

UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS  
UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO  
ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

George Redivo Pinto

ESTUDO DE CASO DA CORRELAÇÃO ENTRE COBERTURA DE CÓDIGO E  
FALHAS REPORTADAS EM UM SISTEMA OPERACIONAL

Porto Alegre

2023

UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS  
UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO  
ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

George Redivo Pinto

ESTUDO DE CASO DA CORRELAÇÃO ENTRE COBERTURA DE CÓDIGO E  
FALHAS REPORTADAS EM UM SISTEMA OPERACIONAL

Trabalho apresentado à disciplina Engenharia de Software Aplicada, pelo Curso de Especialização em Engenharia de Software da Universidade do Vale do Rio dos Sinos - UNISINOS, ministrada pela professora Josiane Brietzke Porto.

Porto Alegre

2023

# Estudo de Caso da Correlação Entre Cobertura de Código e Falhas Reportadas em um Sistema Operacional

<sup>1</sup>George Redivo Pinto

<sup>1</sup> Universidade do Vale do Rio dos Sinos (UNISINOS)  
Porto Alegre – RS – Brasil

**Abstract.** *Software testing is an important step in software development cycle. This step aims to ensure that the code works according to the specifications. Code coverage is a metric that tries to measure the test quality by indicating the percentage of code that were used during test execution. This article aims to research the correlation between code coverage rate and software quality, measured by reported faults. To do that it was done a case study using the features of an operational system and the results indicate that there is a correlation between code coverage rate and software quality.*

**Resumo.** *O teste de software é uma importante etapa no processo de desenvolvimento de software. Essa etapa visa garantir que o software funciona de acordo com as especificações. A cobertura de código é uma métrica que se propõe a medir a qualidade de um teste. Ela consiste em prover uma taxa percentual que indica a porcentagem de código executadas nos testes. O presente artigo visa estudar se existe ou não correlação entre a cobertura de código e a qualidade do software, medida em quantidade de falhas reportadas. Para tal, foi feito um estudo de caso utilizando as funcionalidades de um sistema operacional, chegando em um resultado que indica a existência de correlação entre cobertura de código e qualidade de software.*

## 1. Introdução

A etapa de teste de *software* é uma etapa importante no processo de desenvolvimento de *software* e está cada vez mais presente no cotidiano das equipes de desenvolvimento de *software*. Essa etapa visa garantir a funcionalidade de um dado *software*, podendo ser feita utilizando diversos métodos e em diversos níveis, de acordo com a finalidade do teste. Contudo, o desenvolvimento de testes implica em um impacto na alocação de recursos do projeto, uma vez que muitas horas e profissionais são demandados para a escrita de testes.

A cobertura de código é uma métrica que indica a porcentagem de código que está sendo coberta pelos testes executados. Essa métrica ajuda a medir a qualidade dos casos de teste em questão e também pode servir como uma métrica que indica quando podemos parar de escrever testes, ou seja, quando já temos uma quantidade suficiente de testes para cobrir um determinado código, com a finalidade de redução de custos. Contudo, a decisão pela parada da escrita de testes pode impactar na qualidade do *software* final, uma vez que alguns casos de uso podem ficar descoberto pelos testes. Por esse motivo é imprescindível que tal decisão seja tomada baseada em dados sólidos, a fim de evitar esforços excessivos no desenvolvimento de teste, porém garantindo a qualidade do *software*. Segundo [Ivanković et al. 2019], a adoção da análise de cobertura de código

vem crescendo ao longo dos anos e tem uma boa avaliação de seus usuários quanto à sua efetividade, o que reafirma a necessidade de que se tenha um bom conhecimento sobre a efetividade de tal métrica para medir qualidade de teste e de *software*.

Diversos estudos tentam verificar se existe, de fato, uma correlação entre cobertura de testes e qualidade de *software*, tais como [Barani et al. 2023], [Chioteli et al. 2021] e [Kochhar et al. 2017]. Os resultados obtidos por esses estudos são controversos entre si. [Barani et al. 2023] traz uma revisão sistemática de diversos artigos publicados sobre o assunto e aponta algumas possíveis vulnerabilidades metodológicas nos artigos estudados, o que sugere que mais estudos podem ser feitos para tentar delinear melhor a relação entre cobertura de código e qualidade de *software*.

Nesse sentido, o presente artigo tem por objetivo responder a seguinte questão de pesquisa: existe correlação entre cobertura de código e qualidade de *software* de um sistema operacional embarcado, sendo cobertura de linhas a métrica de cobertura de código e número de falhas reportadas a métrica de qualidade de *software*? Para tal, o presente estudo tem o objetivo de fazer um estudo de caso, conforme descrito em [Azevedo et al. 2011], em um projeto de sistema operacional embarcado desenvolvido por uma empresa brasileira, observando as métricas supracitadas e usando métodos estatísticos para identificar a correlação entre cobertura de código e qualidade de *software*. Nesse contexto, as seguintes etapas serão desenvolvidas:

- Coletar dados de cobertura de código (por repositório) e falhas reportadas do projeto (por funcionalidade);
- Sanitizar dados, excluindo dados inválidos para a pesquisa;
- Relacionar os repositórios com as funcionalidades;
- Fazer análise estatística dos dados coletados.

Os resultados da presente pesquisa contribuem para uma maior clareza sobre a correlação entre cobertura de código e qualidade de *software*, o que pode ajudar a balizar decisões de alocação de recursos humanos na atividade de escrita de testes de *software*. Além disso, os resultados obtidos podem ser utilizados como base acadêmica na área de Engenharia de Software para um melhor entendimento dessa correlação, além de produzir insumos para estudos futuros nesse mesmo contexto.

O artigo está dividido seis sessões, são elas:

1. Introdução: tem por objetivo fornecer uma visão geral da presente pesquisa, além da motivação e etapas desenvolvidas;
2. Fundamentação Teórica e Ferramentas: apresenta um embasamento teórico dos assuntos abordados no artigo, além de listar as principais ferramentas utilizadas na coleta, triagem e análise dos dados;
3. Trabalhos Relacionados: revisita alguns trabalhos recentes relacionados ao tema da presente pesquisa;
4. Metodologia de Pesquisa: apresenta a metodologia utilizada no desenvolvimento do estudo proposto;
5. Resultados: detalha os resultados obtidos da análise dos dados;
6. Considerações Finais: faz um resumo dos resultados, vulnerabilidades da pesquisa, além de indicar possíveis estudos complementares.

Durante o desenvolvimento do presente trabalho foram criados e utilizados diversos *scripts* para organização, triagem e visualização dos dados. Tais *scripts*, bem

como os dados utilizados como insumo da pesquisa, podem ser encontrados na íntegra em <https://github.com/redivo/article-coverage-and-bugs/>. Os dados publicados nesse repositório foram ofuscados a fim de manter o sigilo da empresa fornecedora do caso a ser estudado.

## 2. Fundamentação Teórica e Ferramentas

Esta sessão contém o referencial teórico relativo aos principais conceitos envolvidos no presente estudo. Além disso, serão descritas as principais ferramentas utilizadas durante as fases coleta, sanitização, triagem e análise dos dados utilizados no estudo.

### 2.1. Testes de Software

A etapa de teste de *software* consiste em verificar que um *software* funciona de acordo com os requisitos, buscando por defeitos no *software* em teste [Sommerville 2011].

Neste artigo iremos utilizar as definições descritas em [Maldonado et al. 2016] para os termos de “defeito”, “erro” e “falha”. Segundo a definição do autor, o **defeito** é um “passo, processo ou definição de dados incorretos”. O **erro** é um estado inconsistente do programa, que foi causado por um **defeito**. Por fim, temos a **falha**, que se caracteriza pela produção de um resultado diferente do esperado. Sendo assim, podemos dizer que um teste de *software* visa por encontrar **defeitos** através a observação do comportamento de um programa e a busca de **falhas** de execução do mesmo.

Tais testes podem ser manuais ou automáticos. Segundo [Sommerville 2011], nos testes manuais o testador executa o programa manualmente, injetando alguns dados, e compara o resultado com o que ele espera que seja o resultado correto. Já nos testes automáticos, ou automatizados, os casos de teste são codificados em um programa responsável pela execução dos testes. Esse programa injeta os dados de entrada no *software* testado e verifica se os resultados estão de acordo com o esperado.

Dentro do contexto de teste de *software* podemos subdividir os testes em diversas categorias. Ao decorrer do presente trabalho, veremos duas categorias principais: testes funcionais e testes estruturais. Testes funcionais (ou de caixa-preta) caracterizam-se por testes onde não consideramos aspectos internos do programa a ser testado e é avaliado mais do ponto de vista do usuário [Maldonado et al. 2016]. Já testes estruturais (ou de caixa-branca) são testes baseados na implementação do programa, levando em conta laços de execução, condicionais, definições e usos de variáveis, entre outros [Maldonado et al. 2016].

Uma outra categorização de testes importante de ser mencionada é a categorização por granularidade. Em [Sommerville 2011] o autor divide os testes em 3 níveis de granularidade: (1) teste unitário, onde é testada a menor parte viável de um programa, (2) teste de componente, onde o teste integra diversas unidades de um programa e, por fim, (3) teste de sistema, onde o sistema é testado como um todo, englobando alguns ou todos os componente.

Neste estudo nos debruçaremos sobre uma parcela dos testes automáticos do caso em questão, pois são esses testes que são capazes de fornecer os dados que iremos utilizar. A sessão 4.2 trará mais detalhes sobre os testes utilizados neste estudo.

## 2.2. Cobertura de Código

O fato de um teste automático de *software* não encontrar defeitos não implica em o *software* não ter defeitos. O desenvolvedor do teste pode não ter implementado testes suficientes para todos os casos possíveis, o que pode fazer com que existam defeitos não encontrados pelos testes [Sommerville 2011].

Existem diversas formas de tentar medir a qualidade de um teste, isto é, o quanto bem este teste consegue cobrir o *software* que está sendo testado. Uma dessas formas é a cobertura de código.

Segundo a definição de [Aniche 2012], “cobertura de código é a métrica que nos diz a quantidade de código que está testada e a quantidade de código onde nenhum teste o exercita”. Essa “quantidade de código” pode ser visualizada de diversas formas, como cobertura de linhas, cobertura de decisão, ou *decision coverage*, em inglês, cobertura de métodos, entre outras. Neste estudo iremos utilizar como métrica a cobertura de linhas de código, visto que no projeto em estudo os dados de cobertura são reportados neste formato.

As métricas de cobertura de código podem ser extraídas utilizando diversas ferramentas, de acordo com a tecnologia adotada no projeto. O projeto do caso de estudo é implementado em sua maioria utilizando a linguagem C++, o Google Test [GTEST 2023] como *framework* de testes e a cobertura de código é calculada utilizando gcov [GCOV 2023].

