ADRIANA PRISCILA SANTOS CRUZ

UMA ABORDAGEM VISUAL PARA EVOLUÇÃO DE *TEST SMELLS* EM SOFTWARE JAVA

LAVRAS-MG
2022

ADRIANA PRISCILA SANTOS CRUZ

UMA ABORDAGEM VISUAL PARA EVOLUÇÃO DE *TEST SMELLS* EM SOFTWARE JAVA

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Engenharia de Software e Banco de dados, para a obtenção do título de Mestre.

Prof. Dr. Heitor Augustus Xavier Costa
Orientador

LAVRAS-MG
2022

Ficha catalográfica elaborada pelo Sistema de Geração de Ficha Catalográfica da Biblioteca Universitária da UFLA, com dados informados pelo(a) próprio(a) autor(a).

Cruz, Adriana Priscila Santos.
Uma Abordagem Visual para a Evolução de Test Smells em Software Java/ Adriana Priscila Santos Cruz - 2022.
230 p.

Orientador(a): Heitor Augustus Xavier Costa.

Dissertação (mestrado acadêmico) - Universidade Federal de Lavras, 2022.
Bibliografia.

1. Engenharia de Software. 2. Test Smells. 3. Visualização de Software.

Adriana Priscila Santos Cruz

UMA ABORDAGEM VISUAL PARA EVOLUÇÃO DE *TEST SMELLS* EM SOFTWARE JAVA

A VISUAL APPROACH FOR EVOLUTION OF TEST SMELLS IN JAVA SOFTWARE

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Engenharia de Software e Banco de dados, para a obtenção do título de Mestre.

APROVADA em 28/03/2022
Dr. Ivan Do Carmo Machado UFBA
Dra. Larissa Rocha Soares UEFS

Prof. Dr. Heitor Augustus Xavier Costa
Orientador

LAVRAS-MG
2022

AGRADECIMENTOS

Em primeiro lugar, agradeço a Deus por ter me proporcionado chegar até aqui, por ter me guiado, me dado coragem, sabedoria para atingir meus objetivos e enfrentar todas as dificuldades. Sem Ele eu não teria persistido.

Aos meus pais, pela educação que me deram, por estarem sempre do meu lado e por acreditarem em mim.

A toda minha família, especialmente, ao meu marido por caminhar comigo junto aos meus objetivos compreendendo meus momentos de ausência. Aos meus irmãos pelo apoio que me fizeram sentir que nunca estaria só.

A Universidade Federal de Lavras (UFLA) pela oportunidade de realizar o Programa de Pós-Graduação em Ciência da Computação.

Ao meu orientador Dr. Heitor Augustus Xavier Costa pela oportunidade de trabalhar ao seu lado e por todo apoio.

Aos colegas do grupo de pesquisa Aries Lab, pela troca de conhecimento.

A todos os amigos, que estiveram comigo nessa caminhada e me deram força.

A todos que contribuíram para a realização deste trabalho e a todos que fazem parte da minha vida.

RESUMO

O teste de software é parte do ciclo de desenvolvimento de software, sendo o processo de executar as tarefas do sistema cujo objetivo principal é identificar se o software se comporta conforme foi especificado. O teste também tem como objetivo identificar defeitos no código produzido. Durante esse processo, os desenvolvedores e testadores podem incluir más escolhas de projeto ou implementação do código de teste, levando a inserção dos denominados *test smells*. *Test smells* são más escolhas de *design* ou de implementação do código de teste e sua presença pode torna-lo ineficaz em encontrar *bugs*, gerando retrabalho e custos adicionais. *Test smells* também podem dificultar a compreensão e manutenibilidade do código. Além disso, assim como qualquer artefato de software, o código de teste requer a avaliação da qualidade e manutenção. A manutenção do código de teste pode ocorrer devido a identificação de problemas no código e a evolução do código de produção. Ao longo da evolução do código de teste alguns *test smells* podem ser removidos e outros podem ser inseridos. Para visualizar a evolução de sistemas de software, têm sido utilizadas técnicas de visualização de software, cujo objetivo é transformar o código de produção em uma representação visual para facilitar a sua compreensão. Apesar dessas técnicas contribuírem para visualizar a evolução de sistemas de software, essa abordagem não abrange o código de teste para visualização de *test smells*. Dessa forma, o objetivo desse trabalho foi propor uma abordagem para visualizar a evolução dos *test smells* no código de teste e seu comportamento, por exemplo, se houve aumento ou diminuição de *test smells*, e visualizar os possíveis autores pela sua inclusão. Foi proposta uma abordagem que definiu três estatrégias visuais (*TSInstant, TSEvolution* e *TSAuthor*) e uma ferramenta que implementa essa abordagem através de três técnicas de visualização de software (*Graph View*, *Treemap View e Timeline View*) para 21 *test smells*. Essa abordagem foi implementada em uma ferramenta denominada `TSVizzEvolution` e foi avaliada por meio de um experimento controlado. Essas visualizações podem ajudar os testadores e demais envolvidos nas atividades de teste de *software* a melhorar a qualidade do código de teste.

