependencies in a production class is directly related to the number of mocks in the respective test class, as an example. As a consequence, poor design choices, are harmful to intrinsic quality factors such as the maintainability of the application tests.

Conclusion: The application of mock objects in the assessed system ignore basic OO design principles existent in the literature. This is a sign that developers are making an immature use of the technique. Important software quality aspects are underestimated by the approach analyzed.

Keywords: Mock Objects, OOP, OO Design, Software Quality.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

1.1 Mock Objects

Com o avanço de técnicas de teste de software em ambientes orientados a objetos, surge a concepção de *mock objects*. Esses objetos oferecem aos desenvolvedores a capacidade de "simular" as funcionalidades de outros objetos. Assim, o seu uso oferece facilita o desenvolvimento de testes já que as dependências de um objeto sob teste podem ser *mockadas*. Essa ideia é consolidada por Meszaros (2007) com a descrição de "dublês de teste".

Juntamente a esse fenômeno, diversas abordagens surgem para precisar a utilização desses objetos especiais. É o caso dos estudos: *Mock Roles, Not Objects* (FREEMAN et al., 2004); *Don't mock me, design considerations for mock objects* (LANGR, 2004) e *Mocks aren't stubs* (FOWLER, 2007).

Atualmente a prática de *mock objects* é encontrada na grande maioria de aplicações orientadas a objetos, porém com algumas discrepâncias no seu uso. Como é revelado em (SPADINI et al., 2019).

1.2 Objetivo Geral

Este projeto tem como objetivo realizar um estudo de caso de uma aplicação na indústria no que se refere a ao uso de *mock objects*. O propósito deste é o levantamento de métricas a partir de uma análise sobre essa prática, com a finalidade de se avaliar fatores de qualidade da aplicação. Além disso, a análise levará em consideração fatores desse sistema.

1.3 Objetivos Específicos

Mais especificamente este estudo mostrará como a prática de *mock objects* está intimamente relacionada ao design dessa aplicação. E correlacionará a isso a ausência de princípios básicos de OO no design da aplicação investigada.

Em termos de qualidade, esse estudo se restringirá ao modelo conhecido como *Factor-Criteria-Metrics* (NAIK; TRIPATHY, 2008). Focando-se em aspectos da qualidade interna de um produto de software, como modularidade e testabilidade. Essa qualidade será avaliada por perguntas de pesquisa associadas aos conceitos a serem avaliados.

1.4 Metodologia

Para realizar esse estudo de caso, uma aplicação Java com testes unitários foi escolhida. A aplicação em questão é um microsserviço dentro de um sistema gerenciador de voos particulares.

A pesquisa então foi dividida nas seguintes etapas: (1) estudo dos aportes teóricos relativos ao design e qualidade de aplicações OO; (2) preparação de métricas a serem utilizadas no estudo; (3) análise do uso de *mock objects* na aplicação; (4) aplicação das métricas selecionadas; (5) conclusões acerca da qualidade do produto.

A etapa (3) é baseada nas seguintes questões de pesquisa:

- Como é a prática de *mock objects* na aplicação?
- Qual a relação dessa prática com detalhes de design OO?

Os próximos capítulos estão organizados de acordo com as etapas descritas acima. O Capítulo 2 explora os conceitos teóricos mencionados assim como descreve as métricas a serem utilizadas na aplicação. O Capítulo 3 faz uma contextualização da aplicação, em termos de tecnologias utilizadas e a experiência da equipe de desenvolvimento. O capítulo 4 detalha a análise conduzida assim como os resultados obtidos. O capítulo 5 descreve as ameaças à validade e por fim, as conclusões tiradas são destacadas no Capítulo 6.

2 Conceitos Básicos

2.1 Introdução

Para que se entenda a análise a ser descrita nas seções seguintes, é necessário introduzir conceitos importantes que baseiam o desenvolvimento deste estudo.

2.1.1 Paradigma Orientado a Objetos

O primeiro desses conceitos é o de Programação Orientada a Objetos (POO). Esse conceito é fundamental ao desenvolvimento de software baseado em Java, já que é o paradigma adotado pela linguagem.

Embora esse paradigma seja de conhecimento geral, uma definição precisa para este é difícil de ser obtida. Considere a seguinte definição (HORSTMANN, 2019):

> *(...) object-oriented design is a programming technique that focuses on the data—objects—and on the interfaces to those objects. To make an analogy with carpentry, an "object-oriented" carpenter would be mostly concerned with the chair he is building, and secondarily with the tools used to make it; a "nonobject-oriented" carpenter would think primarily of his tools.*

A definição acima exemplifica o que é o paradigma, mas não atinge uma precisão exata dos seus conceitos. Essa dificuldade em precisar tais ideias é discutida por Martin (MARTIN, 2018) ao afirmar que a maioria das definições consiste simplesmente das ideias de *combinação de data e função* e *encapsulamento, herança e polimorfismo*.

Essa definição dada por Horstmann, no entanto, é importante pois ressalta um aspecto essencial acerca dos paradigmas OO e Estruturado, o foco do desenvolvedor. Na definição dada fica claro a distinção neste, porém na prática isso é muitas vezes dúbio.

Isso é mostrado em (BUDGEN, 2021), com a seguinte citação de Tim Rentsch (1982):

> *My guess is that object-oriented programming will be in the 1980s what structured programming was in the 1970s. Everyone will be in favour of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is.*

Serão levantados, então, os conceitos genéricos desse paradigma para depois se aprofundar nas suas ideias primordiais na tentativa de elucidar suas principais virtudes quanto à sua aplicação.

Como indicado por Horstmann, um programador utilizando o paradigma OO se concentra primeiramente nos dados da aplicação, ao invés de se preocupar na estrutura em que estão inseridos.

Quando o desenvolvedor se concentra na estruturação do código para manipular esses dados, tem-se a ideia do paradigma Estruturado. Este busca responder *o que está acontecendo* ao invés de *o que está sendo afetado* como no paradigma orientado a objetos (SCHILDT, 2021).

Com o foco da programador sobre os dados da aplicação há um favorecimento de um código com definições claras acerca dos dados tratados por esta. Pode-se ponderar, então, que POO vem para superar algumas das limitações impostas pela programação estruturada.

Considerando um exemplo de uma aplicação dentro do domínio de aviação tem-se os seguintes possíveis dados a serem tratados por esta: aeroportos, aeronaves, países, passageiros e diversos outros. Percebe-se que um aeroporto é um **objeto** (ou entidade) que possui **atributos** (ou características) que o definem como único, é o caso de seu nome por exemplo. Essas informações representam os conceitos fundamentais do paradigma.

Conhecendo, então, um objeto e seus principais atributos é possível aplicar a importantíssima ideia de **abstração** para o definir. E dessa forma um aeroporto deixa de ser visto como "uma superfície terrestre dotada de pista, prédios e equipamentos necessários ao tráfego aéreo e à manutenção de aeronaves (...)" e passa a ser simplesmente uma entidade que possui pista e aeronaves por exemplo. O foco é exclusivo aos seus atributos mais importantes, em oposição à sua definição total (SCHILDT, 2021).

Com esse conceito de abstração é possível simplificar incrivelmente a complexidade de problemas. Seguindo no exemplo acima, caso a aplicação tenha que tratar de um objeto do tipo aeroporto, o desenvolvedor não precisa se preocupar se este apresenta pista e aeronaves, pois isso já está explícito em sua categorização.

Esse exemplo é ilustrativa de como esse paradigma atinge um foco nos dados da aplicação. Além disso, há uma grande capacidade de generalização proveniente da ideia de abstração.

Essa ideia de definir esses dados no formato de objetos representa um marco importante em termos de programação, pois permite pela primeira vez que variáveis e procedimentos sejam agrupados em um mesmo escopo. Tecnicamente, essa ideia se origina do conceito de tipos de dado abstrato. Segundo Sebesta (2016), um tipo de dado abstrato é uma estrutura de dado na forma de um registro, mas que inclui subprogramas que alteram seus dados. Uma instância de um tipo de dado abstrato é chamado de objeto.

É justamente a junção dos conceitos de abstração com a de um tipo de dado abstrato (objeto) que fundamentam o paradigma. Isto é, o grande leque de funcionalidades

que POO oferece aos programadores derivam direta ou indiretamente dessa combinação de conceitos.

Destaca-se então, quais são essas funcionalidades, ou conceitos:

a) Encapsulamento: diferentemente de aplicações que utilizam o paradigma estruturado, nas quais o acesso a dados são de forma geral incontrolável e imprevisível. Isso não ocorre em aplicações utilizando POO, isso porque ambos estão no mesmo escopo chamado objeto. Isto é, dados e métodos estão **encapsulados** em objetos nessas aplicações. Como consequência o acesso a estes se torna controlado e previsível (WEISFELD, 2013);

b) Herança: Como o próprio nome sugere, esse conceito indica que as propriedades de um objeto podem ser herdadas de outros. Estabelece-se, então, uma relação hierárquica entre estes. Tem-se como exemplo um jato que possui as características gerais de uma aeronave, porém com alguma particularidades;

c) Polimorfismo: derivado do grego “mais (poli) de uma forma (morphos)” . Esse conceito nos diz que um objeto pode se comportar de diferentes maneiras, dependendo do contexto inserido. Ideia conhecida como acoplamento dinâmico (BUDGEN, 2021).

Ao refletir sobre esses conceitos, percebe-se que a ideia de encapsulamento é basicamente uma extensão da ideia de tipos de dados abstratos. Herança e Polimorfismo aparecem, então, como características complementares a essa de encapsulamento para garantir que esses objetos possam ser abstraídos dentro de uma aplicação.

Com isso, consegue-se formular, possivelmente, a maior virtude do paradigma, a que o distingue dos demais. POO garante a utilização controlada de dados e métodos por meio de objetos ao mesmo tempo que permite uma generalização na aplicação destes para simplificar e flexibilizar a solução de problemas.

Especifica-se agora parte do vocabulário utilizado dentro de POO:

a) Estado de um objeto: conjunto de dados contidos em um objeto. Ou seja, o conjunto de valores dos atributos de um objeto.

b) Comportamento de um objeto: especifica o que um objeto pode realizar, isto é, os métodos que um objeto possui.

c) Mensagens: mecanismo de comunicação entre objetos. Quando um objeto Aeroporto invoca um método de um objeto Aeronave, diz-se que aquele objeto está mandando uma mensagem a este.

Por fim, é válido ressaltar o que Alan Kay, pioneiro na programação OO, pensa a respeito do paradigma (KAY, 2022):

> *OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.*

A partir do que foi apresentado, será discutido, agora, como se pode organizar esses objetos dentro de uma aplicação de forma que se faço o máximo proveito das ferramentas oferecidas por POO. Com isso, será possível alcançar importantes traços de qualidade como o reúso de software.