Uma vez que uma alta taxa de cobertura de linhas código indica que o programa testado está, em grande parte, coberto por testes automáticos, então pode-se pensar que o critério de cobertura de código pode ser uma métrica de qualidade do *software* testado, e não apenas qualidade do teste. É nesse contexto que o presente estudo utilizará as métricas de cobertura de código para verificar se, de fato, existe uma correlação que confirme tal suposição.

## 2.3. Regressão Linear, Correlação e Valor-p

Ao decorrer da análise dos dados deste estudo será necessária uma análise estatística para responder a questão de pesquisa proposta. Para tal, faremos o uso de algumas ferramentas estatísticas, tais como regressão linear e coeficiente de correlação.

No presente estudo teremos como principais os dados de (1) quantidade de falhas reportadas e (2) taxa de cobertura de código. Para verificarmos se existe alguma tendência entre os valores podemos utilizar a **regressão linear**.

Segundo [Diez et al. 2019], modelos de regressão linear podem ser usados para previsões com base em dados de duas variáveis numéricas. A Figura 2.3 mostra um exemplo de regressão linear, onde a tendência é de que os valores do eixo  $y$  subam à medida que os valores do eixo  $x$  sobem.

Outra ferramenta que iremos utilizar neste artigo é o cálculo de coeficiente de correlação. Segundo [Diez et al. 2019], a correlação ( $R$ ) é uma medida com valor entre -1 e 1 que indica o quanto forte é a tendência linear entre as variáveis numéricas estudadas. Quanto mais próximo de -1 for  $R$ , maior é a força de correlação negativa. Quanto mais próximo de 1 for  $R$ , maior é a força de correlação positiva. Por fim, quanto mais próximo

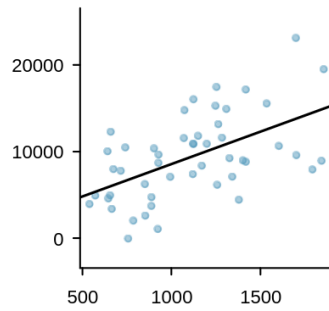


Figura 1. Exemplo de regressão linear. Fonte: [Diez et al. 2019]

de 0 for  $R$ , menor é a força de correlação entre os valores estudados. A Figura 2.3 mostra oito exemplos de coeficiente de correlação  $R$ .

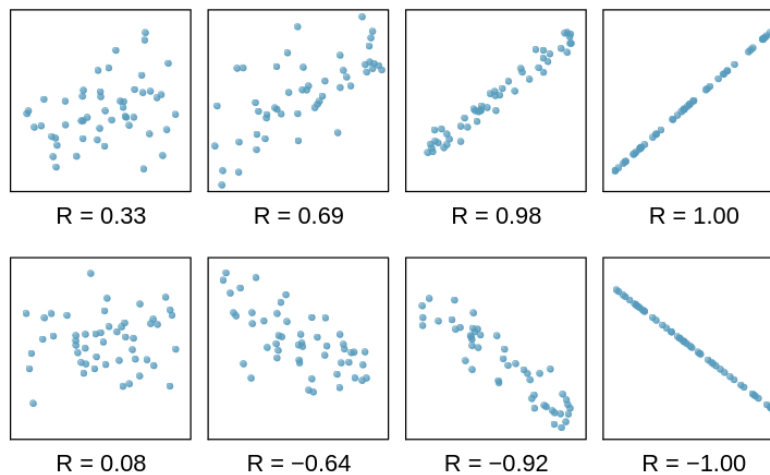


Figura 2. Exemplo coeficiente de correlação. Fonte: [Diez et al. 2019]

Um erro comum na compreensão do termo “correlação negativa” é interpretar que este termo indica a ausência da correlação. Contudo, tal termo exprime a ideia de uma correlação onde  $y$  diminui à medida que  $x$  aumenta. A Figura 2.3 exprime muito bem esse conceito, mostrando casos de correlação positiva (quando  $R = 1$ ), correlação negativa (quando  $R = -1$ ) e, por fim, uma baixa força de correlação (quando  $R = 0.08$ ).

Além disso, o presente estudo utiliza o cálculo do *valor-p* para estimar a probabilidade de que nossa hipótese de que existe correlação entre cobertura de código e número de falhas reportadas é verdadeira. Para explicar o conceito do *valor-p*, bem como seus complementares  $H_0$  e  $H_A$ , podemos recorrer novamente às definições de [Diez et al. 2019]. Inicialmente vamos definir a hipótese nula ( $H_0$ ) como a perspectiva cética, ou seja,  $H_0$  corresponde à hipótese de que não há correlação entre cobertura de código e número de falhas reportadas. Complementarmente podemos definir a hipótese alternativa ( $H_A$ ), ou seja,  $H_A$  corresponde à hipótese de que, sim, existe uma correlação entre cobertura de código e número de falhas reportadas. Já o *valor-p* pode ser interpretado como uma ferramenta para medição da força da evidência mostrada pelos dados, sendo que quanto menor o *valor-p*, maior a probabilidade de que  $H_0$  seja rejeitada e, por consequência,  $H_A$  seja provável a hipótese verdadeira. Como uma regra geral, o autor indica o valor de 5% como base, ou seja, caso o *valor-p* seja maior que 5%, não podemos descartar  $H_0$ .

Os valores de  $R$  e  $\text{valor-}p$  tem definições que podem dar a impressão de sobreposição de conceitos, porém tais valores indicam coisas diferentes. Enquanto o valor de  $R$  indica o quão ajustado à linha de tendência são os valor da amostragem, o  $\text{valor-}p$  indica a probabilidade de que  $H_0$  seja verdadeira.

Com essas ferramentas estatísticas podemos verificar se existe alguma tendência entre os dados estudados e, caso exista, qual a força da correlação e probabilidade de que nossa hipótese seja verdadeira. É importante salientar que as tendências e correlações calculadas não implicam em causalidade, apenas nos dão insumos para uma análise mais criteriosa, além de nos darem indícios que sugerem tal causalidade, porém devem ser confirmadas em estudos futuros.

## 2.4. Ferramentas

Diversas ferramentas foram utilizadas durante a obtenção, triagem e análise dos dados estudados neste artigo.

O Bugzilla [BUGZILLA 2023] é um sistema de gerenciamento de defeitos que possibilita a organização, categorização e discussão de defeitos reportados. O projeto do qual os dados foram coletados utiliza o Bugzilla como sistema de gerenciamento de defeitos. Nele existe um campo chamado “*Component*” que é utilizado para indicar a qual funcionalidade, do ponto de vista do usuário, aquele defeito é relativo, que será utilizado para categorização dos defeitos neste presente estudo.

Os resultados de cobertura de código, no projeto estudado, são calculados pelo gcov [GCOV 2023], que se trata de uma ferramenta para geração de dados de cobertura de código para ser utilizada em conjunto com o compilador GCC [GCC 2023].

No projeto estudado, os testes, incluindo geração de dados de cobertura de código, são executados automaticamente pelo Jenkins [JENKINS 2023] (ferramenta para automatizações de integração contínua) toda a vez que um novo *commit* é submetido para revisão ou para o *branch* principal. Além dos resultados dos testes (sucesso ou falha) também são publicizados os dados de cobertura de código calculados pelo gcov.

Na parte de triagem e análise dos dados, alguns *scripts* em Python foram criados e estão descritos em <https://github.com/redivo/article-coverage-and-bugs/tree/master/data>. Nestes *scripts* foram utilizadas as bibliotecas (1) Plotly [PLOTLY 2023], uma biblioteca Python para geração de gráficos e (2) SciPy [SCIPY 2023], uma biblioteca Python para auxílio de cálculos matemáticos e estatísticos.

## 3. Trabalhos Relacionados

Em [Chioteli et al. 2021] os autores propõem um estudo sobre a relação entre cobertura de código e a ocorrência de falhas, tentando traçar uma correlação entre os dois valores. Como estudo de caso foram usados reportes de falhas em campo do projeto Eclipse. O estudo coletou mais de 2 milhões de incidentes reportados e, após triagem de dados os autores tinham em mãos mais de 126 mil reportes únicos do tipo *stack trace*, que consiste em um tipo de reporte de erro que contém informações como arquivo e linha de onde ocorreu a falha.

Em paralelo foram coletados dados da análise de cobertura de código do projeto



em questão. Como principal métrica de cobertura de código para a comparação foi utilizado a métrica de métodos cobertos, que consiste em relacionar a quantidade de métodos existentes e quantos desses são invocados nos testes.

Foi feita uma análise dos dados relacionando as falhas reportadas e a cobertura de código e a Tabela 1 mostra os dados obtidos. Do total de métodos analisados, 93,6% não são testados e apenas 6,4% dos métodos são testados. Analisando apenas os métodos que não possuem reportes de falhas relacionados essa proporção é de 93,5% de métodos não testados, para 6,5% de métodos testados. Já quando analisamos métodos que possuem reportes de falhas relacionados essa proporção fica em 94,1% de métodos não testados, para 5,9% de métodos testados.

**Tabela 1. Falhas de Métodos. Fonte: [Chioteli et al. 2021]**

<b>Falha Reportada?</b>	<b>Possui teste?</b>		<b>Total</b>
	<b>Não</b>	<b>Sim</b>	
Não	7816	541	8357
Sim	1097	69	1166
Total	8913	619	9523

Com esse resultado o estudo conclui que não é possível afirmar que existe uma correlação entre cobertura de código e uma menor quantidade de ocorrência de falhas. Os valores encontrados variam marginalmente, apresentando um baixo valor de correlação.

Complementarmente, o artigo traz alguns itens a serem levados em conta. Um deles é que o resultado foi obtido utilizando apenas um projeto e tal projeto tem grandes proporções. Os autores argumentam que o resultado não pode ser generalizado e pode vir a se diferente em outros tamanhos de projeto.

O artigo [Barani et al. 2023] traz uma revisão sistemática de outros trabalhos que analisam uma possível correlação entre cobertura de código e efetividade de um conjunto de testes. Para tal, os autores fizeram uma busca utilizando a ferramenta de busca Scopus para a seleção dos trabalhos a serem estudados a fim de responder três questões de pesquisa:

1. Quais evidências existem sobre a correlação entre cobertura de código e efetividade dos testes?
2. Quais fatores impactam nessa relação?
3. Qual métrica de cobertura de código é melhor para prever a efetividade dos testes?

O estudo descreve os termos de busca utilizados, além de duas etapas de filtragem de estudos que foram necessárias para obter um material mais conciso para a revisão sistemática. Após a primeira busca eles encontraram 427 resultados para a busca e, após os filtros restaram 33 estudos para serem analisados.