Palavras-chave: *Test Smells*, Visualização de Software, Evolução de Software.

ABSTRACT

Software testing is part of the software development cycle, being the process of performing system tasks whose main objective is to identify whether the software behaves as specified. Testing also aims to identify defects in the code produced. During this process, developers and testers may include poor design or implementation choices in the test code, leading to the insertion of so-called test smells. Test smells are bad design or implementation choices for test code and their presence can make it ineffective at finding bugs, generating rework and additional costs. Test smells can also make code difficult to understand and maintain. Also, like any software artifact, test code requires quality and maintainability assessment. Test code maintenance can occur due to the identification of code problems and the evolution of the production code. As the test code evolves, some test smells may be removed and others may be inserted. To visualize the evolution of software systems, software visualization techniques have been used, whose objective is to transform the production code into a visual representation to facilitate its understanding. Although these techniques help to visualize the evolution of software systems, this approach does not cover test code to visualize test smells. Thus, the objective of this work was to propose an approach to visualize the evolution of test smells in the test code and its behavior, for example, if there was an increase or decrease of test smells, and visualize the possible authors for their inclusion. An approach was proposed that defined three visual strategies (TSInstant, TSEvolution and TSAuthor) and a tool that implements this approach through three software visualization techniques (Graph View, Treemap View and Timeline View) for 21 types of test smells. This approach was implemented in a tool called TSVizzEvolution and was evaluated through a controlled experiment. These views can help testers and others involved in software testing activities improve the quality of test code.

Keywords: Test Smells, Software Visualization, Software Evolution.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

Teste de software é o processo de executar o software de maneira controlada para verificar se ele se comporta conforme o especificado, sendo fundamental para a sua avaliação durante seu desenvolvimento (CRESPO *et al*., 2004). O teste não garante um software livre de erros, porém é necessário ter testes durante o projeto de sistemas de software para minimizar os erros e diminuir os custos (SILVA *et al*., 2016).

O teste de software pode ser realizado de maneira manual ou automatizada. No teste manual, o testador assume o papel de usuário executando o sistema de software para verificar seu comportamento e encontrar defeitos observáveis. No teste automatizado, *scripts* de código de teste são desenvolvidos utilizando ferramentas de teste, por exemplo, JUnit[1] (PALOMBA; ZAIDMAN, 2017). Testes automatizados, se bem planejados, podem trazer benefícios em comparação com o teste manual, por exemplo, no teste manual, o testador precisar repetir diversas vezes a mesma atividade para verificar o comportamento; por outro lado, em testes automatizados, há ferramentas computacionais que realizam testes diversas vezes de forma mais rápida. Outro benefício é a redução de custos, testes mais rápidos geram entregas mais rápidas e tornam-se menos dispendiosos para o testador. Porém, se o teste automatizado for mal projetado, ele pode falhar na detecção de erros, fazendo com que ocorram erros no código de produção e, consequentemente, levar a custos extras (GAROUSI *et al*., 2018).