2.1.2 Design Orientado a Objetos

Essa organização de objetos dentro da aplicação é chamada de design ou arquitetura. Há certa diferença entre esses termos nas diversas bibliografias. Mas assim como é exposto em (MARTIN, 2018) os termos serão utilizados de forma intercambiável, interpretando-os como o formato do sistema, isto é, sua visão em alto nível mesclado com detalhes de implementação (baixo nível)

Os propósitos de se estabelecer esse design são diversos. Mas o estudo se restringirá a um em específico. Como afirmado em (MARTIN, 2018), o objetivo principal é minimizar o esforço humano envolvido na criação e manutenção de um sistema. Quando esse esforço é baixo e ainda assim atende as demandas de um cliente, esse design é dito bom.

Uma das ideias de design mais utilizadas no contexto de aplicações orientadas a objetos é a ideia de um design voltado ao domínio. O domínio de uma aplicação consiste do setor ou ambiente da realidade em que o sistema será utilizado. Têm-se como exemplos os domínios bancário, de aviação, *e-commerce* entre vários outros.

Geralmente, as regras desse domínio são modeladas por um especialista e então repassadas a arquitetos e desenvolvedores de software que têm a árdua missão de implementar esse modelo oferecido. Como exemplo pode-se pensar que no domínio de aviação, mais especificamente no de aeroportos, deve ser possível verificar todos os horários em que um dado aeroporto operará, qual sua localização, quais os possíveis aeroportos alternativos e o horário de funcionamento destes, entre vários outros comportamentos.

Dada as inúmeras possibilidade de se organizar uma aplicação quanto seus objetos, é válido considerar que algumas soluções se destaquem como exemplos de boas práticas. Este é o caso introduzido por Eric Evans com a ideia de design orientado a domínio (EVANS, 2004). Segundo o autor, o modelo de um domínio deve ter uma autoria compartilhada. Os arquitetos e desenvolvedores devem elaborar conjuntamente com especialistas desse domínio um modelo que satisfaça a todas partes. Isso pois, em teoria, mudanças de regras de negócio implicam em mudanças de código, assim como o oposto.

Partindo dessa premissa então, percebe-se que dependendo da arquitetura de um sistema, o simples fato de se alterar uma porção de código pode impactar diretamente as regras de negócios definidas. E isto é catastrófico. Sabendo disso, tem-se que o objetivo

primário do design voltado ao domínio é o isolamento desse código (correspondente ao modelo desse domínio) em uma camada específica.

Essa camada é conhecida como "Camada do Domínio" e representa o conjunto das regras de negócio do sistema. Ainda no exemplo de aviação, teria-se nessa camada, por exemplo, os objetos Aeronave, Aeroporto, Rota, e todos as outras entidades levantadas no processo de modelagem. Pode-se pensar nessa camada como a responsável por armazenar as informações do domínio, em outras palavras, seu estado.

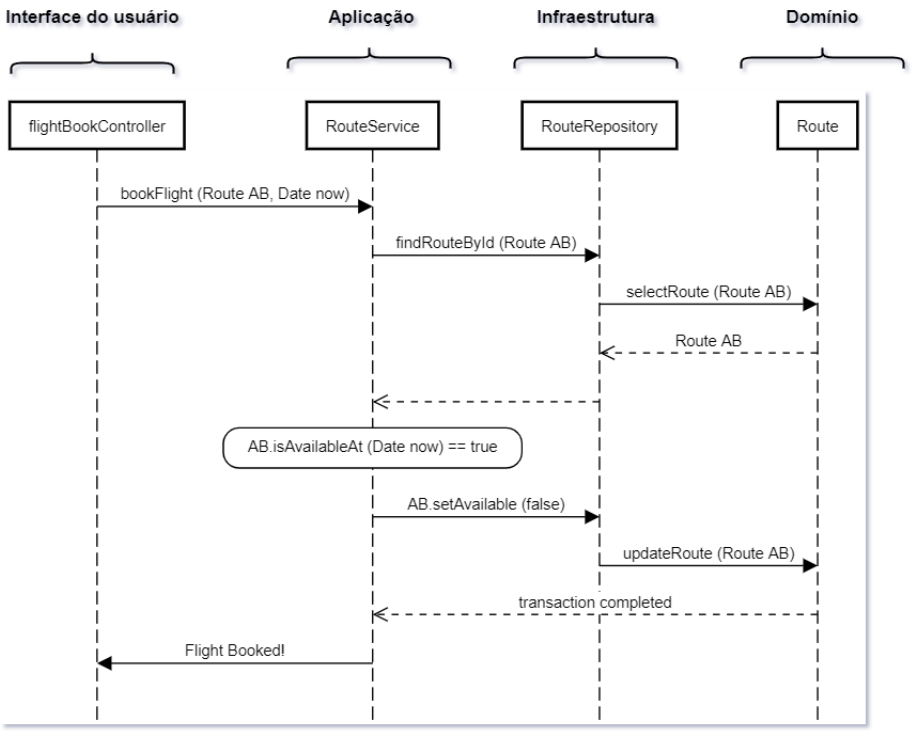
Esses dados armazenados na Camada de Domínio são operados na Camada de Aplicação. Aqui são realizadas operações sobre esses dados, como por exemplo filtrando todos os aeroportos de um certo país. Não há armazenamento de dados referente às regras de negócio.

Apoiando essas duas camadas, tem-se a Camada de Infraestrutura oferecendo comunicação entre as demais por meio de serviços comuns como persistência de dados e coordenação de mensagens.

Um exemplo de possível interação entre essas camadas é apresentado a seguir.

Um usuário ao fazer uma reserva de voo para uma certa rota, "pergunta" a um serviço especializado na Camada de Aplicação a disponibilidade desta. Esse serviço, então, busca na Camada de Domínio os objetos correspondentes a essa reserva por meio da Camada de Infraestrutura. E por meio desses objetos, o serviço consegue verificar o estado destes por meio de seus métodos e validar o pedido do usuário. Caso seja possível realizar a reserva, o serviço finalmente altera o estado desses objetos e domínio e os persiste na infraestrutura (banco de dados, por exemplo) por meio da Camada de Infraestrutura.

Figura 1 – Exemplo de interação entre as camadas de uma aplicação orientada ao domínio



Fonte: Autoria própria

Aliado a essa ideia de design voltado ao domínio, dois outros princípios importantíssimos para a construção e obtenção de um bom design serão apresentados.

O primeiro desses é o de padrões de design. Como mencionado anteriormente, transformar um modelo de regra de negócios em código OO é uma tarefa difícil. Deve-se mapear todas as relações desse modelo limitando-se às ferramentas fornecidas por uma linguagem. Esse mapeamento deve ser específico o suficiente para atender as essas regras de negócios iniciais, mas flexível o suficiente para atender futuras mudanças.

Essa dificuldade é compartilhada principalmente por desenvolvedores inexperientes em OO. Dadas as diversas possibilidades de se realizar essa codificação, esses desenvolvedores tendem a seguir princípios de outros paradigmas para essa solução, como por exemplo os de linguagens estruturadas ou procedurais.

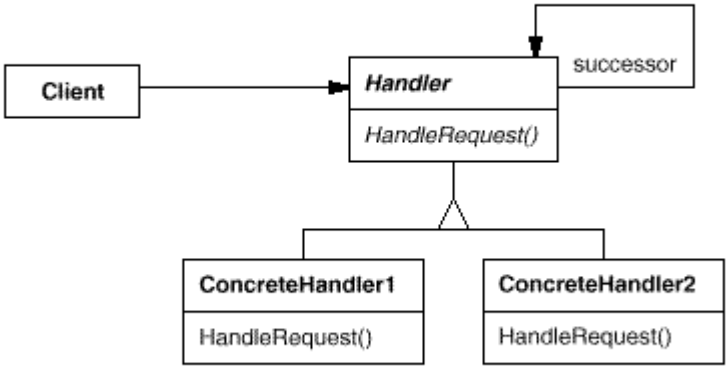
Com esse problema em mente, surge a ideia de padrões de design. Como exemplificado em (GAMMA et al., 1995), padrões de design existem em diversas outras áreas além da de tecnologia. Escritores, por exemplo, não precisam se preocupar em estabelecer uma estrutura textual inédita. Pelo contrário, eles seguem designs existentes como por exemplo a "Jornada do Herói"[1]. Uma estrutura de narrativa que envolve um herói partindo em

[1] Disponível em: <https://en.wikipedia.org/wiki/Hero's_journey>

uma aventura, e em seguida encontra uma crise e a supera e finalmente volta para casa transformado.

Assim como no exemplo anterior, pode-se pensar, então, em padrões relacionados ao desenvolvimento baseado em POO. Ideias como “representar estado por meio de objetos” e ”utilizar objetos decoradores para facilitar adição e remoção de características” são ditas padrões de design OO. Desenvolvedores utilizam desses conceitos para estruturar o mapeamento de regras de negócio em código. E alcançam assim o objetivo primário de um bom design, a facilidade em atender pedidos do cliente com baixo esforço.

Figura 2 – Exemplo do padrão de comportamento “cadeia de responsabilidades”



Fonte: Gamma et al. (1995)

Esses padrões são construídos com a finalidade de sanar problemas recorrentes quanto à estruturação de código. E carregam consigo uma descrição detalhada sobre o seu caso de uso, estrutura do padrão (Figura 2), exemplos de implementação, benefícios que fornecem e outros.

O segundo princípio a ser discutido aqui é o de código limpo. Que assim como as ideias citadas acima, busca aumentar a eficiência da construção de aplicações OO.

Segundo (MARTIN, 2018), o grande poder das linguagens OO reside no poder de controle total das dependências do sistema pelo arquiteto que dessa forma consegue alcançar a ideia de uma arquitetura *plugin*.

Esse tipo de arquitetura garante uma enorme flexibilidade e extensibilidade à aplicação. Isso, pois, o sistema é visto como um conjunto de blocos independentes e conectados, os quais representam diversas funcionalidades. Alguns desses blocos são primordiais e carregam o que é insubstituível na aplicação, porém os demais são completamente substituíveis. Além disso, blocos novos podem ser adicionados a este conjunto.

Olhando como esses blocos estão conectados, tem-se uma visão de alto nível da aplicação. Mudando essa perspectiva para um bloco em específico, percebe-se detalhes de como ele funciona individualmente, o que são detalhes de baixo nível.

Para alcançar essa arquitetura, alguns princípios são reforçados pelo autor. Estes são conhecidos como *SOLID*:

a) S - Princípio de Responsabilidade Única: Uma classe deve ter um, e somente um, motivo para mudar;

b) O - Princípio Aberto-Fechado: Objetos devem estar abertos para extensões, mas fechados a modificações;

c) L - Princípio da substituição de Liskov: Uma classe derivada deve ser substituível por sua classe base;

d) I - Princípio da Segregação da Interface: Uma classe não deve ser forçada a implementar interfaces e métodos que não irão utilizar;

e) D - Princípio da Inversão de Dependência: Uma classe estável deve depender de abstrações e não de implementações.