Segundo os autores, alguns estudos prévios apontavam que existe uma forte correlação entre cobertura de código e efetividade dos testes. Por outro lado alguns estudos indicavam que essa correlação era baixa ou até nula.

Como resultado desse artigo os autores argumentam que uma possível causa da não convergência dos resultados dos estudos seja a falta de padronização entre os estudos,

uma vez que os diversos estudos revisados possuem abordagens diferentes de análise, além de casos de uso bem distintos entre eles. Para obter melhores resultados os autores recomendam atenção em alguns pontos que podem prejudicar os resultados de estudos, tais como descartar projetos muito pequenos ou com uma cobertura excessivamente baixa. Outro ponto levantado pelos autores é uma lista de fatores que podem interferir na relação entre cobertura de código e efetividade dos testes, como porcentagem de cobertura, testes que exercitem tanto casos normais, quanto casos de exceção, tamanho do conjunto de testes, natureza das falhas, complexidade do código, entre outros.

## **4. Metodologia de Pesquisa**

Essa pesquisa utilizou o método de estudo de caso e analisou os resultados por uma perspectiva quantitativa.

Segundo [Yin 2001], “você poderia utilizar o método de estudo de caso quando deliberadamente quisesse lidar com condições contextuais – acreditando que elas poderiam ser altamente pertinentes ao seu fenômeno de estudo”. Nesse sentido o presente estudo utilizará como caso um sistema operacional embarcado em operação desde 2015 em conjunto com suas métricas de cobertura de código e quantidade falhas encontradas.

Com base nos dados coletados, será feita uma análise quantitativa que, segundo a definição de [Creswell 2007], utiliza raciocínio de causa e efeito, observação de dados, teste de teorias, coleta de dados, entre outros instrumentos.

### **4.1. Delineamento de Pesquisa**

A presente pesquisa é de caráter quantitativo, visto que se pretende traçar uma correlação entre duas variáveis numéricas quantitativas – percentual de cobertura de código e quantidade de falhas reportadas. Para descobrir se existe tal relação, os números encontrados serão submetidos a análises estatísticas.

Para a coleta de dados, será feita uma pesquisa exploratória em um estudo de caso de um projeto de sistema operacional. Esse modelo de pesquisa é amplamente utilizado ([Barani et al. 2023], [Chioteli et al. 2021] e [Kochhar et al. 2017]) em estudos do tipo, uma vez que a coleta de dados é precisa e de fácil acesso, além de trazer uma representação muito próxima à realidade do caso estudado.

Apesar de a pesquisa ter um caráter quantitativo em seu objetivo principal, a utilização de métodos qualitativos foram necessários em etapas intermediárias da pesquisa. Durante a triagem dos dados obtidos, mais especificamente da classificação dos repositórios, foi necessária a utilização da técnica de entrevistas para viabilizar tal classificação, uma vez que esta não estava documentada. Além disso, os resultados obtidos foram apresentados amplamente para o setor de pesquisa e desenvolvimento da empresa fornecedora do caso de estudo, incluindo times de outros projetos que não o projeto estudado, a fim de que fosse feita uma revisão por pares que, segundo [Azevedo et al. 2011], são outros pesquisadores que podem criticar as interpretações e conclusões apresentadas. Para coleta do parecer dos pares foi aberto um canal de comunicação via o aplicativo Microsoft Teams, que recebeu alguns questionamentos e ponderações que serão apresentadas na sessão 5.

## 4.2. Unidade de Análise

A pesquisa ocorreu em uma empresa brasileira que projeta e desenvolve *hardware* e *software*. O enfoque do presente estudo será em um projeto específico de sistema operacional, abrangendo todas as funcionalidades desenvolvidas pela empresa dentro deste sistema operacional.

O sistema conta com 85 funcionalidades de usuário cadastradas e divididas em 316 repositórios *Git* ativos. Uma funcionalidade de usuário, no contexto do presente artigo, é uma característica ou função exercida pelo sistema operacional que seja percebida pelo usuário. Desta forma, funções internas ao sistema operacional que não geram funcionalidade do ponto de vista do usuário não são consideradas funcionalidade de usuário. Cada funcionalidade pode ser implementada por 1 ou mais repositórios e cada repositório pode atender 0 ou mais funcionalidades de usuário. Um exemplo dessa representação pode ser visto na Figura 4.2.

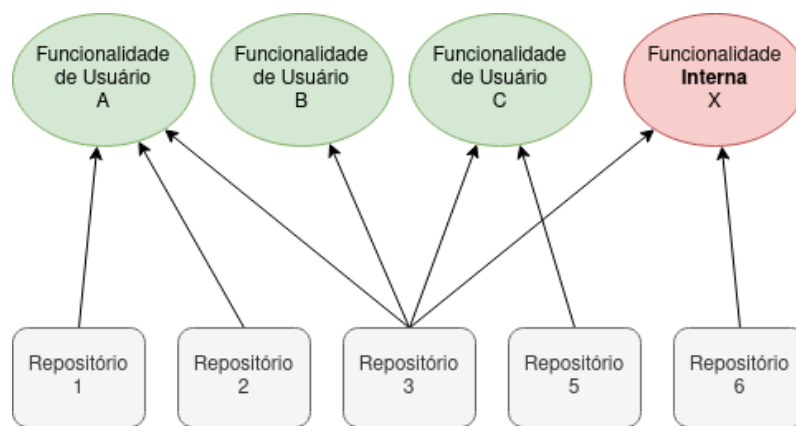


Figura 3. Exemplo de relação entre repositórios de funcionalidades

No contexto do projeto estudado, os testes de *software* são divididos em duas categorias, de acordo com o ambiente onde eles são executados.

A primeira categoria são os testes executados em ambientes de máquinas virtuais, ou *Virtual Machines* (VMs), em servidores especializados. Essa categoria executa testes funcionais e estruturais de granularidade unitária e de componente, além de calcular os dados de cobertura de código. Esses testes são executados em dois momentos: (1) sempre que um dado código é submetido para revisão, disparado pela submissão do código e (2) diariamente, disparado por uma tarefa agendada do Jenkins.

A segunda categoria é a de testes executado no equipamento destino. Nessa etapa existem vários equipamentos que terão seu sistema operacional atualizado com a versão sobre teste e será executada uma bateria de testes a fim de testar as funcionalidades utilizando os equipamentos reais e não mais máquinas virtuais. Essa categoria de testes executa testes funcionais de granularidade de sistema, contendo o sistema operacional com todos os componentes.

A Figura 4.2 mostra um resumo das etapas do desenvolvimento de software do projeto em questão, enfatizando os momentos onde os testes ocorrem. Analisando a figura, podemos perceber que os testes executados em VMs e disparados pela submissão para revisão executam com base no código em revisão, já os testes periódicos disparados

por tarefa agendada o Jenkins executam com base no topo do repositório.

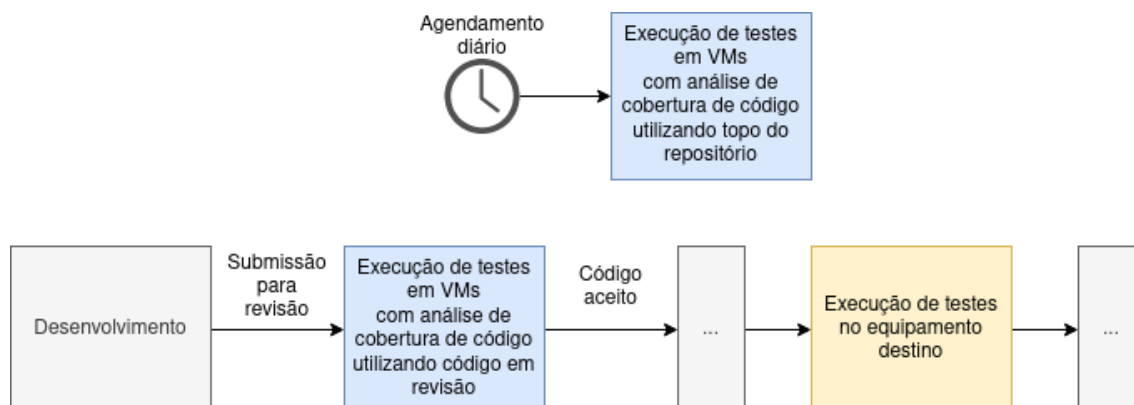


Figura 4. Pipeline resumido da integração com enfoque na execução dos testes

Outro ponto importante a salientar é o contexto dos dados de cobertura de código. Os dados gerados são específicos por repositório, ou seja, temos dados de cobertura de código dos 316 repositórios separadamente.

Uma vez que os testes em ambiente de máquinas virtuais são os testes que produzem dados de cobertura de código, são esses testes que serão utilizados durante o presente estudo, deixando de fora os testes executados nos equipamentos destino. Os dados utilizados no estudo compreendem aos dados obtidos do topo dos repositórios, através da execução periódica disparada pelo Jenkins.

Sobre a linguagem de programação adotada pelo projeto, a esmagadora maioria do código é escrito nas linguagens C e C++. A coleta de dados de cobertura de código foi feita em todos os repositórios que reportam cobertura de códigos C e C++, sendo descartados códigos auxiliares escritos em outras linguagens, como Python e Shell.

A coleta de dados de falhas reportadas compreende a todas as falhas relacionadas com o projeto estudado e que não tenham sido corrigidas até o momento da pesquisa. Isso inclui tanto falhas que ocorreram em campo, quanto falhas identificadas dentro da empresa, seja por testes em equipamentos destino, testes de aceitação manual ou uso interno. A decisão de considerar apenas as falhas que não foram corrigidas se deu devido ao fato de que ao considerar o conjunto de todas as falhas, estaríamos tendo interferência do histórico, o que poderia interferir o resultado do estudo, uma vez que os dados de cobertura de código são relativos ao momento atual. E, por fim, a decisão de contabilizar todas as falhas, e não apenas as de campo, se deu devido à compreensão de que os defeitos encontrados internamente não são menos relevantes, ou menos válidos para o presente estudo, do que defeitos encontrados em campo.

#### 4.3. Coleta de Dados

O presente estudo se baseia em dados de duas naturezas distintas: (1) dados de falhas reportadas e (2) dados de cobertura de código. Ambos os dados foram coletados no dia 28/10/2023, representando o estado atual do projeto, tanto do ponto de vista das falhas reportadas, quanto do ponto de vista da cobertura de código. Uma vez que os dados coletados representam o estado do projeto na data de coleta, estes não podem ser analisados sob

uma óptica de evolução temporal do projeto, mas sim sob uma perspectiva de “fotografia” do projeto na data de coleta.