Os testes automatizados têm sido amplamente utilizados na indústria por causa de seus benefícios em relação aos testes manuais. A Accenture[2] investe em manutenção e evolução de testes US$50-$120 milhões por ano (GRECHANIK *et al*., 2009), e para o Microsoft Office 2007[3], havia mais de 1 milhão de casos de testes (GAROUSI *et al*., 2018). Por causa da quantidade de código de teste que as empresas lidam atualmente, começou a ter preocupação com a qualidade dos testes, porém, na prática, ocorrem más escolhas de *design* ou de implementação do código de teste. Esses problemas são denominados *test smells* e sua presença pode afetar negativamente a compreensão, a evolução e a manutenção do código de teste (CAMPOS *et al.*, 2021) e ocasionar testes menos eficazes em encontrar *bugs* no código de produção (SPADINI *et al*., 2018). Para apoiar o desenvolvimento de código de teste, existem boas práticas de programação a serem seguidas para evitar a inserção de *test smells*. Por exemplo, algumas metodologias ágeis (*e.g.*, *Extreme Programming* - XP) recomendam a criação de uma classe de teste para cada classe do sistema. Com essa quantidade de testes, pode-

[1] https://junit.org/
[2] https://www.accenture.com/br-pt
[3] https://www.microsoft.com/pt-br/microsoft-365/previous-versions/download-office-2007

se ter como vantagem a identificação de problemas sem afetar o código de produção, pois, para um sistema de software com milhares de classes, tem-se milhares de classes de testes, facilitando a detecção de problemas nesse sistema (DEURSEN *et al*., 2001).

1.1 Motivação

As boas práticas de programação nem sempre são seguidas, resultando na inserção de *test smells* no código de teste (MESZAROS, 2007; PERUMA *et al.,* 2019). Portanto, o código das classes de teste deve ser verificado continuamente quanto aos possíveis problemas, os quais devem ser removidos, quando encontrados (TAHIR *et al.*, 2016). Diante disso, algumas ferramentas foram propostas para automatizar a detecção e a correção de *test smells*. Dentre essas ferramentas, as que detectam maior quantidade de *test smells* são `tsDetect`[4] (PERUMA, 2021) que identifica a presença de 21 *test smells* e a `JNose Test`[5] (VIRGÍNIO *et al*., 2020), que estende `tsDetect` e quantifica cada os *test smells* por classes de teste, apresentando os resultados em arquivos `.csv`. No entanto, a complexidade e a quantidade significativa de dados gerados podem prejudicar a análise dos dados, pois a representação dos dados geralmente não é intuitiva, o que pode dificultar a geração de conhecimento.

Dessa maneira, técnicas e ferramentas de visualização de software representam uma estratégia para facilitar a compreensão dos dados (CASERTA; ZENDRA, 2011). Por exemplo, `VITRuM`[6] (PECORELLI *et al*., 2020) e `TestQ`[7] (BREUGELMANS; ROMPAEY, 2008) são ferramentas que utilizam recursos visuais relacionados a *test smells*. `VITRuM` apresenta uma correlação entre os *test smells* e medidas de código e mostra esses dados na forma de um gráfico de linha (PECORELLI *et al*., 2020). `TestQ` apresenta a associação entre *test smells* e os métodos das classes de teste usando diferentes visualizações (BREUGELMANS; ROMPAEY, 2008). Contudo, `TestQ` e `VITRuM` não exibem o comportamento dos *test smells* durante a evolução do código de teste. Além disso, não há visualização para apresentar um relacionamento entre classes de teste, *test smells* e (possíveis) autores de *test smells*. Essas ferramentas fornecem maneiras de mostrar informações sobre *test smells*, porém outras técnicas de visualização podem ser exploradas para apresentar mais informações para ajudar nas atividades dos envolvidos nas atividades de teste a melhorar a qualidade do código de teste. Por

[4] https://testsmells.github.io/index.html
[5] https://github.com/arieslab/jnose
[6] https://plugins.jetbrains.com/plugin/14160-vitrum
[7] https://code.google.com/archive/p/tsmells/

isso, neste trabalho, é apresentada uma abordagem para exibir as ocorrências, a evolução e os possíveis responsáveis de *test smells*.