Com esses conceitos apresentados, compreende-se boas práticas acerca da modelagem das regras de negócio em código. Isto é, a organização entre os objetos que constituem a aplicação e a forma como estes devem ser implementados. Isso representa um corte longitudinal no sistema amarrando todos os níveis envolvidos em sua construção: transformação de um problema real em código (objetos), organização da interação e disposição desses objetos (visão alto-nível), comportamento específico desejado dos diferentes objetos envolvidos nessa aplicação (visão de baixo nível)

Apresenta-se, então, princípios de qualidade acerca desses sistemas OO considerando essas diretrizes de modelagem e construção.

2.2 Qualidade de Software

Definir a qualidade de um produto é um assunto muito amplo e complexo de ser realizado. A noção mais intuitiva em relação a qualidade de um produto de software é em relação a falta de ”bugs” deste. Essa ideia, no entanto, representa apenas uma dimensão dentro de várias existentes associadas a essa ideia de qualidade.

Esse caráter multidimensional é revelado nas definições de Crosby e Juran (1), e Gryna (2) (GALIN, 2004; KAN, 2003):

> *Software quality is: 1. The degree to which a system, component, or process meets specified requirements. 2. The degree to which a system, component, or process meets customer or user needs or expectations.*

Enquanto a primeira definição se concentra no aspecto funcional de qualidade, isto é em fatores intrínsecos ao produto, a segunda leva em conta também fatores externos como o cliente, os usuários e o processo utilizado. A primeira forma é conhecida como definição "*small q*" de qualidade e a segunda como "*big Q*" (KAN, 2003).

2.2.1 Fatores de Qualidade

Para tentar quantificar esses conceitos vários modelos foram desenvolvidos ao longo dos anos. O mais conhecido deles, talvez, é o de McCall que representa a qualidade de um software em fatores e critérios de qualidade (NAIK; TRIPATHY, 2008). Esses fatores representam características externas do sistema como sua corretude, reusabilidade, manutenibilidade entre outros. Cada um desses fatores possui atributos internos (critérios de qualidade) relacionados ao processo de desenvolvimento de software, como é o caso do critério modularidade associado ao fator manutenibilidade.

Nesse estudo, o foco estará na visão do produto, isto é, nas suas propriedades intrínsecas e na sua capacidade de atingir suas especificações. Uma hipótese que surge acerca dessas propriedades é a de que elas são diretamente proporcionais às qualidades externas deste. Em outras palavras, se a qualidade intrínseca de um produto é melhorada, sua qualidade externa também será em um certo nível.

Especificamente, o estudo focalizará três fatores de qualidade presentes no modelo de McCall: corretude, testabilidade e manutenibilidade. O primeiro desses fatores indicará se o produto atinge suas especificações, o segundo a facilidade em testá-lo e o terceiro a facilidade em alterá-lo.

2.2.2 Critérios de Qualidade

Com o intuito de medir esses comportamentos, serão utilizadas métricas associadas às subcategorias associadas a cada um desses fatores. Para manutenibilidade e testabilidade, serão usados os seguintes critérios de qualidade simplicidade e modularidade, e para o fator de corretude será usado o critério de completude.

Simplicidade indica a facilidade com que o software pode ser entendido, modularidade indica a independência dos módulos dentro deste e completude indica o quanto das especificações do produto foram alcançadas (NAIK; TRIPATHY, 2008).

2.2.3 Métricas

Para avaliar os critérios de qualidade, algumas métricas foram selecionadas. Estas são as de Chidamber-Kemerer (CK) (CHIDAMBER; KEMERER, 1994) e de Martin (MARTIN, 1994).

As métricas de Chidamber-Kemerer são amplamente utilizadas para avaliar o design de sistemas orientados a objetos com efeito em fatores de qualidade. Diversos estudos foram feitos autorizando a eficácia dessas métricas (BASILI; BRIAND; MELO, 1996). As métricas são detalhadas a seguir:

a) CBO (*Coupling Between Objects*): quantifica o número de dependências que uma classe tem;

b) DIT (*Depth Inheritance Tree*): quantifica o número de "pais" que uma classe possui. Todas as classes possuem no mínimo DIT = 1, já que herdam do tipo *java.lang.Object*;

c) NOC (*Number of Children*): quantifica o número de "filhos" que uma classe possui;

d) LCOM (*Lack of Cohesion of Methods*): quantifica quão coesos os métodos em uma classe são. Isto é, se estes operam em variáveis disjuntas, maior será o valor dessa métrica;

e) RFC (*Response for a Class*): indica a quantidade de métodos que uma classe pode invocar direta ou indiretamente;

f) WMC (*Weight Method Class*): quantifica a complexidade de uma classe dado o número de possíveis fluxos dentro desta.

As métricas de Martin garantem uma forma de analisar as inter relações entre os módulos de um sistema OO com uma visão focada nas dependências entre estas (MARTIN, 1994). Assim como as métricas de CK, estas já foram analisadas por diversas fontes (SHAIK, 2012).

As métricas são apresentadas a seguir:

a) A (*Abstractness*): quantifica o grau de abstração de um módulo;

b) Ce (*Efferent Coupling*): quantifica o total de tipos (dependências) conhecidos por um módulo;

c) Ca (*Afferent Coupling*): precisam o quanto módulos externos dependem do módulo avaliado;

d) I (*Instability*): quantifica a probabilidade do módulo sofrer alterações;

e) D (*Normalized Distance from Main Sequence*): quantifica o balanço entre abstração e instabilidade de um módulo.

2.3 Teste de Software

Como já levantado, talvez a primeira ideia associada à qualidade de um produto é sua capacidade de executar o que lhe foi proposto. Essa execução, geralmente, tem uma

margem de aceitação a falhas. Para demonstrar essa capacidade de um sistema, testes são realizados.

É importante ressaltar, no entanto, que testes não são capazes de demonstrar a corretude de um sistema por completo. Isso é evidenciado pela célebre frase de Dijkstra (NAIK; TRIPATHY, 2008):

> *Testing can only show the presence of errors, not their absence.*

Porém o custo associado a uma prova rigorosa de corretude (métodos formais, por exemplo) inviabiliza sua aplicação. Dessa forma, têm-se nos testes a melhor alternativa de garantir o correto funcionamento de um sistema.

Em relação aos testes de software, existe uma área especializada na etapa de desenvolvimento destes. Essa área engloba todas as atividades de teste realizadas pela equipe de desenvolvimento. Essas atividades são conhecidas por testes unitários, testes de componentes e teste de sistema (SOMMERVILLE, 2016). Como essas duas últimas atividades expandem os conceitos de testes unitários, o estudo se restringirá ao escopo deste.

2.3.1 Testes unitários

Testes unitários se encarregam de testar componentes de um programa como métodos ou classes. A ideia de unitário vem do fato desses métodos serem a menor unidade passível de teste. Sempre que possível é ideal que testes unitários sejam automatizados com ajuda de algum framework.

Basicamente, um teste unitário é realizado em cinco passos (NAIK; TRIPATHY, 2008). Primeiramente há a seleção do método a ser testado, então há uma seleção de valores a serem fornecidos para este, seguido de uma definição dos valores esperados após a execução do mesmo. Após isso, o método é executado com os valores selecionados e posteriormente os resultados obtidos são comparados com os valores esperados.

Como dito, é ideal que esses testes sejam automatizados. Na aplicação analisada o framework adotado é o JUnit 4, versão 4.12. Esse framework utiliza a ideia de "*runners*", os quais permitem estender as funcionalidades da ferramenta (TUDOSE, 2020). Na aplicação, os testes unitários utilizam majoritariamente ou um *runner* do framework Mockito pela anotação *@RunWith(MockitoJUnitRunner.class)* ou um *runner* do framework Spring Boot pela anotação *@RunWith(SpringRunner.class)*. Dessa forma as funcionalidades dentro desses testes diferem dado o *runner* escolhido.

2.3.2 Mock Objects

Para que os métodos dos testes unitários sejam testados, muitas vezes é necessário incluir dependências externas a esse objeto sob teste (SPADINI et al., 2019). Caso se inclua implementações reais dessas dependências, caminha-se a noção de teste de componente. Uma outra possibilidade é a inclusão de *dublês de teste* (MESZAROS, 2007). Esses dublês são utilizados justamente para "imitar" uma dependência. Entre esses possíveis dublês, tem-se:

a) *Dummy Objects*

b) *Fake Objects*

c) *Stubs*

d) *Spies*

e) *Mock Objects*

Conforme indicado por Fowler (2007), apenas esse último fornece capacidade de verificar o comportamento da aplicação em contraste com a verificação de estado citado acima como o último passo básico de um teste unitário. Esta característica de verificação de comportamento é um divisor em termos de abordagem ao desenvolvimento de testes e origina a técnica conhecida como *Mock Objects*.

É importante, nesse ponto, estar definido algumas indicações quanto às terminologias utilizadas.

O termo *mock object* (ambas iniciais em caixa baixa) será utilizado para se referir a um dos possíveis dublês de teste nomeados por Meszaros. De acordo com Sommerville (2016), *mock objects* são objetos com a mesma interface dos objetos dependentes sendo utilizados e que conseguem simular suas funcionalidades. São objetos que podem ser acessados rapidamente, sem acesso a bancos de dados ou disco.

Já o termo *Mock Objects* (ambas iniciais em caixa alta) será utilizado para se referir a técnica introduzida (FREEMAN et al., 2004). Essa técnica consiste do uso de *mock objects* aliado à prática de TDD com o intuito de descobrir interfaces entre objetos de uma aplicação, com enfoque ao design entre estes. Segundo os autores:

> *(...) [Mock Objects] is really a technique for identifying types in a system based on the roles that objects play.*

Uma das funcionalidade oferecidas por ambos os *runners* citados acima é o da criação desses mocks. Enquanto Mockito utiliza a anotação *@Mock*, Spring Boot o faz pela anotação *@MockBean*.

Além da capacidade de criar *mock objects*, o framework Mockito oferece outras funcionalidades aliadas a esta que são: stubbing de respostas, verificação de interação, configuração de *spies* e reset de mocks.

3 Contexto da Aplicação

Como descrito anteriormente, a aplicação está no domínio de aviação, mais especificamente na parte de aeroportos. A aplicação é responsável por gerir dados desses aeroportos, oferecendo serviços como armazenamento, disponibilização e validação de informações como ICAO, nome, latitude, grade de horários, aeroportos alternativos etc. É um sistema implementado utilizando Java 10 e o framework Spring Boot 2.

Em termos de design, a aplicação é um microsserviço que apresenta uma arquitetura em camadas, aplicando conceitos de design orientado ao domínio. A mensageria utilizada é o serviço Kafka (FOUNDATION, 2022b) em conjunto com o padrão de mensagem Avro (FOUNDATION, 2022a). Para interação com o banco de dados, a biblioteca jOOQ (GMBH, 2022) foi escolhida.

É importante ressaltar que diversas classes da aplicação são geradas por plugins específicos desses serviços (Avro e jOOQ).