Apesar de ambos os dados terem sido coletados na mesma data, tais coletas foram feitas de formas distintas, as quais serão apresentas nas sessões 4.3.1 e 4.3.2.

#### 4.3.1. Falhas Reportadas

As falhas reportadas compreendem à quantidade de todas as falhas encontradas tanto externamente (por clientes) quando internamente (por colaboradores). O projeto estudado utiliza a ferramenta Bugzilla como ferramenta gerenciadora de falhas tanto para falhas descobertas internamente, quanto para falhas descobertas externamente, o que nos permite acesso à toda a base de dados de falhas do projeto. Contudo, o mesmo serviço do Bugzilla é utilizado na empresa para reportar falhas de outros projetos e esse detalhe precisou ser levado em consideração no momento da formulação da busca.

O Bugzilla possui uma ferramenta de busca onde é possível listar bugs de acordo com um determinado filtro de busca e a saída dessa busca pode ser de dois tipos: (1) HTML, onde é exibida uma interface web contendo os resultados ou (2) um arquivo do tipo CSV contendo uma linha para cada resultado encontrado, sendo as colunas configuráveis via requisição HTTP. O formato escolhido foi o de arquivo CSV pois facilita a automação da triagem de dados.

A busca criada para selecionar a lista de falhas reportadas relacionadas ao projeto e que ainda não foram solucionadas utilizou filtragem por dois campos do Bugzilla:

- **Status:** Compreende ao estado atual do bug, que pode assumir diversos valores e se propõe a informar em qual fase de sua correção ele está, compreendendo desde as fases mais iniciais, como *UNCONFIRMED*, que informa que não foi feita uma análise inicial, até as fases finais, como *CLOSED*, onde o problema já foi resolvido e a solução foi aceita pelas equipes responsáveis pela aceitação. O termo escolhido para o filtro desse campo foi o de “**status!=(RESOLVED OR CLOSED)**”. Ambos *RESOLVED* e *CLOSED* representam falhas reportadas que já tiveram sua correção integrada ao sistema operacional, porém *RESOLVED* é um passo anterior, onde os desenvolvedores já fizeram as implementações, testes e integração, porém ainda não foi dado o aceite da correção, enquanto *CLOSED* é o estado onde o aceite foi concedido. Essa escolha foi feita pois no fluxo de trabalho definido na empresa estudada as falhas são marcadas como *CLOSED* apenas no fechamento de uma dada versão, sendo que elas podem já ter sido corrigidas, logo considerar ambos os termos nos traz uma fidelidade maior sobre os dados do projeto;
- **Project:** Compreende ao projeto em que foi encontrado esse problema. Existe um valor de *Project* específico para o projeto estudado, o qual vai ser utilizado na busca. Sendo assim, temos que o termo da busca de projeto é “**project==PROJETO**”, sendo PROJETO o nome que representa o projeto de sistema operacional estudado, mas não será divulgado por questões de sigilo empresarial.

Unindo os dois termos de busca, um que seleciona o estado da falha e o outro que seleciona o projeto relativo à falha, temos o seguinte termo de busca: “**(status!=(RESOLVED OR CLOSED)) AND (project==PROJETO)**”. Tal termo de busca

tem por objetivo selecionar todas as falhas reportadas que ainda não tem solução e que foram encontradas no projeto estudado.

Como resultado, o Bugzilla gera arquivo no formato CSV contendo 2 colunas: (1) campo *bugId*, que se trata de um número identificador do reporte de falha e (2) campo *component*, que informa em qual funcionalidade de usuário a falha foi encontrada. O primeiro não terá utilidade no presente estudo, mas é um campo obrigatório nos reportes do Bugzilla, já o segundo será utilizado para relacionar a quantidade de falhas reportadas com os índices de cobertura de código.

Além do reporte de falhas reportadas no Bugzilla foi necessário obter a lista de todos os valores possíveis do campo *component*, o qual representa o conjunto de funcionalidade de usuário disponíveis no sistema. Essa lista é necessária para conhecermos todas as funcionalidades de usuário, uma vez que a funcionalidade pode não ter nenhuma falha reportada e, dessa forma, não apareceria no reporte de falhas reportadas e ainda não corrigidas. Essa lista foi utilizada para a estrita de um *script*, o qual será descrito na sessão 4.4.

Com os dados obtidos é possível fazer a contagem de falhas reportadas por funcionalidade de cliente e tal triagem dos dados será detalhada na sessão 4.4. Os dados resultantes da coleta de falhas reportadas não podem ser exibidos na íntegra por questões de sigilo empresarial, porém foram compilados e anonimizados e serão exibidos na sessão 5.

#### 4.3.2. Cobertura de Código

A cobertura de código compreende à uma medida que indica a relação de quantidade de código testado e código não testado. O projeto estudado calcula tais dados utilizando a ferramenta gcov e executando os testes escritos com auxílio do *framework* Google Test, utilizado para desenvolvimento de testes de código utilizando as linguagens C e C++.

Os dados de cobertura de código estavam inicialmente sendo disponibilizados através do *plugin* Cobertura [COBERTURA 2023], disponível do Jenkins (Figura 4.3.2), e as métricas de cobertura exportadas para o projeto são:

- **Cobertura de Linhas:** Corresponde à quantidade de linhas de código testadas em relação ao total de linhas de código válidas do repositório;
- **Cobertura de Arquivos:** Corresponde ao total de arquivos contendo código executado nos testes em relação ao total de arquivos C e C++ do repositório;
- **Cobertura de Classes:** Corresponde ao total de classes acessadas nos testes em relação ao total de classes C e C++ do repositório;
- **Cobertura de Métodos:** Corresponde ao total de métodos chamados na execução dos testes em relação ao total de métodos C e C++ contidos no repositório. Um detalhe sobre este campo é que funções em códigos não orientados a objetos também são contabilizadas como métodos no contexto desse reporte.

Dentre todas métricas disponíveis, a métrica de Cobertura de Linhas é a mais adequada. Segundo [Barani et al. 2023], métricas mais genéricas, como cobertura de arquivos ou cobertura de classes, são menos efetivas para relacionar a qualidade de código, ainda que tenham validade. No artigo os autores recomendam a utilização da métrica

Project Coverage summary




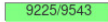
Name	Files	Classes	Methods	Lines
Cobertura Coverage Report	96%  22/23	96%  22/23	99%  1351/1370	97%  9225/9543

Figura 5. Exemplo de exibição do *plugin* Cobertura

de Cobertura de Decisão, porém tal métrica não está disponível atualmente no projeto e não pode ser utilizada. Em vista disso, a métrica considerada mais adequada foi a de Cobertura de Linhas.

O *plugin* Cobertura exibe dados utilizando uma interface web, o que dificulta a coleta de dados, uma vez que seria necessário coletar manualmente os dados de cobertura de código dos 316 repositórios. Para facilitar a coleta, o *script* de geração de dados para o *plugin* Cobertura foi alterado a fim de gerar dados no formato de arquivos JSON, além do formato original, destinado ao *plugin* Cobertura. Como resultado, o *script* passou a gerar uma série de arquivos JSON que foram compilados em apenas 1 arquivo JSON contendo uma lista de repositórios e seus respectivos dados de cobertura de código, conforme mostra a Figura 4.3.2. Tal *script* não pode ser exibido no presente artigo devido a questões relativas a sigilo empresarial.

```
{
  "Nome do Repositório": {
    "lines-valid": <número de linhas válidas>,
    "line-rate": <taxa de cobertura>,
    "lines-covered": <número de linhas cobertas>
  },
  ...
}
```

Figura 6. Exemplo de formatação do arquivo JSON gerado após a compilação dos dados de cobertura de código.

Dessa forma teremos como resultado um arquivo JSON contendo todos os 316 repositórios e, para cada um deles, seus respectivos dados de linhas válidas, linhas cobertas e taxa de cobertura. Os dados de linhas válidas correspondem ao total de linhas de código C e C++, excluindo linhas que não possuem execução, como linhas em branco, linhas de comentário e linhas de declarações de métodos, por exemplo. Número de linhas cobertas corresponde ao total de linhas válidas que foram executadas em pelo menos 1 teste. Por fim, a taxa de cobertura corresponde à divisão das linhas cobertas pelas linhas válidas, conforme a Figura 4.3.2.

$$taxaDeCobertura = \frac{linhasCobertas}{linhasValidas}$$

Figura 7. Fórmula de cálculo de taxa de cobertura.

Uma vez que os dados de cobertura de código foram exportados via JSON, eles passaram a estar disponíveis através do Jenkins. Dessa forma é foi possível coletar os dados de cobertura por repositório através de *download* dos arquivos. Cada repositório exporta seu dados diariamente, conforme descrito na sessão 4.2, e esses dados foram

coletados através de requisições **wget** em um terminal Linux com acesso ao Jenkins, sendo feita 1 requisição para cada um dos 316 repositórios. Ao final, os dados foram organizados, de acordo com o descrito na sessão 4.4.

#### 4.4. Triagem e Análise de dados

Antes de os dados serem analisados foi necessário um trabalho de triagem de dados, em especial da porção de dados que compreende aos dados de cobertura de código, a fim de organizá-los e classificá-los. Após a coleta, temos como insumo três dados distintos: (1) um arquivo contendo dados de falhas reportadas, (2) uma lista de funcionalidades de usuários cadastradas no Bugzilla e (3) um arquivo contendo os dados de cobertura de código por repositório Git.

Os dados de falhas reportadas foram exportados em um arquivo CSV contendo a primeira linhas como título das colunas e as demais sendo uma linha para cada falha reportada, com a filtragem de acordo com o descrito na sessão 4.3.1. Em conjunto com a lista de funcionalidades de usuário foi criado um *script* com o intuito de contabilizar a quantidade de falhas reportadas em cada funcionalidade de usuário. O *script* produziu um arquivo JSON contendo uma lista de funcionalidades de usuário e, para cada uma delas, a quantidade de falhas reportadas, contabilizadas através do arquivo CSV gerado pelo reporte do Bugzilla. Caso alguma funcionalidade da lista de funcionalidades de usuário não tenha nenhuma falha associada no arquivo CSV o valor de falhas reportadas para tal funcionalidade é considerado 0.