1.2 Objetivo

A hipótese sobre a qual este trabalho baseia-se é se a visualização de *test smells* pode facilitar a sua identificação e, consequentemente, facilitar a compreensão de problemas no código de teste, minimizando a ocorrência de erros em sistemas de software.

Assim, o objetivo é elaborar uma abordagem visual para auxiliar os envolvidos nas atividades de testes na visualização das ocorrências e da evolução de *test smells* e saber (possivelmente) os responsáveis pelos *test smells*, facilitando a visualização de *test smells* e, consequentemente, a compreensão do código de teste. Auxiliando-os nas suas atividades, facilita-se a elaboração de testes sem erros para garantir a qualidade do código de teste. Para isso, foram escolhidas três técnicas de visualização: i) *Graphs* (CRUZ *et al*., 2016); ii) *Timeline* (NOVAIS *et al*., 2012); e iii) *Treemap* (CRUZ *et al*., 2016). Para alcançar esse objetivo, os seguintes procedimentos foram necessários:

a) **Realizar uma pesquisa na literatura**. Pesquisar o estado da arte sobre uso de técnicas de visualização de software para *test smells*;
b) **Elaborar uma abordagem visual para apoiar os envolvidos nas atividades de testes**. Elaborar uma abordagem visual para representar as ocorrências, a evolução e os possíveis autores de *test smells* no código de teste;
c) **Definir o uso técnicas de visualização**. Verificar quais são as técnicas de visualização mais adequadas para exibir as ocorrências, a evolução de *test smells* e os seus possíveis autores;
d) **Implementar uma ferramenta**. Desenvolver uma ferramenta para automatizar a abordagem elaborada;
e) **Avaliar a abordagem**. Avaliar, por meio de um experimento controlado, se a abordagem implementada na ferramenta facilitou a tarefa de visualizar *test smells* e seus possíveis autores durante a evolução do código de teste comparando-a com outra ferramenta do estado da arte.

1.3 Questões de pesquisa

Para atingir o objetivo de avaliar como a abordagem auxilia na visualização de *test smells* por meio da ferramenta `TSVizzEvolution`, foi definida a questão de pesquisa principal (QP) e da questão principal foram geradas duas questões (QP1, QP2):

> Questão de Pesquisa (QP): Como a abordagem, por meio da ferramenta `TSVizzEvolution`, auxilia os usuários a visualizar *test smells* e tomar decisões de projeto?

O objetivo da QP é identificar se a abordagem implementada na ferramenta `TSVizzEvolution` auxiliam os usuários a visualizar os *test smells* de forma mais intuitiva, a entender onde ocorrem esses problemas no código durante a evolução do sistema de software e auxiliar nas decisões de projeto e nos processos de tomada de decisão.

> QP1: Como a ferramenta `TSVizzEvolution` pode facilitar as decisões de projeto e os processos de tomada de decisão?

O objetivo da QP1 é entender se as visualizações da `TSVizzEvolution` podem auxiliar os envolvidos nas atividades de testes a tomar decisões de projeto. Para responder QP1, foram feitas perguntas durante as tarefas realizadas no experimento sobre a percepção dos participantes quanto às informações que estavam visualizando dos *test smells* se auxiliaria nas decisões de projeto e nos processos de tomada de decisão.

> QP2: Como a ferramenta `TSVizzEvolution` facilita a visualização de *test smells* em comparação com a ferramenta `VITRuM`?

O objetivo da QP2 é entender se as visualizações da `TSVizzEvolution` facilitam a representação dos *test smells* em comparação com as visualizações da `VITRuM`. Para responder a QP2, foram comparadas as duas ferramentas em relação a exibição dos *test smells* e a facilidade em que a visualização proporciona para que os usuários das ferramentas possam identificar e corrigir os problemas no código de teste. Esses questionamentos foram abordados no questionário ao final do experimento.