Figura 3 – Exemplo de uma classe gerada automaticamente pelo plugin jooq-codegen (GMBH, 2022)

```
/**
 * This class is generated by jOOQ.
 */
@Generated(
    value = {
        "http://www.jooq.org",
        "jOOQ version:3.10.8"
    },
    comments = "This class is generated by jOOQ"
)
/all, unchecked, rawtypes/
public class Airport extends TableImpl<AirportRecord> {
```

Fonte: Autoria própria

A equipe responsável pelo desenvolvimento da aplicação segue o molde proposto pela metodologia Scrum. E entre os participantes da equipe, cinco são desenvolvedores. Em relação à experiência média desses desenvolvedores temos um tempo aproximado de 2 anos, embora inicialmente aplicação tenha sido construída por especialistas. É importante ressaltar que há uma certa rotatividade entre os integrantes da equipe.

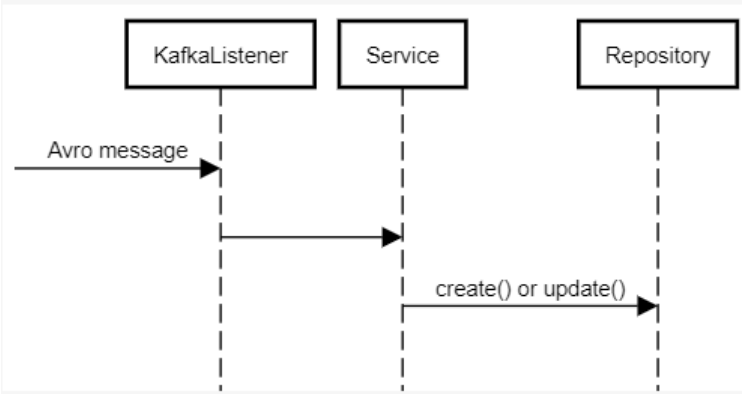
A aplicação possui 21 versões entregues ao longo de um período de 2 anos e 9 meses a um cliente internacional.

Em termos de gerenciamento de projeto, algumas decisões em termos de requisitos eram bruscas e entravam no escopo de uma *Sprint* forçadamente.

Em relação ao desenvolvimento de testes unitários, o TDD não era uma obrigação. Então alguns desenvolvedores utilizavam seus princípios enquanto outros não.

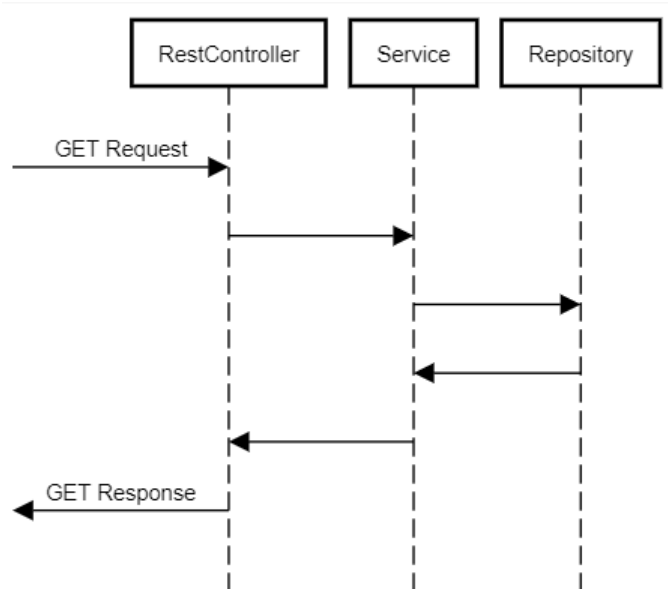
Dentre os fluxos da aplicação, dois eram possíveis. Ou via mensagens Kafka ou via requisições REST. conforme indicado nas figuras a seguir:

Figura 4 – Fluxo da aplicação como *event sourcing*



Fonte: Autoria própria

Figura 5 – Fluxo da aplicação como API



Fonte: Autoria própria

4 Análise

Com o que foi apresentado até aqui, é possível agora investigar as perguntas elencadas na introdução. Para respondê-las, diversos dados foram coletados. A obtenção destes se deu da seguinte forma:

Para obtenção de métricas relacionadas ao projeto, o plugin **Metrics Reloaded**[1] para a IDE IntelliJ e a ferramenta **CK**[2] (SPADINI et al., 2019) foram utilizadas. Ambas as ferramentas realizam uma análise estática do código fornecendo métricas como as descritas na seção anterior.

Para obter informações sobre os mocks utilizados na aplicação, a ferramenta *Find in Files* da IDE IntelliJ foi utilizada, assim como a ferramenta **MockExtractor**[3] (SPADINI et al., 2017). Com a primeira é possível descobrir todas as ocorrências no projeto de uma determinada cadeia de caracteres. Assim, para se encontrar o número de mocks declarados no sistema basta pesquisar pelas palavras chaves "@Mock" e ".mock()". Estas duas representam o total de possibilidades de se instanciar um mock em uma classe de teste. A segunda ferramenta é utilizada para determinar os tipos de dados dos mocks utilizados, isto é, as classes "mockadas" nos testes unitários.

Assim que os dados eram coletados, eles eram carregados no ambiente de desenvolvimento **R Studio** para eventuais aplicações de filtros, análises, construções de gráficos e aplicações de estatísticas.

4.1 Como é a prática de *mock objects* na aplicação?

Para responder a essa dúvida os seguintes dados foram levantados inicialmente:

- proporção entre classes de teste que utilizam mock ou não;
- quantidade total de mocks utilizados na aplicação;
- tipos das dependências "mockadas";
- distribuição dos mocks nas classes de teste;
- frameworks utilizados para criação de mocks.

Expandindo a Tabela I apresentada em Spadini et al. (2017) nas Tabelas 1 e 2, percebe-se as seguintes proporções entre testes unitários com e sem mocks, e de dependências "mockadas" e não "mockadas".

[1] Disponível em: <https://plugins.jetbrains.com/plugin/93-metricsreloaded>
[2] Disponível em: <https://github.com/mauricioaniche/ck>
[3] Disponível em: <https://github.com/ishepard/MockExtractor>

Tabela 1 – Comparação entre diferentes aplicações Java

Projeto	**Nº de classes**	**LOC**	**Nº de testes unitários**	**Nº de testes unitários com mocks**	**Proporção entre testes com e sem uso de mocks**
Sonarqube	5771	701k	2034	652	0.32
Spring framework	6561	997k	2020	299	0.14
VRaptor	551	45k	126	80	0.63
Alura	1009	75k	239	91	0.38
Airport Centre	395	208k	137	30	0.21

Tabela 2 – Comparação entre as dependências "mockadas" nessas aplicações

Projeto	**Nº de dependências "mockadas"**	**Nº de dependências não "mockadas"**	**Proporção entre dependências "mockadas" e não "mockadas"**
Sonarqube	1411	12136	0.116
Spring framework	670	21908	0.03
VRaptor	258	1075	0.24
Alura	229	1436	0.159
Airport Centre	40	1299	0.03

A primeira Tabela indica um uso moderado de mocks dentro da aplicação quando comparado aos demais sistemas. Já a segunda indica que apenas uma porcentagem muito baixa das classes da aplicação é "mockada". Isso pode ser explicado pela grande presença de classes geradas por plugins externos, como discutido na seção anterior. Essas classes desempenham um papel auxiliar na aplicação e não são consideradas tanto na cobertura de testes quanto no desenvolvimento de testes unitários.

Utilizando a funcionalidade "Find in Files" da IDE IntelliJ, 101 mocks foram encontrados dentro dessas trinta classes de teste, como é mostrado na Tabela 3. A Tabela exibe também os frameworks utilizados para instanciar cada um destes mocks.

A escolha do framework está intimamente associada ao contexto do teste em questão. Testes relacionados às classes de Controller (REST) fazem o uso em conjunto das anotações *@RunWith(SpringRunner.class)* e *@WebFluxTest(controllers=<Controller.class>)*. Isso porque essas anotações facilitam a construção do ambiente de teste, no contexto de beans do framework Spring. Com o uso da primeira anotação, os mocks são integrados ao teste por meio da anotação *@MockBean*, exclusivamente. Um outro cenário que essa anotação é utilizada é em testes de integração, no qual a anotação *@RunWith(SpringRunner.class)* também é utilizada.

Os demais testes que utilizam a anotação *@RunWith(MockitoJUnitRunner.class)* são testes unitários que não implementam anotações adicionais para configurações extras.

Tabela 3 – Frameworks utilizados na criação de *mock objects*

Framework utilizado	**Nº de mocks**
Mockito	91
Spring Boot	10
Total	101

Buscando um entendimento acerca de como esses mocks são utilizados nas classes de teste, uma análise quantitativa foi feita sobre as funcionalidades mais utilizadas da API fornecida pelo framework Mockito[4]. Essa análise foi realizada manualmente, passando por todas as classes de teste que utilizam mocks e verificando quais funcionalidades estes implementam.

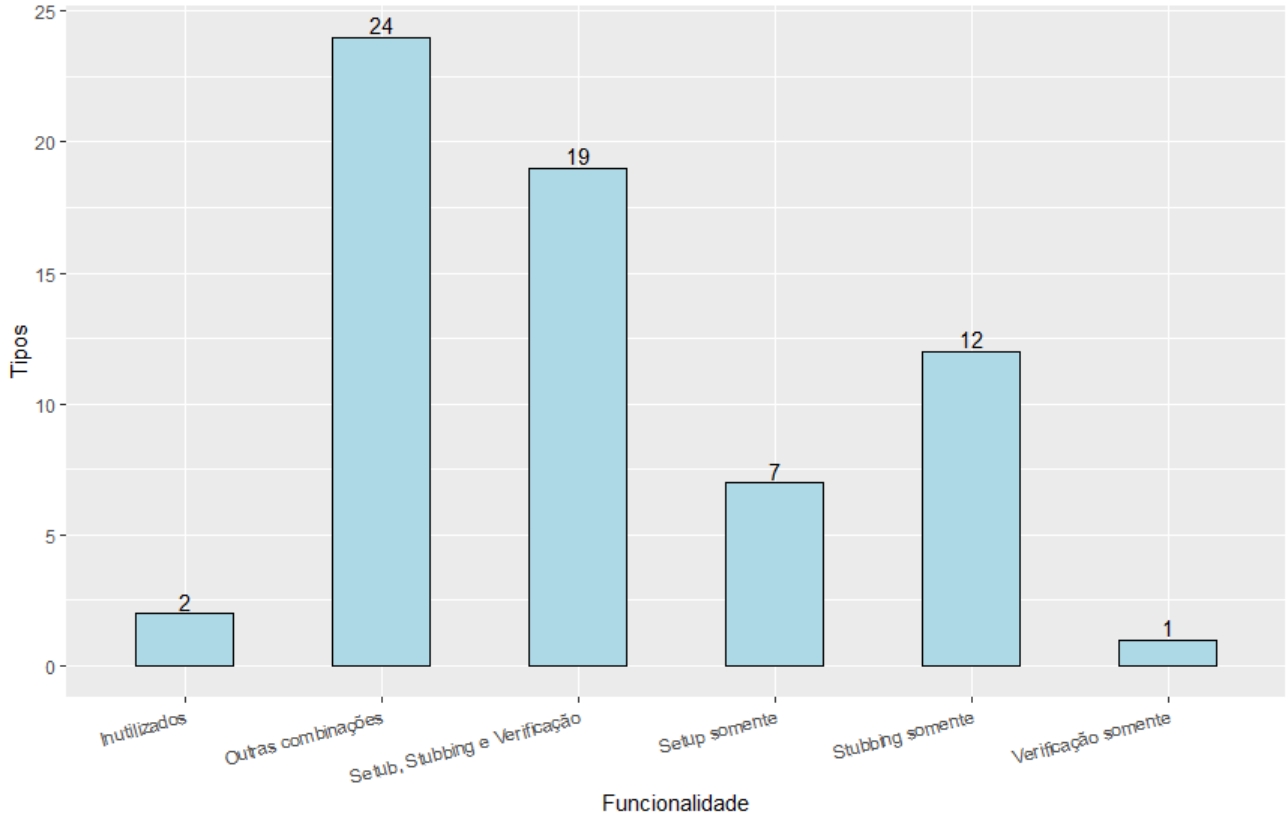
Para realizar essa análise, as seguintes categorias foram consideradas:

1. **Stubbing** - contendo os seguintes métodos da API: when(), then(), thenReturn()
2. **Verificação** - contendo os seguintes métodos da API: verify(), verifyZeroInteractions(), verifyNoMoreInteractions()
3. **Setup** - Mocks utilizados no setup de um teste, como ilustrado no seguinte trecho de código

```
@Before
public void setup() {
    service = new TimezoneService(repositoryMock, airportServiceMock);
}
```

[4] Disponível em: <https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html>

Figura 6 – Funcionalidades utilizadas por cada mock



Fonte: Autoria própria

Observe na Figura 6 que o total de tipos é superior ao exibido na Tabela 2. Isso ocorre, pois um mesmo tipo pode ser utilizado em diferentes classes de teste. A categoria "Outras combinações" inclui todas as combinações possíveis entre às três funcionalidades categorizadas. Isto é, *Setup* e *Stubbing*, *Setup* e Verificação, e *Stubbing* e Verificação. Como os valores obtidos por essas combinações não eram expressivos, todas elas foram agrupadas em uma única coluna.

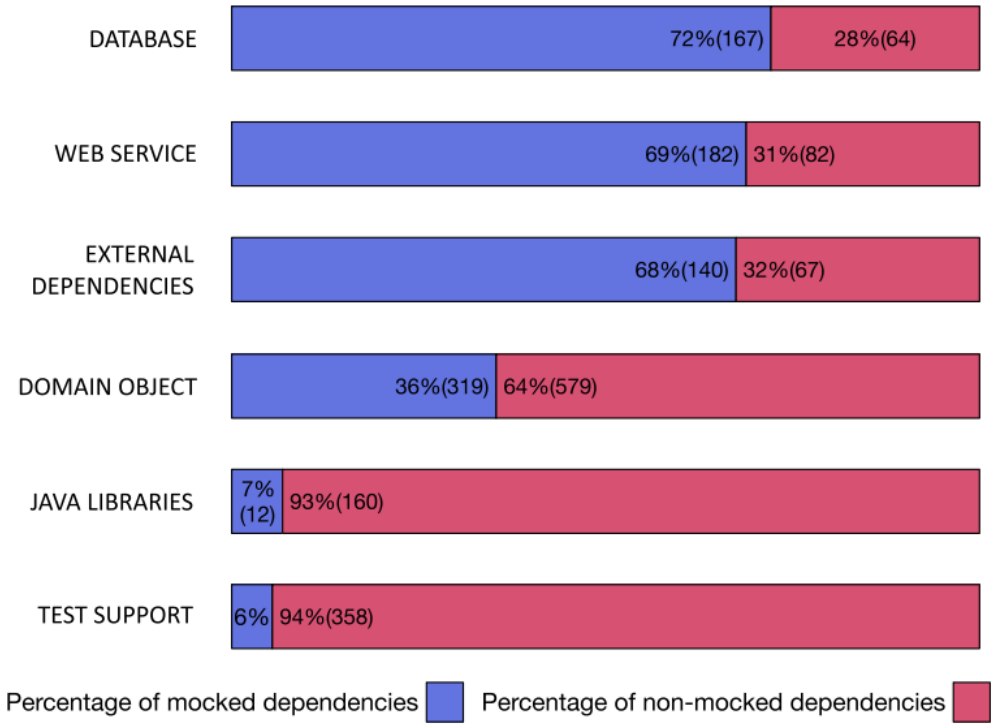
Em (FOWLER, 2007), o autor aponta como o uso de mocks favorece uma verificação de comportamento dentro de uma unidade de teste em contraste com a tradicional verificação de estado. Adicionalmente, o autor levanta fatores que podem levar um desenvolvedor a optar por outros tipos de dublês de teste, um "stub" por exemplo. Por fim, essas abordagens são comparadas com o intuito de descobrir quando se deve utilizar um mock ou alguma outra forma de dublê. O autor é cauteloso ao concluir que não existem vantagens claras de se adotar o uso de mocks somente.

Observando a Figura 6, percebe-se também que o uso de mocks na aplicação serve a diversos propósitos, se opondo a ideia de Fowler de se usar dublês específicos em cada situação. Isso porque esses mocks são usados em diferentes aspectos de testes. Nota-se, por

exemplo, que 12 tipos de mock fazem o uso exclusivo da funcionalidade *stubbing*, isto é, são utilizados unicamente para alcançar uma verificação de estado. No entanto, 19 outros tipos são utilizados com uma finalidade distinta, a de verificação de comportamento. Isso revela um possível uso indiscriminado de mocks.

Além disso, em 7 classes de teste existem mocks que são utilizados somente para configurar o teste em questão. Esse comportamento se opõe ao da maioria dos desenvolvedores e que é analisado em (SPADINI et al., 2017). Como pode ser observado na categoria "TEST SUPPORT" da Figura 7, desenvolvedores utilizam mocks somente 6% das vezes para auxiliar um teste. Essa característica da aplicação indica um possível uso imaturo da técnica.

Figura 7 – Dependências *mockadas* pelos desenvolvedores



Fonte: Spadini et al. (2017)

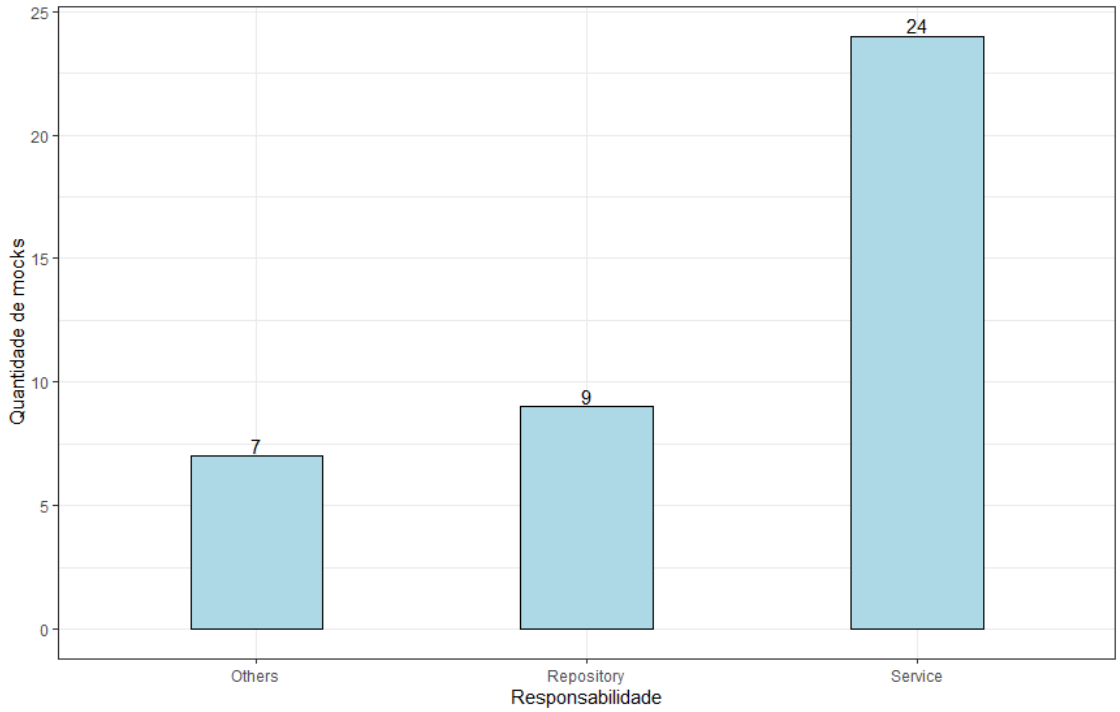
Dessas 40 dependências "mockadas" a distribuição em relação às responsabilidades das classes é apresentada na Figura 8.

Dentro da categoria "Outros" têm-se tipos que são de bibliotecas externas. Esse dado contraria o expresso por Freeman et al. (2004) ao afirmar que deve-se "mockar" somente vizinhos imediatos da unidade sob teste. A própria página do framework Mockito também sugere o mesmo[5].

[5] Mais em: <https://github.com/mockito/mockito/wiki/How-to-write-good-tests>

Isso, no entanto, se opõe a prática dos desenvolvedores novamente. Como mostrado por Spadini et al. (2017), 68% dos desenvolvedores preferem utilizar *mock objects* para dependências externas. Conforme indicado na categoria "EXTERNAL DEPENDENCIES" da Figura 7.