A Figura 4.4 mostra um exemplo da formatação do arquivo JSON gerado pelo *script* de contagem de falhas reportadas. Na estrutura, o campo **isComponent** indica se o elemento da lista é um *component*, ou seja, se o elemento da lista é uma funcionalidade de usuário. A necessidade desse campo será discutida mais adiante na presente sessão. O outro campo presente na estrutura gerada é o campo **bugs**, o que indica a quantidade de *bugs* relativos àquele *component*, isto é, a quantidade de falhas reportadas relativas àquela funcionalidade de usuário. Como resultado dessa primeira triagem obtivemos um arquivo no formato JSON contendo uma lista de todas as funcionalidades de usuário e, dentro de cada elemento dessa lista, a informação de quantas falhas reportadas estão relacionadas à dada funcionalidade.

```
{
  "Nome da funcionalidade" : {
    "isComponent": true,
    "bugs": <number>
  },
  ...
}
```

Figura 8. Exemplo de formatação do arquivo JSON gerado após alteração do *script* de contagem de falhas reportadas.

Nesse ponto, temos de um lado a lista de funcionalidades e seus respectivos números de falhas reportadas e de outro lado temos a lista de repositórios e seus respectivos dados de cobertura de código. O próximo passo foi relacionar os repositórios e as funcionalidades. Para tal, foram feitas entrevistas com os líderes técnicos de todas as equipes. A técnica de entrevista consiste em uma coleta de dados onde o entrevistador toma o entrevistado como fonte de informação [Gil 2008]. Uma vez que as informações



de relacionamento entre repositórios e funcionalidade não estão documentadas e em certos casos podem ser difusas, isto é, um repositório pode servir a mais de uma funcionalidade de usuário, a entrevista com os líderes técnicos foi uma solução que atendeu as necessidades da presente pesquisa.

O modelo de entrevista escolhido foi o modelo de entrevista focalizada que, de acordo com a definição de [Gil 2008], é uma entrevista pouco estruturada, onde o entrevistador foca em um tema, no caso do estudo foi a relação entre repositórios e funcionalidades de usuário, e permite que o entrevistado fale livremente, porém tentando manter o foco no tema definido. As entrevistas foram feitas individualmente com 5 líderes técnicos, cobrindo todos os repositórios. Para cada um deles a entrevista começou com o seguinte questionamento:

*Dado esses repositórios, a qual funcionalidade de usuários cada um deles pertence?*

Como resultado das entrevistas obtivemos os dados de funcionalidades de usuário e os dados de cobertura de código unidos em um único arquivo em formato JSON, conforme o exemplo exibido na Figura 4.4. Essa união foi feita manualmente durante as entrevistas. Os campos **isComponent**, **bugs**, **lines-valid**, **line-rate**, **lines-covered** continuam com o mesmo significado dos passos anteriores, porém aqui podemos ver um novo campo: **reposCoverage**. O campo **reposCoverage** é um elemento que contém uma lista com todos os repositórios referentes à funcionalidade no qual este elemento está inserido, bem como os dados de cobertura de código destes repositórios. Além disso, foram criados outros agrupamentos de repositórios que não fazem referência a uma funcionalidade específica e, por esse motivo, foram marcadas como **false** no campo **isComponent**. Essa marcação facilita a compilação de dados feita por alguns *scripts* que serão detalhados adiante nesta mesma sessão.

Após a integração dos dados, tivemos repositórios sendo classificados de 3 formas diferentes:

- **Repositórios que não estão em campo:** são repositórios que não foram entregues no sistema operacional, ou foram obsoletados em algum momento. Tais repositórios foram agrupados em uma divisão chamada *Not In Field*, que reúne os repositórios que ainda não estão ou não estão mais sendo utilizados e, por esse motivo, não possuem falhas reportadas relativas a eles. Esses repositórios foram desconsiderados nas análises estatísticas;
- **Repositórios de sistema:** são repositórios que servem a mais de uma funcionalidade simultaneamente. Estes foram agrupados em uma divisão chamada *System repos*. Uma vez que esses repositórios não foram relacionados a uma funcionalidade específica, eles foram descartados de grande parte das análises estatísticas;
- **Repositórios de funcionalidade:** são repositórios que servem a apenas uma funcionalidade de usuário, sendo colocados dentro do elemento referente a sua funcionalidade.

O último passo referente ao tratamento inicial dos dados, antes da análise dos mesmo, foi a ofuscação de dados. Os dados originais contém os nomes reais das funcionalidades e repositórios e eles foram ofuscados por questões de sigilo empresarias. O *script* responsável pela ofuscação de dados está disponível em <https://github.com/redivo/article-coverage-and-bugs/blob/>

```

{
  "Nome da funcionalidade" : {
    "isComponent": true,
    "bugs": <number>,
    "reposCoverage": {
      "nome-do-repositório-x": {
        "lines-valid": <number>,
        "line-rate": <number>,
        "lines-covered": <number>
      },
      ...
    }
  },
  "Outras divisões": {
    "isComponent": false,
    "reposCoverage": {
      "nome-do-repositório-y": {
        "lines-valid": <number>,
        "line-rate": <number>,
        "lines-covered": <number>
      },
      ...
    }
  },
  ...
}

```

**Figura 9.** Exemplo de formatação do arquivo JSON resultante da junção manual dos dados de falhas reportadas por funcionalidade e dados de cobertura de código por repositório.

master/data/obfuscate-data.py e ele é responsável por receber como entrada o arquivo JSON com nomes originais e gerar um novo arquivo JSON com nomes genéricos, seguindo o seguinte padrão: **Component\_<número>** e **Repository\_<número>**. Além das questões de sigilo empresarial, a ofuscação dos dados permite-nos uma análise de dados excluindo vieses, uma vez que não é possível distinguir funcionalidades e repositórios.

O passo seguinte à triagem de dados foi a análise dos mesmos. Tendo como base um arquivo JSON contendo todos os dados de falhas reportadas divididas por funcionalidade e seus respectivos repositórios contendo dados de cobertura de código, foi possível analisar os dados e gerar os resultados estatísticos. O arquivo na íntegra pode ser acessado através de [https://github.com/redivo/article-coverage-and-bugs/blob/master/data/obfuscated\\_data.json](https://github.com/redivo/article-coverage-and-bugs/blob/master/data/obfuscated_data.json).

Como objetivo principal deste estudo temos a seguinte questão de pesquisa: “Existe correlação entre cobertura de código e qualidade de *software* de um sistema operacional embarcado, sendo cobertura de linhas a métrica de cobertura de código e número de falhar reportadas a métrica de qualidade de *software*?” e para respondê-la podemos recorrer a cálculos estatísticos. Nesse sentido, o primeiro passo foi gerar um gráfico de bolhas (veja exemplo na Figura 10) com o eixo *x* sendo a taxa de cobertura de código, o eixo *y* a quantidade de falhas reportadas e cada bolha sendo uma funcionalidade, onde o tamanho da bolha indica a quantidade de linhas de código, representando o tamanho, em linhas da funcionalidade. Esse tipo de gráfico permite termos uma visualização gráfica inicial dos dados.

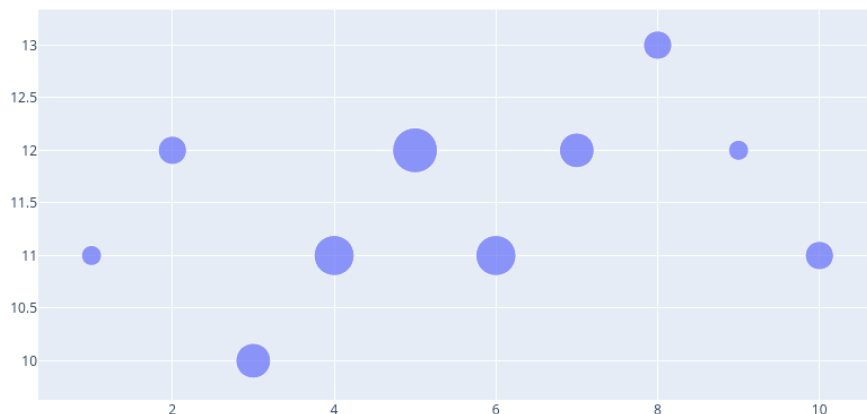


Figura 10. Exemplo de gráfico de bolhas. Fonte: [PLOTLY 2023].

De forma complementar, foi utilizado uma linha de regressão linear adicional ao gráfico de bolhas (veja exemplo na Figura 11). Segundo [Diez et al. 2019], modelos de regressão linear podem ser usados para previsões com base em dados de duas variáveis numéricas. No caso estudado as variáveis são taxa de cobertura de código e número de falhas reportadas. Para a geração do gráfico contendo as bolhas e a linha de regressão linear foi utilizado a biblioteca Plotly em sua versão para a linguagem Python [PLOTLY 2023]. Através dessa biblioteca é possível gerar os gráficos de forma fácil utilizando o método de Mínimos Quadrados Ordinários, ou *Ordinary Least Squares* (OLS), em inglês.

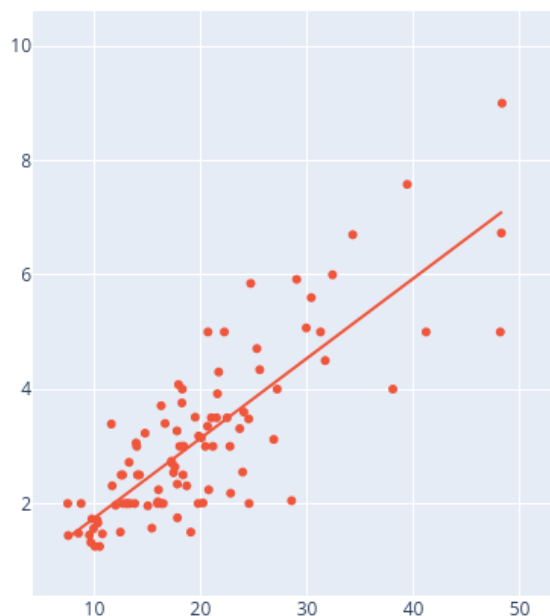


Figura 11. Exemplo de gráfico de bolhas com linha de regressão linear. Fonte: [PLOTLY 2023].