Para responder as questões de pesquisas foram criadas tarefas semelhantes no experimento controlado para ambas ferramentas. Cada participante realizou as tarefas com `TSVizzEvolution` e `VITRuM` e respondeu 11 perguntas para cada ferramenta. As tarefas consistiam em visualizar os *test smells*, cujo objetivo era analisar como os usuários conseguem visualizar as ocorrências, a evolução e os possíveis autores de *test smells*.

1.4 Estrutura do Trabalho

O restante do trabalho está organizado da seguinte forma. O método de pesquisa utilizado nesse trabalho é descrito no Capítulo 2. A fundamentação teórica sobre *test smells*, a visualização e a evolução de software são apresentadas no Capítulo 3. A abordagem proposta e

a ferramenta desenvolvida para visualização de *test smells* são relatadas no Capítulo 4. A avaliação da abordagem e da ferramenta é descrita no Capítulo 5. Alguns trabalhos relacionados são abordados no Capítulo 6. As considerações finais são apresentadas no Capítulo 7.

2 MÉTODO DE PESQUISA

Neste capítulo, são apresentadas as classificações e as etapas de desenvolvimento da pesquisa realizada. Essa pesquisa pode ser classificada (Prodanov; Freitas, 2013):

a) **Quanto à natureza**. Esse trabalho pode ser classificado como **pesquisa aplicada**, pois está associada a uma aplicação de visualização de código de teste, contribuindo para a visualização de *test smells* durante a evolução desses sistemas e seus possíveis autores;

b) **Quanto aos objetivos**. Esse trabalho pode ser caracterizado como **pesquisa explicativa**, pois a abordagem desenvolvida é aplicada e avaliada. A pesquisa explicativa é caracterizada pelo fato do pesquisador procurar por fatores que contribuem ou determinam a ocorrência de um evento, o que é feito nesse trabalho ao visualizar *test smells* durante a evolução de sistemas de software, para identificar se sua quantidade aumenta/diminui com a evolução, e os possíveis autores;

c) **Quanto à abordagem**. Esse trabalho pode ser caracterizado como **pesquisa qualitativa,** pois, após a aplicação da abordagem, foi possível descrever e qualificar o comportamento dos *test smells*, como a quantidade de ocorrências de *test smells* no código de teste durante a sua evolução. Além disso, foram obtidos o tempo gasto para a visualização e os possíveis autores dos *test smells*;

d) **Quanto aos procedimentos técnicos**. Essa pesquisa pode ser classificada como um **estudo de caso**, que consiste em coletar e analisar informações sobre o assunto da pesquisa. Foi feito um estudo das técnicas de visualização de software e *test smells* e foi proposta uma solução.

Com base nessa classificação, foi elaborado o método de pesquisa para a realização deste trabalho (Figura 2.1):

a) **Etapa 1 - Revisão da Literatura**. Nessa etapa, foi feita uma revisão não sistemática (*ad hoc*), onde foram buscados na literatura artigos sobre *test smells*, estratégias visuais para representar *test smells* e técnicas de visualização de software, em bases de bibliotecas científicas virtuais utilizadas no meio acadêmico;

b) **Etapa 2 - Abordagem**. Nessa etapa, foi elaborada a abordagem para a representação visual das ocorrências, evolução e possíveis autores de *test smells*. Foram definidas três estratégias de análise visual para *test smells:* i) *TSInstant*; ii) *TSAuthor*; e iii) *TSEvolution.*

c) **Etapa 3 - Definição das Técnicas para a Visualização de *Test Smells***. Nessa fase, foram definidas as técnicas de visualização mais adequadas para as estratégias visuais criadas na abordagem. A seguir as estratégias e as técnicas de visualização: i) *TSInstant*: *Graphs e*

Treemap; ii) *TSAuthor*: *Graphs*; e iii) *TSEvolution*: *Timeline. Graphs* foi escolhida por mostrar a ligação entre os artefatos de software, podendo representar os *test smells* e suas relações com o projeto, com as classes de teste, com os métodos e com autores, permitindo visualizar as informações instantaneamente. *Treemap* foi escolhida por apresentar a proporcionalidade das ocorrências por meio de retângulos aninhados, podendo representar os *test smells* com maiores ocorrências com os retângulos maiores, também instantaneamente. *Timeline* foi escolhida por ser mais intuitiva para exibir a evolução de *test smells*, pois permite visualmente a comparação de versões do código de teste por meio de sua representação de retângulos em uma linha do tempo;