Figura 8 – Distribuição das 40 dependências "mockadas" pela aplicação em relação às responsabilidades desses tipos

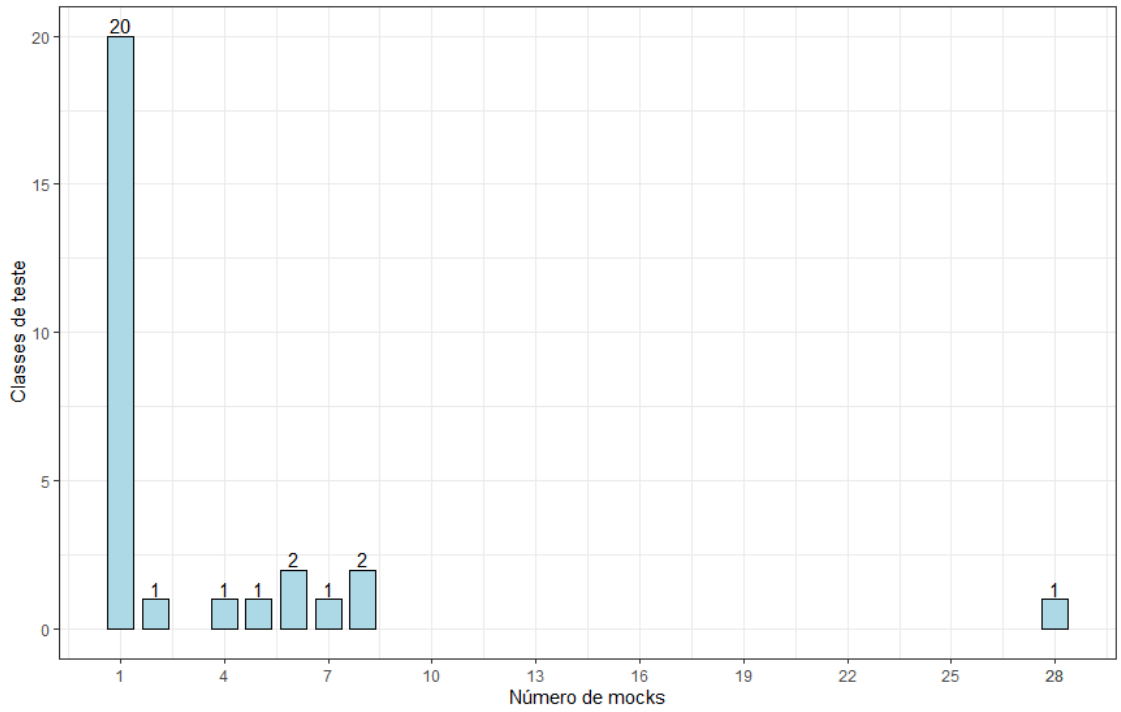


Fonte: Autoria própria

4.2 Qual a relação dessa prática (de *mock objects*) com detalhes de design OO?

Para entender como os 101 mocks utilizados na aplicação estão distribuídos nas 30 classes de teste, uma análise foi feita buscando entender essa distribuição. O resultado é apresentado na Figura 9.

Figura 9 – Distribuição dos 101 mocks nas 30 classes de teste que os implementam



Fonte: Autoria própria

Fica claro nesta imagem, como a variância dessa distribuição é expressiva (com média de 3.1 mocks por classe de teste). Enquanto 20 classes de teste apresentam somente um tipo "mockado", outra apresenta 28.

Observando essa distribuição constate-se, também, uma evidência expressiva em termos de más práticas de design OO. Por que uma única classe de teste apresenta tantos mocks? Em (FREEMAN et al., 2004), o autor aponta que para uma prática eficiente de *Mock Objects*, somente os vizinhos da classe sob teste devem ser "mockados". Além disso, há uma recomendação de se evitar o uso exagerado de mocks em um único teste.

Considerando, então, essa relação entre mocks e dependências "vizinhas" de uma classe sob teste, temos o primeiro indício que essa quantidade desproporcional e concentrada de mocks em uma camada da aplicação pode estar evidenciando um problema de dependências.

Continuada a análise, buscou-se entender como esses 101 mocks estavam distribuídos em termos de responsabilidade das classes de teste, isto é, existe predominância de mocks em alguma camada específica da aplicação?

Para responder a essa questão, as 30 classes de teste que utilizam mocks foram separadas em 4 categorias:

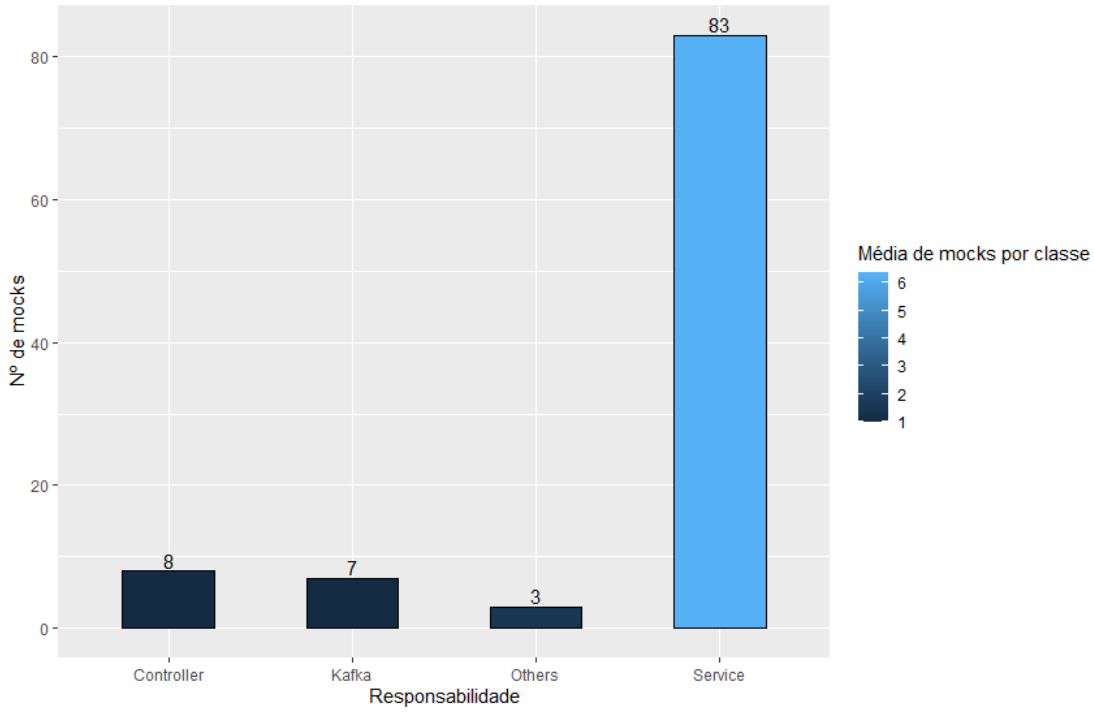
1. **Service** (13 classes de teste) - Classes de teste que contém a string “Service” no nome;

2. **Controller** (8 classes de teste) - Classes de teste que possuem a string “Controller” no nome;

3. **Kafka** (7 classes de teste) - Classes de teste que possuem as strings “Listener” ou “Producer” no nome;

4. **Outros** (2 classes) - As demais classes.

Figura 10 – Distribuição dos 101 mocks em relação à responsabilidade das classes de teste que os utilizam



Fonte: Autoria própria

Observando a Figura 10 percebe-se que há uma clara concentração de mocks em uma camada específica dos testes da aplicação. Classes de teste na camada de Serviço possuem em média 5 mocks a mais do que classes de teste de outras camadas.

Para clarear o porquê dessas classes utilizarem tantos mocks, uma análise foi feita utilizando as métricas de Chidamber-Kimerer e de Martin.

Para realizar tal análise, foram filtradas do resultado fornecido pela ferramenta CK as classes que continham a substring “Service” em seu nome (contemplando classes de

teste e produção). E em seguida aplicou-se a média sobre os valores obtidos. O mesmo processo foi feito porém considerando o conjunto de todas as demais classes da aplicação.

A Tabela 4 indica a média obtida para cada uma das métricas.

Tabela 4 – Métricas de Chidamber-Kemerer

Classes	**CBO**	**DIT**	**LCOM**	**NOC**	**RFC**	**WMC**
Camada de Serviço	15.7	1	1.42	0	55.7	26.7
Demais classes	8.93	2.07	6.65	0	25.7	21.5

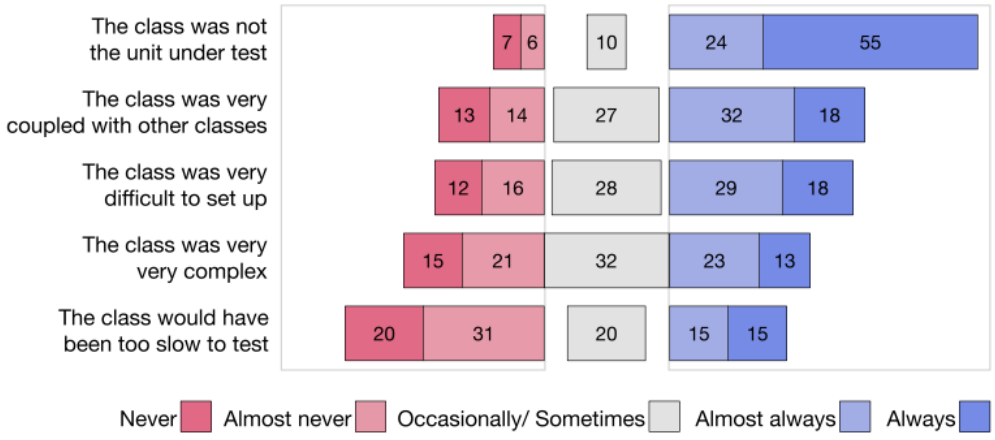
Examinando a métrica CBO, percebe-se que as classes da camada de Serviço possuem em média quase o dobro de dependências das demais classes. Nota-se, então, que a implementação de mocks nessa camada da aplicação está associada ao número de dependências das classes dessa camada.

Essa prática vai ao encontro do que é levantado por Spadini et al. (2017). Estudo o qual revela que desenvolvedores tendem a usar mocks para classes com muitas dependências e alto acoplamento. Essa informação é mostrada na Figura 11, na categoria "*(*The class was very coupled with other classes)".

Essa resposta vinda dos desenvolvedores, no entanto, vai contra o que é proposto em (LANGR, 2004). O autor comenta:

> *A final reason often cited for mocking is that the system contains a large dependency chain. (...) Having to construct a few dependent objects is acceptable. But it demonstrates a design flaw if building the test context becomes onerous. While having a large dependency chain seems like a legitimate reason to code mocks, avoid the temptation.*

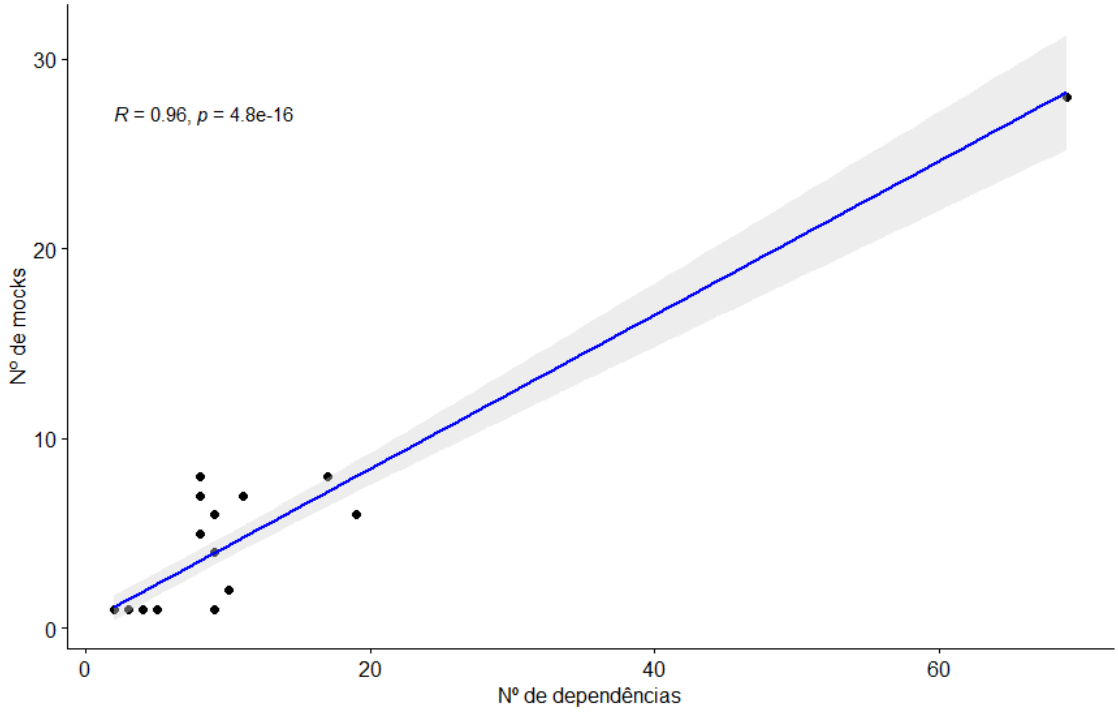
Figura 11 – Situações em que desenvolvedores fazem uso de *mock objects*



Fonte: Spadini et al. (2017)

Para entender se existe uma possível correlação positiva entre o número de mocks em uma classe de teste e a quantidade de dependências dessa mesma classe em produção, o seguinte gráfico foi construído utilizando o índice de correlação de Pearson.