A geração dos gráficos foi feita através do *script* descrito em [https://github.com/redivo/article-coverage-and-bugs/blob/master/data/digest\\_data.py](https://github.com/redivo/article-coverage-and-bugs/blob/master/data/digest_data.py). Note que, como exibido na Figura 4.4, o JSON possui os dados de cobertura de código por repositório, porém precisamos dos dados por funcionalidade. Para resolver essa questão, o *script* gerador do gráfico também faz um

pré-processamento dos dados, somando os dados de cobertura de código de todos os repositórios de uma mesma funcionalidade (veja fórmula na Figura 4.4), obtendo assim os dados de cobertura de código de uma funcionalidade.

$$taxaDeCoberturaFuncionalidade = \frac{\sum_{i=a}^N linhasCobertas_i}{\sum_{i=a}^N linhasValidas_i}, \text{ sendo } a \text{ e } N \text{ o primeiro e último repositório referente à funcionalidade, respectivamente}$$

**Figura 12. Fórmula de cálculo de taxa de cobertura de uma funcionalidade de usuário.**

Com o gráfico de bolhas e regressão linear já é possível concluir se existe ou não correlação entre as variáveis estudadas, porém ainda não é possível quantificar a força dessa correlação. Para tal o *script* utiliza a biblioteca ScyPy [SCIPY 2023] para fazer o cálculo de correlação  $R$ , descrito na Figura 4.4, e *valor-p*, descrito na Figura 4.4. Com o valor resultante dos cálculos é possível mensurar a força e a validade da correlação encontrada [Diez et al. 2019].

$$R = \frac{\sum (x-m_x)(y-m_y)}{\sqrt{\sum (x-m_x)^2 \sum (y-m_y)^2}}$$

**Figura 13. Fórmula da correlação de Pearson. Fonte: [SCIPY 2023]**

$$valor-p = \frac{(1-r^2)^{n/2-2}}{B(\frac{1}{2}, \frac{n}{2}-1)}, \text{ sendo } n \text{ o número amostral e } B \text{ a função beta}$$

**Figura 14. Fórmula da correlação valor-p. Fonte: [SCIPY 2023]**

Por fim, alguns cálculos estatísticos auxiliares foram utilizados em algumas análises minoritárias, como o cálculos de média aritmética e mediana, a fim de capturar tendências de valores para grupos de repositórios.

## 5. Resultados

Esta sessão tem como objetivo apresentar os resultados obtidos no estudo.

Todos os resultados descritos nesta sessão se baseiam no arquivo contido em [https://github.com/redivo/article-coverage-and-bugs/blob/master/data/obfuscated\\_data.json](https://github.com/redivo/article-coverage-and-bugs/blob/master/data/obfuscated_data.json), que consiste em um arquivo no formato JSON contendo todos os números de falhas reportadas e cobertura de código, bem como os relacionamentos entre repositórios e funcionalidades de usuário, conforme descrito na sessão 4.4.

Após a triagem dos dados coletados, podemos ver a lista de falhas reportadas por funcionalidade conforme descrito na tabela constante no Apêndice A.

Observando a tabela é possível perceber que as funcionalidade possuem um número variado de falhas reportadas e repositórios relacionados. Dado que o objetivo do presente estudo é traçar uma correlação entre cobertura de código de número de falhas reportadas, as funcionalidades que não possuem nenhum repositório relacionado foram descartadas, uma vez que são os repositórios que possuem os dados de cobertura de código.

Os casos de funcionalidades sem repositórios relacionados ocorrem quando a funcionalidade é implementada utilizando apenas repositórios de sistema, ou seja, apenas

repositórios quem atendem a mais de uma funcionalidade. Dessa forma não é possível atribuir a cobertura de código à funcionalidade, seguindo as regras estabelecidas na sessão de 4.4.

Após o descarte das funcionalidade sem repositórios, restaram 64 funcionalidades, conforme tabela constante no Apêndice B. Apesar dos resultados filtrados, ainda é necessário relacionar os dados de funcionalidade com os dados de cobertura de código.

A Figura 15 mostra um gráfico de bolhas que dispõe a relação entre taxa de cobertura de linhas (eixo  $x$ ) e a quantidade de falhas reportada (eixo  $y$ ). O tamanho das bolhas representa a quantidade de linhas totais da funcionalidade, porém a proporção não é linear.

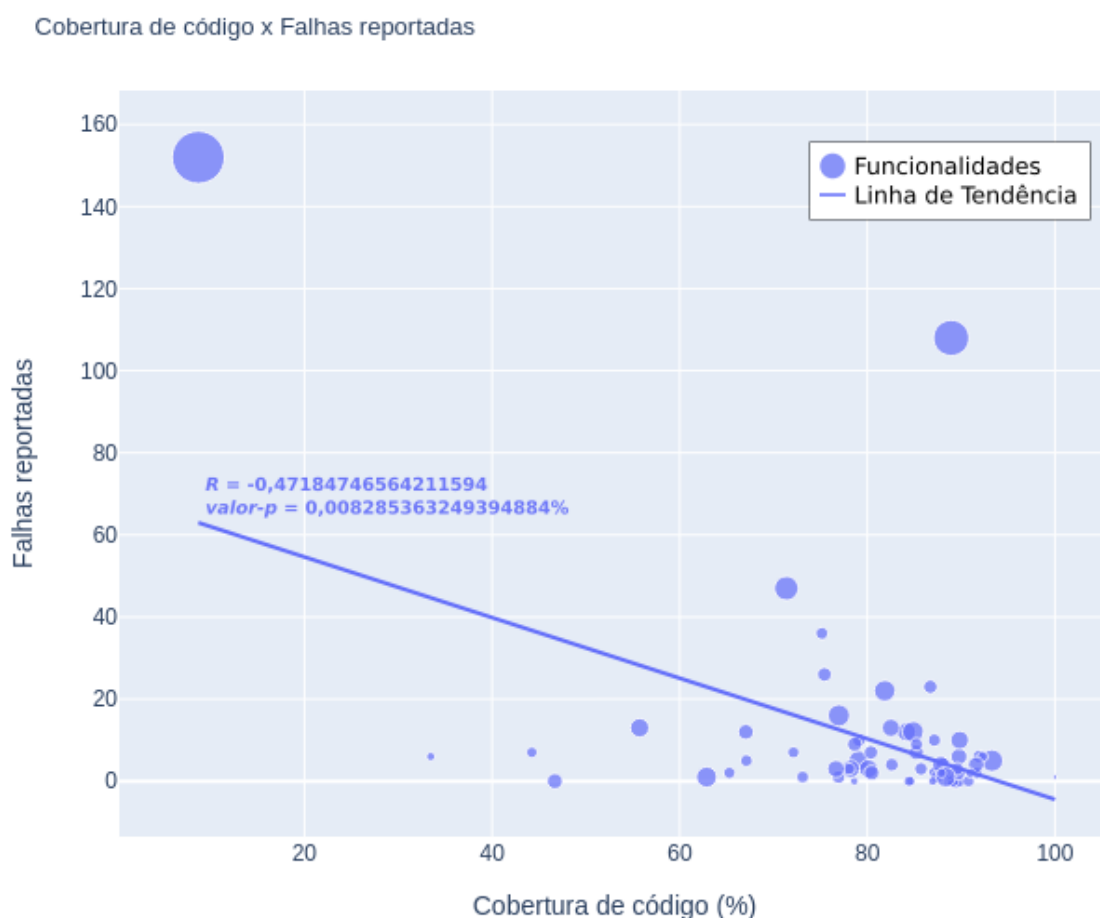


Figura 15. Cobertura de código versus falhas reportadas. Gráfico inclui todos as funcionalidades com pelo menos 1 repositório associado.

Em um primeiro momento é possível observar uma grande quantidade de funcionalidades no quadrante inferior direito, o que representa uma concentração de funcionalidades com alta cobertura de código e baixo número de falhas reportadas. À medida que a cobertura de código diminui, a variabilidade da quantidade de falhas reportadas tende a aumentar, formando uma espécie de cone que diminui sua variabilidade no eixo  $y$  à medida que a cobertura de código, representada no eixo  $x$  aumenta, tendendo a  $y = 0$ .

A linha de tendência, calculada através da regressão linear por Mínimos Quadrados Ordinários, também corrobora a tese de que existe uma correlação. A linha indica uma correlação negativa entre os valores de cobertura de código e falhas reportadas, isto é, a quantidade de falhas reportadas tende a ser menor à medida que a cobertura de código aumenta. Além da linha de tendência, o coeficiente de correlação também indica uma correlação sustentável. O coeficiente de correlação  $R \approx 0,47$  mostra que a correlação não é nula ( $R = 0$ ), mas também não é perfeita ( $R = -1$  ou  $R = 1$ ), ficando no meio do caminho entre ambos.

Como resultado da revisão por pares foi sugerido o cálculo do *valor-p*, que estima a probabilidade de que a hipótese de que existe correlação entre cobertura de código e falhas reportadas seja verdadeira. Podemos utilizar o valor de corte de 5%, sugerido em [Diez et al. 2019], e desta forma temos que  $\text{valor-p} \approx 0.008\%$ , corroborando com a hipótese de que existe correlação entre cobertura de código e falhas reportadas, podendo, desta forma, descartar a hipótese nula.

Outro ponto de interessante de se notar são as duas funcionalidades com as maiores quantidades de falhas reportadas. Ambas as funcionalidades tem grande quantidade de código, indicado pelos tamanhos de suas respectivas bolhas, o que pode indicar que o tamanho da funcionalidade, medido em linhas de código, também possa vir a ser um indicativo de quantidade de falhas reportadas. Contudo, uma vez que o presente estudo não se destina a estudar a relação do tamanho de código com a quantidade de falhas reportadas, não vamos nos aprofundar nesse tema.

Outra visualização possível é a cobertura de código versus falhas reportadas excluindo funcionalidades sem bugs reportados (Figura 16). O objetivo dessa visualização é incluir apenas funcionalidades que se tem certeza que foram utilizadas, uma vez que não é possível descobrir uma falha sem a utilização da funcionalidade. Observando a Figura 16 podemos ver que a tendência permanece muito próxima à Figura 15, mantendo uma inclinação próxima na linha de tendência e com um  $R \approx 0,49$ , valor muito próximo ao  $R \approx 0,47$  da Figura 15. Além disso, temos que  $\text{valor-p} \approx 0.015\%$ , valor ainda bem abaixo da linha de corte de 5%. Isso indica que ambas as visualizações são equivalentes, do ponto de vista estatístico.

Analisando os dados da tabela constante no Apêndice B, podemos separar as funcionalidades em dois grandes grupos: (1) o grupo das funcionalidade que contém falhas reportadas e (2) o grupo das funcionalidades que não contém falhas reportadas. Em termos de média e mediana destes grupos, temos os valores presentes na Tabela 2. Apesar de uma análise bem mais simplificada do que as análises utilizando regressão linear, a análise de médias e medianas também nos dá subsídios para afirmar que existe uma correlação entre cobertura de código e falhas reportadas. O grupo de funcionalidades sem falhas tem 2,39% a mais na média de cobertura de código e 4,49% a mais quando se trata dos valores de mediana.