Figura 12 – Correlação entre o número de mocks de uma classe de teste e o número de dependências dessa classe correspondente em produção



Fonte: Autoria própria

Como explicitado pela imagem, o índice de correlação com valor de 0.96 juntamente com o *p-valor* baixíssimo, sugerem fortemente essa correlação positiva no contexto da aplicação em análise.

Voltando à Tabela 4, percebe-se também uma grande diferença na métrica RFC entre as classes analisadas. Nota-se que as classes da camada de Serviço invocam mais métodos quando comparadas às demais. Com isso em mente, pode-se vislumbrar que os testes destas, aliados ao uso de *mock objects*, implicará em uma quantidade maior de *stubbing* e verificação de comportamento.

Por fim, uma análise sobre as métricas de Martin (MARTIN, 1994) foi realizada.

Tabela 5 – Métricas de Martin

Classes	**A**	**Ce**	**Ca**	**D**	**I**
Camada de Serviço	0	559	4219	0.12	0.88
Demais classes	0.008	419.94	349.55	0.50	0.50

Fica claro pela Tabela acima, como a aplicação vai contra princípios básicos de design. Como é o caso do princípio de inversão de dependências. Isso é verificado pela primeira métrica "Abstração". Percebemos que a camada de Serviço da aplicação faz uso somente de classes concretas, inutilizando a flexibilidade oferecida por classes abstratas e interfaces. Como consequência temos uma altíssima instabilidade dessa camada, expressa pela última métrica "I".

Para entender como essa instabilidade afeta o desenvolvimento, um script em Python foi criado se baseando no funcionamento do comando *git rev-list*. Esse comando pode ser utilizado para buscar quantos commits alteraram um determinado arquivo. O algoritmo do script é descrito a seguir:

1. Busca e armazenamento de todos os arquivos *.java* dentro de um diretório específico (/src e /test foram utilizados);

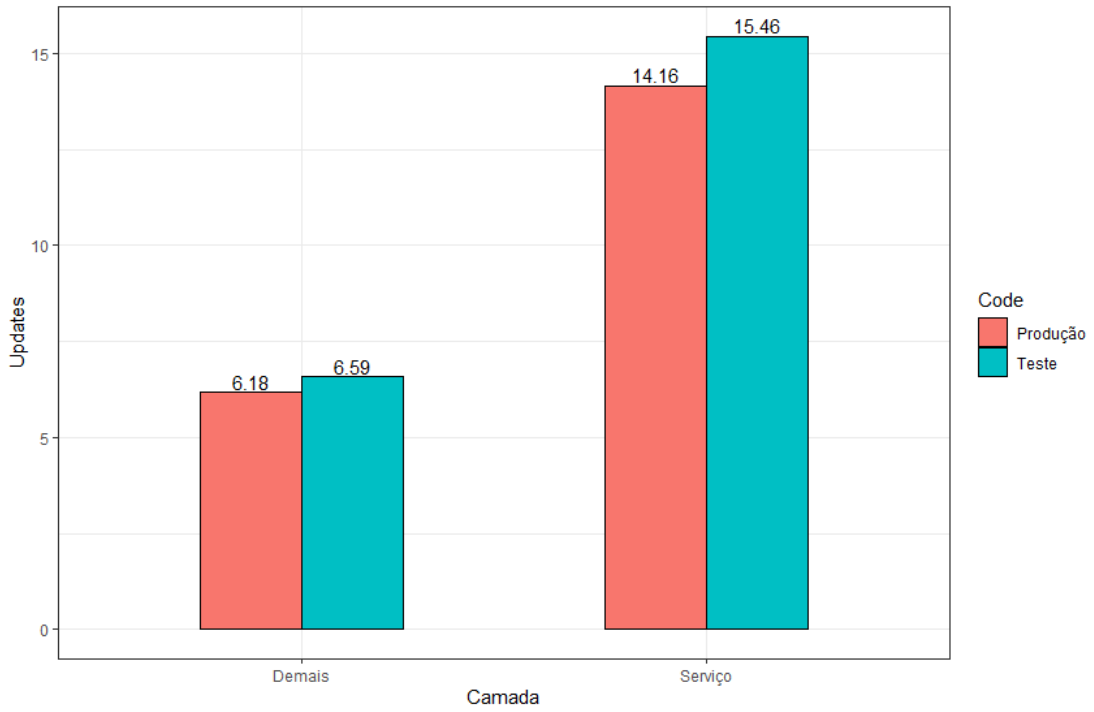
2. Para cada arquivo encontrada no passo 1, o seguinte comando é executado:

```
git rev-list HEAD --count | <caminho encontrado no passo 1>
```

3. Após essa execução, o nome desse arquivo e o resultado obtido no passo 2, são escritos em uma linha de uma tabela .csv

Com a execução do script fica explícito quantas vezes cada classe foi alterada ao longo do desenvolvimento. Os resultados estão indicados na Figura 13:

Figura 13 – Número de alterações médias das classes da aplicação



Fonte: Autoria própria

Como se percebe na Figura 13, as classes tanto de teste como de produção na camada de Serviço sofrem 2.5 vezes mais alterações do que as demais. Um indicativo claro de um design ineficiente. Esse comportamento instável afeta diretamente o uso de mocks, já que refatorações simples podem escalar rapidamente como levantando em (LANGR, 2004).

A Tabela 5 evidencia novamente a disparidade em termos de dependências entre as classes dessa camada com as demais. E como a prática de *mock objects* se distancia em muito do proposto em (FREEMAN et al., 2004). Nesse, o autor aponta:

> *We stated that it encouraged better structured tests and, more importantly, improved domain code by preserving encapsulation, reducing dependencies and clarifying the interactions between classes.*

Com os dados apresentados, é evidente que vários fatores de design foram desconsiderados no desenvolvimento da aplicação. Isso pode ser resumido nos seguintes aspectos:

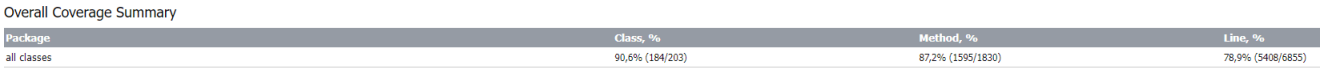
Segundo (EVANS, 2004) a camada de Serviço deve servir como abstração que provê funcionalidade relacionadas à camada de Domínio. Este conceito relaciona-se diretamente com a ideia de inversão de dependência, a letra D da sigla SOLID (MARTIN, 2018). A ideia desse princípio é justamente tornar implementações de alto-nível independentes das de

baixo-nível por meio do uso de interfaces, assegurando assim a ideia de baixo acoplamento entre estas.

Adicionalmente, (GAMMA et al., 1995) estabelecem diversos padrões de comportamento que definem formas de se organizar uma classe e suas dependências de forma a minimizar o acoplamento destas.

Conclui-se então, que o uso de mocks nessa aplicação está estreitamente ligado à ideia de design da mesma. Isso é corroborado pelo fato de que uma camada da aplicação com design ineficiente apresenta muito mais mocks e é "mockada" mais vezes, mesmo considerando um possível uso impróprio desses. Mas independente desses fatores, a corretude do mesmo não é necessariamente prejudicada, conforme indicado pela cobertura de testes na Figura a seguir:

Figura 14 – Cobertura dos testes unitários da aplicação



Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	90,6% (184/203)	87,2% (1595/1830)	78,9% (5408/6855)

Fonte: Autoria própria

5 Ameaças à validade

5.1 Validade de constructo

Ameaças à validade de constructo se referem aos instrumentos de pesquisa:

1. A análise realizada apoia-se em diversas ferramentas prontas. O correto funcionamento destas não é assegurado pelo estudo feito. Esse caráter é mitigado por: (1) as ferramentas utilizadas são conhecidas na literatura existente no assunto, (2) inspeção manual de amostras dos dados obtidos por essas ferramentas;

2. O script desenvolvido em Python analisa somente a quantidade de alterações de uma dada classe Java, mas não leva em conta o conteúdo alterado. Dessa forma, existe a possibilidade de classes não serem representativas do atributo de qualidade manutenibilidade dadas as alterações feitas nessas. Esse caráter é mitigado por uma inspeção manual de uma amostra dos dados levantados.

5.2 Validade interna

Ameaças à validade interna se referem a fatores desconsiderados e que podem afetar as variáveis e as relações investigadas:

1. Algumas etapas da análise foram realizadas de forma manual, e por isso, a existência de ruído nos dados utilizados não é descartada;

2. Em relação às classes de produção, a análise considerou a totalidade dessas. Dentro desse total, um conjunto é referente a classes geradas automaticamente por plugins externos, conforme indicado no Capítulo 3. Dada a natureza dessas classes, os resultados podem indicar problemas de design nestas ao invés de revelar más práticas dos desenvolvedores da aplicação.

5.3 Validade externa

Ameaças à validade externa se referem à generalização dos resultados:

1. Os desenvolvedores da aplicação analisada são de relativa rotatividade e possuem experiência diversa quanto ao POO. Dessa maneira, o uso de *mock objects* na aplicação está associado a um contexto de alta variância;

2. A prática de *mock objects* no sistema analisado está limitada principalmente a: (1) tecnologias utilizadas, (2) metodologia de desenvolvimento de testes adotada pela equipe e (3) domínio da aplicação. Para uma generalização dos resultados, uma análise considerando essas diferentes variáveis deve ser conduzida;

3. O sistema analisado representa somente uma fração da totalidade de aplicações existentes. Para generalizar os resultados, uma amostra representativa deve ser escolhida e investigada.