**Tabela 2. Médias e medianas das taxas de cobertura de código**

<b>Grupo</b>	<b>Média</b>	<b>Mediana</b>
Funcionalidades com falhas reportadas	80,54%	83,40%
Funcionalidades sem falhas reportadas	82,93%	87,89%

Cobertura de código x Falhas reportadas (excluindo funcionalidades sem falhas reportadas)

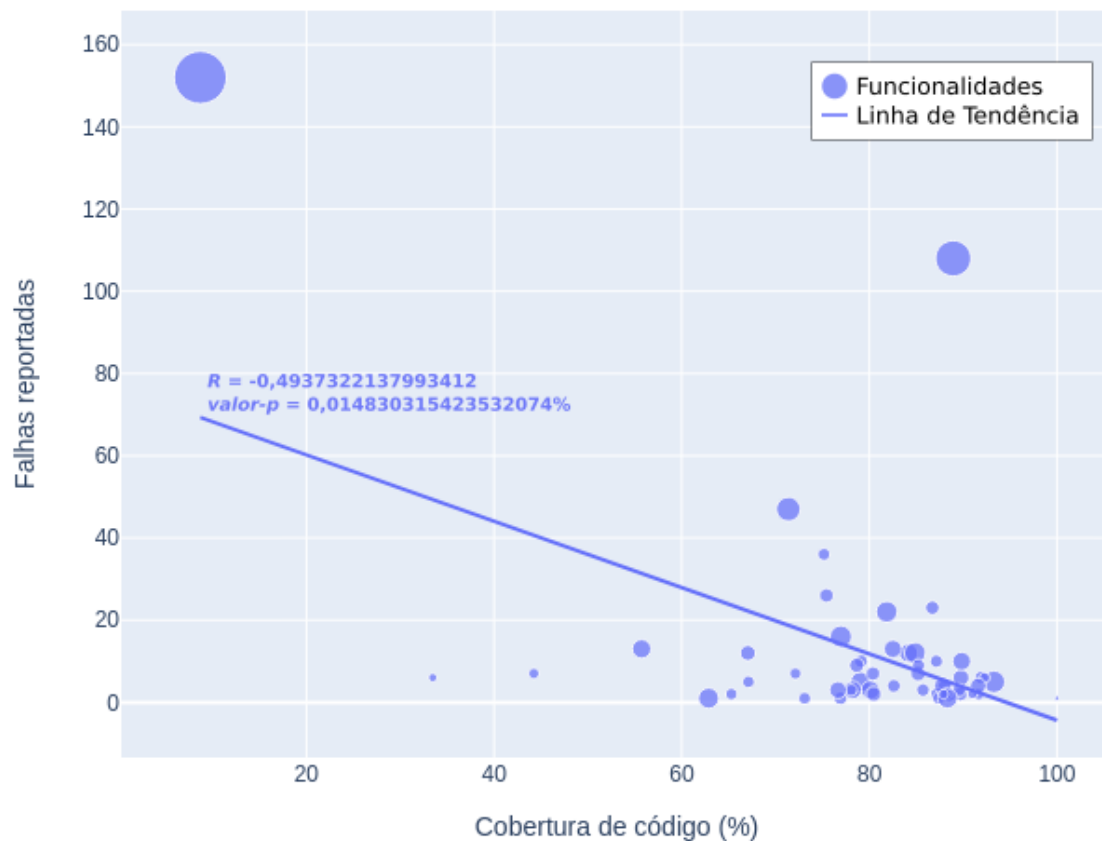


Figura 16. Cobertura de código versus falhas reportadas. Gráfico exclui funcionalidades sem falhas reportadas.

Apesar dessa última análise ser bem mais simplista do que a primeira, é mais um ponto de concordância corroborando para a existência de uma correlação negativa entre cobertura de código e falhas reportadas, isto é, quanto maior a taxa de cobertura de código menor é a tendência de falhas reportadas.

Uma análise mais cuidadosa nos gráficos exibidos nas Figuras 15 e 16 nos traz mais informações sobre a relação entre a cobertura de código e a quantidade de falhas reportadas. Apesar da clara tendência de correlação negativa, podemos notar que um baixo índice de cobertura de código não implica necessariamente em uma alta quantidade de falhas reportadas. Os resultados mostram algumas ocorrências de funcionalidades com menos de 60% de cobertura de código e uma quantidade de falhas reportadas relativamente baixa. Complementarmente, temos pelo menos um exemplo de funcionalidade com mais de 80% de cobertura de código e um número de falhas reportadas claramente maior que a maioria, o que nos mostra que uma alta cobertura de código não garante uma baixa quantidade de falhas reportadas.

Dessa forma, podemos concluir que existem fortes indícios que nos levam a crer que uma alta taxa de cobertura de código influencia em uma baixa quantidade de falhas reportadas, porém uma alta taxa de cobertura de código, por si só, não é uma garantia

de um baixo número de falhas reportadas, apesar de ser um bom índice que indica uma tendência de qualidade de *software*.

### 5.1. Limitações Apresentadas

Nesta sessão serão apresentados alguns pontos de limitação e possíveis vulnerabilidades do presente estudo, a fim de ressaltar pontos de atenção durante a análise dos resultados obtidos. Tais limitações foram percebidas durante o desenvolvimento do estudo e/ou reportadas durante o processo de revisão por pares.

No estudo de caso apresentado neste artigo temos uma análise de 85 funcionalidades distribuídas em 316 repositórios distintos. Ainda que o  $N$  amostral tenha um tamanho relevante, ele apresenta algumas vulnerabilidades, as quais discutiremos neste parágrafo. Todo o código analisado data de diversos períodos, deste meados de 2013, quando o projeto começou a ser implementado até o ano de 2023, quando o presente estudo foi elaborado. Durante todo esse período o projeto contou com dezenas de equipes diferentes e centenas de desenvolvedores. Apesar da grande pluralidade de estilos de desenvolvimento, o projeto sempre esteve dentro da mesma empresa, com a mesma cultura de desenvolvimento, as mesmas linguagens de desenvolvimento e, por fim, atendendo um perfil de cliente que foi pouco modificado durante a vida do projeto. Estas constantes podem gerar um certo viés que imponha os resultados obtidos, sem que estes sejam genéricos o suficiente para que sejam comparáveis com outros projetos. Além disso, as falhas são reportadas por um perfil de usuário específico, o que pode fazer com que os valores de falhas reportadas não sejam comparáveis com os valores de outros projetos. A fim de minimizar essas limitações mais estudos são necessários, envolvendo projetos de tamanhos, perfis e tecnologias distintas.

Outro ponto que pode ser considerado como uma limitação é a métrica de cobertura de código utilizada. Neste estudo utilizamos a métrica de cobertura de linhas de código, porém esta pode não ser a métrica mais adequada para a correlação entre cobertura de código e falhas reportadas. Existem outras métricas de cobertura de código, como cobertura de decisão, que poderiam ter sido utilizadas em conjunto com a métrica de cobertura de linhas, a fim de termos uma maior base de dados para análise, o que poderia gerar conclusões mais precisas a respeito da existência ou não da correlação e de qual a melhor métrica a se utilizar para mensurar tal correlação.

Durante a triagem dos dados foi necessário traçar a relação entre repositórios e funcionalidades de usuário. Tal correlação pode ser bem difusa. Alguns repositórios podem servir a mais de uma funcionalidade em proporções diferentes, bem como podem existir funcionalidades de usuário muito semelhantes e a classificação das falhas pode ficar dificultada. Essas questões de classificação, tanto das falhas, quanto dos repositórios, são feitas por pessoas e, devido à característica difusa, podem ter sido classificadas de forma não adequada. Essas falhas de classificação podem gerar alterações tanto positivas, quanto negativas nos resultados, o que nos traz mais um ponto de atenção. Uma vez que as classificações são feitas qualitativamente, elas estão sujeitas a erro humano e podem representar uma menor precisão dos resultados.

Alguns dos repositórios foram classificados como **repositórios de sistema**. Essa denominação denota repositórios que não pertencem a apenas uma funcionalidade, atendendo a duas ou mais funcionalidades simultaneamente. Tais repositórios não foram con-



siderados nos cálculos e análises, uma vez que não seria possível atribuir seus dados de cobertura a uma só funcionalidade. Essa abordagem deixa de fora alguns repositórios, o que pode impactar nos resultados, visto que tais repositórios podem ser responsáveis por falhas das funcionalidades, mas não são contabilizados nas estatísticas de cobertura de código da funcionalidade. Para tornar a abordagem mais precisa, seria necessário obter mais informações sobre a influência de **repositórios de sistema** nas funcionalidades às quais eles interferem.

Da perspectiva dos testes utilizados como base das informações de cobertura de código, podemos observar mais uma limitação metodológica. O conjunto de testes que geram informações de cobertura de código é um subconjunto do total de testes existentes. Os testes que executam no equipamento destino não geram estatísticas de cobertura de código pois a ferramenta **gcov** não está embarcada no equipamento. Cada funcionalidade pode ter uma proporção diferente de testes com e sem estatísticas de cobertura de código, o que pode interferir na comparação entre funcionalidades.

Tais limitações e vulnerabilidades não invalidam o resultado do estudo, porém indicam pontos que devem ser observado durante a leitura e interpretação dos resultados. De maneira geral, podemos compreender que os resultados observados são válidos no contexto do projeto estudado e durante o período observado, não sendo garantida uma generalização dos resultados para outros projetos. Apesar disso, os dados obtidos podem ser utilizados como indícios quando analisados em conjunto com mais pesquisas do mesmo tema, ou de pesquisas mais abrangentes. Além disso, é importante ressaltar que o presente artigo se propõe a um estudo de **correlação** e, com as informações que temos no contexto estrito deste estudo, não é possível garantir uma correlação causal.

## 6. Considerações Finais

A etapa de teste de *software* é uma etapa de extrema importância dentro do processo de desenvolvimento de *software* de times e empresas no mundo inteiro. Essa etapa visa encontrar os defeitos de um sistema antes destes serem dados como prontos, a fim de garantir a qualidade do *software* desenvolvido. Contudo, as tarefas de planejamento e escrita de testes podem demandar recursos relevantes para os projetos.

Como forma de mensuração da qualidade e abrangência dos testes existem diversas métricas utilizadas pelos desenvolvedores e uma delas é a cobertura de código. Tal métrica se propõe a mensurar a quantidade de código testado no universo de todo o código em questão, gerando um dado percentual. Esse dado diz estritamente a porcentagem de código testado, porém é possível usá-lo para inferir outras informações. Uma das informações que se tenta inferir é a qualidade do código, que pode ser mensurada através de várias métricas e uma delas é a quantidade de falhas reportadas, contudo a academia não tem um consenso sobre se existe, de fato, uma correlação entre as métricas de cobertura de código e qualidade de *software*.

A fim de verificar se é possível traçar uma correlação entre cobertura de código e qualidade do *software*, este artigo faz um estudo de caso em um projeto de sistema operacional embarcado a fim de estudar tal correlação. Para tal, foram feitas medições de cobertura de código em todo o projeto, bem como foram coletados dados de falhas reportadas nas funcionalidade do sistema operacional estudado. Os dados foram triados, classificados e analisados separando as funcionalidades, para que se possa comparar os

números de cobertura de código e falhas reportadas entre funcionalidades distintas.

Como resultado o estudo concluiu que existe uma correlação negativa entre cobertura de código e falhas reportadas no sistema operacional estudado. Isso significa que para o contexto estudado a quantidade de falhas cai à medida que a taxa de cobertura de código aumenta. É importante salientar que tal correlação não implica em causalidade e o presente artigo não se propõe a investigar se há causalidade na correlação, contudo temos indícios para acreditar que tal causalidade existe. Os indícios se baseiam nos cálculos de coeficiente de correlação  $R$  e *valor-p*. O primeiro indica a força de correlação dos dados estudados, ou seja, a aderência dos dados à linha de correlação traçada, enquanto o segundo indica, em linhas gerais, se a hipótese de correlação levantada no presente estudo pode ser considerada válida. Após a análise de todos os dados estatísticos foi observado que tais métricas corroboram para a hipótese de que, sim, existe uma correlação negativa entre a taxa de cobertura de código e o número de falhas reportadas.

Os resultados encontrados se somam a um grande conjunto de trabalhos acadêmicos que investigam esse tema, contribuindo para o entendimento acadêmico na Engenharia de Software no que tange aos estudos de qualidade de *software* e pode ser utilizado como insumo para estudo futuros. Além disso, os resultados obtidos contribuem com a indústria de desenvolvimento de *software*, uma vez que corroboram com a hipótese de que a taxa de cobertura de código pode, sob certas circunstâncias, ser utilizada como uma métrica de qualidade de *software*.

Estudos futuros podem cobrir aspectos não cobertos pelo presente trabalho acadêmico a fim de gerar um conhecimento ainda mais completo sobre o tema e sanar alguns pontos que ficaram abertos no presente estudo. Nesse sentido, algumas sugestões de temas para estudos futuros são:

- Comparação de cobertura de linhas e cobertura de decisões como métrica de qualidade de *software*;
- Estudo da correlação entre cobertura de código e qualidade de *software* utilizando diversos projetos e diversas linguagens;
- Estudo da correlação entre cobertura de código e qualidade de *software* embarcado utilizando cobertura de código gerada por testes em equipamentos.

## Referências

- Aniche, M. (2012). *Test-Driven Development*. Casa do Código.
- Azevedo, D., Machado, L., e da Silva, L. V. (2011). *MÉTODOS E PROCEDIMENTOS DE PESQUISA*. Editora Unisinos.
- Barani, M., Labiche, Y., e Rollet, A. (2023). On factors that impact the relationship between code coverage and test suite effectiveness: a survey. *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 15:381–388.
- BUGZILLA (2023). Bugzilla issue tracker. Disponível em: <https://www.bugzilla.org>. Acesso em: 15 Nov. 2023.
- Chioteli, E., Batas, I., e Spinellis, D. (2021). Does unit-tested code crash? a case study of eclipse. *Pan-Hellenic Conference on Informatics (PCI)*, 25:260–264.
- COBERTURA (2023). Cobertura jenkins plugin. Disponível em: <https://plugins.jenkins.io/cobertura/>. Acesso em: 15 Nov. 2023.
- Creswell, J. W. (2007). *Projeto de Pesquisa*. Bookman.
- Diez, D., Çetinkaya Rundel, M., e Barr, C. D. (2019). *OpenIntro Statistics*. OpenIntro.
- GCC (2023). Gcc compiler. Disponível em: <https://gcc.gnu.org>. Acesso em: 15 Nov. 2023.
- GCOV (2023). gcov – gnu compiler coverage tool. Disponível em: <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>. Acesso em: 15 Nov. 2023.
- Gil, A. C. (2008). *Métodos e Técnicas de Pesquisa Social*. Atlas.
- GTEST (2023). Google test framework. Disponível em: <https://github.com/google/googletest>. Acesso em: 15 Nov. 2023.
- Ivanković, M., Petrović, G., Just, R., e Fraser, G. (2019). Code coverage at google. *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 27:995–963.
- JENKINS (2023). Jenkins automation server. Disponível em: <https://www.jenkins.io/>. Acesso em: 15 Nov. 2023.
- Kochhar, P. S., Lo, D., Lawall, J., e Nagappan, N. (2017). Code coverage and postrelease defects: A large-scale study on open source projects. *IEEE Transactions on Reliability*, 66:1213–1218.
- Maldonado, J. C., Jino, M., e Delamaro, M. (2016). *Introdução ao teste de software*. Elsevier.
- PLOTLY (2023). Plotly open source graphing library for python. Disponível em: <https://plotly.com/python/>. Acesso em: 15 Nov. 2023.
- SCIPY (2023). Scipy – fundamental algorithms for scientific computing in python. Disponível em: <https://docs.scipy.org/>. Acesso em: 15 Nov. 2023.
- Sommerville, I. (2011). *Engenharia de Software*. Pearson Education do Brasil.
- Yin, R. K. (2001). *Estudo de Caso*. Bookman.

## APÊNDICE A – Tabela de funcionalidades, falhas e repositórios relacionados

Funcionalidade	Falhas	Repositórios	Funcionalidade	Falhas	Repositórios
Funcionalidade_001	7	2	Funcionalidade_044	6	1
Funcionalidade_002	0	0	Funcionalidade_045	2	1
Funcionalidade_003	0	0	Funcionalidade_046	6	2
Funcionalidade_004	0	0	Funcionalidade_047	26	4
Funcionalidade_005	0	0	Funcionalidade_048	5	2
Funcionalidade_006	0	0	Funcionalidade_049	12	3
Funcionalidade_007	0	0	Funcionalidade_050	4	2
Funcionalidade_008	0	1	Funcionalidade_051	2	2
Funcionalidade_009	0	0	Funcionalidade_052	2	1
Funcionalidade_010	4	0	Funcionalidade_053	7	5
Funcionalidade_011	108	26	Funcionalidade_054	22	7
Funcionalidade_012	8	0	Funcionalidade_055	2	2
Funcionalidade_013	1	0	Funcionalidade_056	10	2
Funcionalidade_014	14	0	Funcionalidade_057	152	12
Funcionalidade_015	2	0	Funcionalidade_058	1	19
Funcionalidade_016	2	0	Funcionalidade_059	3	1
Funcionalidade_017	6	0	Funcionalidade_060	13	4
Funcionalidade_018	4	5	Funcionalidade_061	1	2
Funcionalidade_019	36	3	Funcionalidade_062	3	1
Funcionalidade_020	47	3	Funcionalidade_063	10	3
Funcionalidade_021	1	1	Funcionalidade_064	6	7
Funcionalidade_022	3	0	Funcionalidade_065	3	2
Funcionalidade_023	23	0	Funcionalidade_066	2	2
Funcionalidade_024	9	0	Funcionalidade_067	4	6
Funcionalidade_025	2	0	Funcionalidade_068	9	1
Funcionalidade_026	2	0	Funcionalidade_069	1	5
Funcionalidade_027	3	0	Funcionalidade_070	2	2
Funcionalidade_028	0	8	Funcionalidade_071	6	5
Funcionalidade_029	0	1	Funcionalidade_072	3	5
Funcionalidade_030	0	2	Funcionalidade_073	2	1
Funcionalidade_031	0	4	Funcionalidade_074	23	1
Funcionalidade_032	0	1	Funcionalidade_075	4	2
Funcionalidade_033	0	4	Funcionalidade_076	7	1
Funcionalidade_034	0	2	Funcionalidade_077	2	4
Funcionalidade_035	2	0	Funcionalidade_078	3	2
Funcionalidade_036	0	2	Funcionalidade_079	13	7
Funcionalidade_037	12	1	Funcionalidade_080	12	4
Funcionalidade_038	5	3	Funcionalidade_081	10	5
Funcionalidade_039	1	2	Funcionalidade_082	9	1
Funcionalidade_040	7	2	Funcionalidade_083	1	3
Funcionalidade_041	16	2	Funcionalidade_084	2	1
Funcionalidade_042	3	2	Funcionalidade_085	0	1
Funcionalidade_043	5	2			

**APÊNDICE B – Tabela de funcionalidades, falhas e repositórios relacionados,  
excluindo funcionalidades sem repositórios relacionados**

<b>Funcionalidade</b>	<b>Falhas</b>	<b>Repositórios</b>	<b>Funcionalidade</b>	<b>Falhas</b>	<b>Repositórios</b>
Funcionalidade_001	7	2	Funcionalidade_054	22	7
Funcionalidade_008	0	1	Funcionalidade_055	2	2
Funcionalidade_011	108	26	Funcionalidade_056	10	2
Funcionalidade_018	4	5	Funcionalidade_057	152	12
Funcionalidade_019	36	3	Funcionalidade_058	1	19
Funcionalidade_020	47	3	Funcionalidade_059	3	1
Funcionalidade_021	1	1	Funcionalidade_060	13	4
Funcionalidade_028	0	8	Funcionalidade_061	1	2
Funcionalidade_029	0	1	Funcionalidade_062	3	1
Funcionalidade_030	0	2	Funcionalidade_063	10	3
Funcionalidade_031	0	4	Funcionalidade_064	6	7
Funcionalidade_032	0	1	Funcionalidade_065	3	2
Funcionalidade_033	0	4	Funcionalidade_066	2	2
Funcionalidade_034	0	2	Funcionalidade_067	4	6
Funcionalidade_036	0	2	Funcionalidade_068	9	1
Funcionalidade_037	12	1	Funcionalidade_069	1	5
Funcionalidade_038	5	3	Funcionalidade_070	2	2
Funcionalidade_039	1	2	Funcionalidade_071	6	5
Funcionalidade_040	7	2	Funcionalidade_072	3	5
Funcionalidade_041	16	2	Funcionalidade_073	2	1
Funcionalidade_042	3	2	Funcionalidade_074	23	1
Funcionalidade_043	5	2	Funcionalidade_075	4	2
Funcionalidade_044	6	1	Funcionalidade_076	7	1
Funcionalidade_045	2	1	Funcionalidade_077	2	4
Funcionalidade_046	6	2	Funcionalidade_078	3	2
Funcionalidade_047	26	4	Funcionalidade_079	13	7
Funcionalidade_048	5	2	Funcionalidade_080	12	4
Funcionalidade_049	12	3	Funcionalidade_081	10	5
Funcionalidade_050	4	2	Funcionalidade_082	9	1
Funcionalidade_051	2	2	Funcionalidade_083	1	3
Funcionalidade_052	2	1	Funcionalidade_084	2	1
Funcionalidade_053	7	5	Funcionalidade_085	0	1