6 Conclusões

6.1 Fatores de qualidade

Utilizando as métricas obtidas, podemos agora, avaliar alguns aspectos da qualidade interna do produto.

Observando a métrica RFC (Tabela 4), percebemos que as classes da aplicação, possuem um número elevado de métodos potencialmente invocados. Isso vale ainda mais para as classes da camada de Serviço. O valor dessa métrica é uma representação da falta de simplicidade desse sistema. Como foi apresentado, esse critério está diretamente ligado ao fator de qualidade **testabilidade**. Concluí-se, então, que existe uma dificuldade em se testar funcionalidades da aplicação. Essa dificuldade é expressa pela instabilidade das classes de teste, como mostrado na figura 12.

Com o uso das métricas CBO, LCOM (Tabela 4), Ce e Ca (Tabela 5), percebe-se que embora as classes apresentem uma coesão considerável, o acoplamento entre elas acaba por enfraquecer em muito o critério de qualidade modularidade. Como consequência direta, tem-se que o traço de **manutenibilidade** deste é altamente prejudicado.

Por fim pode-se concluir que o fator de qualidade **completude** independe da aplicação de boas práticas de design OO, visto a corretude deste (Figura 14).

6.2 Considerações finais

Utilizando o modelo conhecido como *Factor-Criteria-Metrics* consegue-se avaliar alguns fatores de qualidade do design da aplicação. Essa avaliação foi embasada principalmente por uma análise sobre o uso de *mock objects* no sistema.

É válido ressaltar que o modelo *Factor-Criteria-Metrics* não é o único utilizado para avaliação de qualidade. Uma outra proposta notável é conhecida como *Factor-Strategy Quality Model* (MARINESCU, 2005).

Como revelado pelos resultados, a prática de *mock objects* na aplicação parece estar intimamente ligada à qualidade do seu design. Os resultados indicam também como o fator de qualidade "manutenibilidade" é prejudicado no sistema. E de acordo com (GLASS, 2001):

> *Maintenance typically consumes about 40 to 80 percent (60 percent average) of software costs. Therefore, it is probably the most important life cycle phase.*

Com isso em mente, é válido considerar que o desenvolvimento da aplicação embora

seja em uma linguagem OO. Muitas vezes desconsidera suas principais virtudes. Isso é reforçado pela existência de um "objeto Deus" na aplicação. Um objeto que possui dependências de diversos tipos distintos, e é representado pela classe de teste com 28 tipos de mocks. Este é um exemplo de *code smell* que indica uma inclinação dos desenvolvedores a adoção de práticas consideradas procedurais ou estruturadas.

Em relação ao uso de mocks na aplicação, fica claro que o seu uso diverge de algumas literaturas. As recomendações feitas em (FREEMAN et al., 2004) e (LANGR, 2004) não são seguidas na grande maioria dos casos. Isso porque os mocks são utilizados como facilitadores para a construção de testes, ao invés de ferramentas para elaboração de design. Essa prática, no entanto, parece estar alinhada com o que é feito pela maioria dos desenvolvedores (SPADINI et al., 2019).

Isso evidencia como a teoria e a prática sobre mock objects muitas vezes se desencontram. [1]

Uma ideia que surge nesse contexto é a de que *mock objects* servem a diversos propósitos. Um desses, todavia, parece estar associado a uma ideia primordial ao desenvolvimento OO que é o de design. E assim é válido considerar que a aplicação da técnica de *Mock Objects* por programadores inexperientes em OO pode servir como um reforçador de boas práticas.

6.3 Lições Aprendidas

1. *Mock objects* oferecem aos desenvolvedores não somente a capacidade de simular dependências, mas também a oportunidade de exercitarem conceitos importantes de design OO e qualidade de software;

2. O custo de manutenção associado ao uso de *mock objects* pode ser determinante para se avaliar a forma como os testes são desenvolvidos em uma aplicação;

3. A literatura sobre *mock objects* e o uso desses na prática não são convergentes.

4. A forma como *mock objects* são utilizados em um sistema pode servir como fator preditivo da qualidade de design OO dessa aplicação;

5. O design ineficiente de uma aplicação não implica em uma menor corretude do mesmo.

[1] Fowler, Becker e Heinemeier discutem sobre possíveis cenários das práticas de TDD e de mock objects em: <https://martinfowler.com/articles/is-tdd-dead/>. Essa discussão é complmentada em: <https://www.thoughtworks.com/en-us/insights/blog/mockists-are-dead-long-live-classicists>

Referências

BASILI, V.; BRIAND, L.; MELO, W. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, v. 22, n. 10, p. 751–761, out. 1996. ISSN 0098-5589, 1939-3520, 2326-3881. Disponível em: <https://ieeexplore.ieee.org/document/544352/>. Citado na página 22.

BUDGEN, D. *Software design: creating solutions for ill-structured problems*. Third edition. Boca Raton: CRC Press, 2021. (Chapman & Hall/CRC innovations in software engineering). ISBN 978-1-315-30007-8. Citado 2 vezes nas páginas 13 e 15.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, IEEE, v. 20, n. 6, p. 476–493, 1994. Citado na página 21.

EVANS, E. *Domain-driven design: tackling complexity in the heart of software*. Boston: Addison-Wesley, 2004. ISBN 978-0-321-12521-7. Citado 2 vezes nas páginas 16 e 40.

FOUNDATION, A. S. *Apache Avro*. 2022. Disponível em: <https://avro.apache.org/docs/current/>. Acesso em: 10 abr 2022. Citado na página 27.

FOUNDATION, A. S. *Apache Kafka*. 2022. Disponível em: <https://kafka.apache.org/intro>. Acesso em: 10 abr 2022. Citado na página 27.

FOWLER, M. Mocks aren't stubs. *Online article at martinfowler. com http://bit. ly/18BPLE1*, 2007. Citado 3 vezes nas páginas 11, 24 e 32.

FREEMAN, S. et al. Mock roles, not objects. In: *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. [S.l.: s.n.], 2004. p. 236–246. Citado 6 vezes nas páginas 11, 24, 33, 35, 40 e 46.

GALIN, D. *Software quality assurance*. Harlow, England ; New York: Pearson Education Limited, 2004. ISBN 978-0-201-70945-2. Citado na página 20.

GAMMA, E. et al. *Design patterns: elements of reusable object-oriented software*. Reading, Mass: Addison-Wesley, 1995. (Addison-Wesley professional computing series). ISBN 978-0-201-63361-0. Citado 3 vezes nas páginas 18, 19 e 41.

GLASS, R. L. Frequently forgotten fundamental facts about software engineering. *IEEE software*, IEEE Computer Society, v. 18, n. 3, p. 112, 2001. Citado na página 45.

GMBH, D. G. *jOOQ*. 2022. Disponível em: <https://www.jooq.org/>. Acesso em: 10 abr 2022. Citado 2 vezes nas páginas 5 e 27.

HORSTMANN, C. S. *Core Java*. Eleventh edition. Boston: Pearson, 2019. OCLC: on1012801820. ISBN 978-0-13-516630-7 978-0-13-516631-4. Citado na página 13.

KAN, S. H. *Metrics and models in software quality engineering*. 2nd ed. ed. Boston: Addison-Wesley, 2003. ISBN 978-0-201-72915-3. Citado 2 vezes nas páginas 20 e 21.

KAY, A. *Alan Kay over OOP*. 2022. <http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en>. Disponível em: <http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en>. Acesso em: 03 abr 2022. Citado na página 15.

LANGR, J. Don't mock me: Design considerations for mock objects. In: CITESEER. *Agile Development Conference*. [S.l.], 2004. v. 2004. Citado 4 vezes nas páginas 11, 37, 40 e 46.

MARINESCU, R. Measurement and quality in object-oriented design. In: IEEE. *21st IEEE International Conference on Software Maintenance (ICSM'05)*. [S.l.], 2005. p. 701–704. Citado na página 45.

MARTIN, R. Oo design quality metrics. *An analysis of dependencies*, v. 12, n. 1, p. 151–170, 1994. Citado 3 vezes nas páginas 21, 22 e 38.

MARTIN, R. C. *Clean architecture: a craftsman's guide to software structure and design*. London, England: Prentice Hall, 2018. (Robert C. Martin series). OCLC: on1004983973. ISBN 978-0-13-449416-6. Citado 4 vezes nas páginas 13, 16, 19 e 40.

MESZAROS, G. *xUnit test patterns: refactoring test code*. Upper Saddle River, NJ: Addison-Wesley, 2007. (The Addison-Wesley signature series). OCLC: ocm77821686. ISBN 978-0-13-149505-0. Citado 2 vezes nas páginas 11 e 24.

NAIK, K.; TRIPATHY, P. *Software testing and quality assurance: theory and practice*. Hoboken, N.J: John Wiley & Sons, 2008. OCLC: ocn166380555. ISBN 978-0-471-78911-6. Citado 3 vezes nas páginas 11, 21 e 23.

SCHILDT, H. *Java The Complete Reference, Twelfth Edition, 12th Edition*. [s.n.], 2021. OCLC: 1284987392. ISBN 978-1-260-46342-2. Disponível em: <https://learning.oreilly.com/library/view/-/9781260463422/?ar>. Citado na página 14.

SEBESTA, R. W. *Concepts of programming languages*. Eleventh edition, global edition. Boston Munich: Pearson, 2016. (Always learning). ISBN 978-1-292-10055-5. Citado na página 14.

SHAIK, A. Object Oriented Software Metrics and Quality Assessment: Current State of the Art. *International Journal of Computer Applications*, v. 37, p. 10, 2012. Citado na página 22.

SOMMERVILLE, I. *Software engineering*. 10. ed., global ed. ed. Boston Munich: Pearson, 2016. (Always learning). ISBN 978-1-292-09613-1. Citado 2 vezes nas páginas 23 e 24.

SPADINI, D. et al. To Mock or Not to Mock? An Empirical Study on Mocking Practices. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. Buenos Aires, Argentina: IEEE, 2017. p. 402–412. ISBN 978-1-5386-1544-7. Disponível em: <http://ieeexplore.ieee.org/document/7962389/>. Citado 4 vezes nas páginas 29, 33, 34 e 37.

SPADINI, D. et al. Mock objects for testing java systems: Why and how developers use them, and how they evolve. *Empirical Software Engineering*, v. 24, n. 3, p. 1461–1498, jun. 2019. ISSN 1382-3256, 1573-7616. Disponível em: <http://link.springer.com/10.1007/s10664-018-9663-0>. Citado 4 vezes nas páginas 11, 24, 29 e 46.

TUDOSE, C. *JUnit in action*. Third edition. Shelter Island, NY: Manning Publications Co, 2020. ISBN 978-1-61729-704-5. Citado na página 23.

WEISFELD, M. A. *The object-oriented thought process*. Fourth edition. Upper Saddle River, NJ: Addison-Wesley, 2013. ISBN 978-0-321-86127-6. Citado na página 15.