Figura 2.1 - Método de Pesquisa

Fonte: Do autor (2021).

d) **Etapa 4 - Ferramenta**. Nessa etapa, para automatizar a abordagem proposta, as estratégias visuais foram adaptadas a três técnicas de visualização de software e implementadas em uma ferramenta computacional, que recebe como entrada o arquivo de saída da ferramenta `JNose Test`;

e) **Etapa 5 - Avaliação**. Após a finalização, a abordagem proposta foi avaliada por meio da ferramenta para identificar se facilitou o processo de visualização de *test smells* perante a evolução do código de teste. Para isso, foi realizado um experimento controlado com o uso do *design* quadrado latino e aplicação de questionários. A ferramenta proposta foi comparada a outra ferramenta do estado da arte e os resultados foram analisados qualitativamente. Após o uso da abordagem e da ferramenta, foi possível descrever e qualificar o comportamento dos *test smells*, como identificar a quantidade de ocorrências de *test smells* em código de teste durante a sua evolução e determinar os possíves autores dos *test smells*. Além disso, foram obtidas as percepções dos participantes sobre a ferramenta proposta e o tempo gasto para a visualização de *test smells*.

3 FUNDAMENTAÇÃO TEÓRICA

3.1 Considerações Iniciais

Com o intuito de minimizar erros no código de produção, esforços têm sido destinados para o desenvolvimento de código de teste com mais qualidade (TUFANO *et al*., 2016). Contudo, a presença de *test smells* pode ter impacto negativo na manutenção do código de teste por gerarem testes complexos e difíceis de entender e de modificar, prejudicando a sua independência e a sua estabilidade (ROMPAEY, 2006). Para investigar as abordagens utilizadas na identificação, na compreensão e na evolução do código de teste em relação à presença de *test smells*, foi realizada uma pesquisa *ad-hoc* na literatura em bases científicas. A visualização de software é utilizada para facilitar a compreensão de sistemas de software e, consequentemente, a realização do processo de manutenção, pois, ao entender o comportamento, as estruturas ou as demais características desses sistemas representadas por técnicas e ferramentas de visualização, tornam menos dispendiosas as manutenções.

Este capítulo está organizado da seguinte forma. Na Seção 3.2, são apresentados os *test smells* estudados neste trabalho. Na Seção 3.3, é descrita a ferramenta `JNose Test`. Na Seção 3.5, são apresentados os conceitos de visualização de software. Na Seção 3.5, são abordados os conceitos de evolução de software.

3.2 *Test Smells*

Test smells são causados por más escolhas de *design* ou de implementação no código de teste (semelhante a *code smells* para o código de produção). Como indicadores de possíveis *test smells*, tem-se a maneira como os casos de teste são documentados ou organizados e como os casos de teste interagem com cada outro caso de teste, com o código de produção e com recursos externos (TUFANO *et al*., 2016). Muitas vezes, *test smells* não são percebidos pelos testadores como problemas reais de *design* (JUNIOR *et al.,* 2020). Além disso, o responsável pela inserção de *test smells* pode não identificar a sua presença em seu próprio código (SPADINI *et al*., 2018). Contudo, no estudo de BAVOTA *et al*., 2012, foi destacado empiricamente o efeito negativo de *test smells* quanto à compreensibilidade e à manutenilibidade do código e ao alto custo, destacando a importância de investir esforços no desenvolvimento de ferramentas para *test smells* (TUFANO *et al*., 2016).

Assim como o código de produção passa por evoluções, o código de teste evolui para atender novos casos de teste. Porém, não se sabe ao certo se a evolução do código de teste é

responsável pela introdução de *test smells* (GREILER *et al*., 2013). Mas, sabe-se que, mesmo que *tests smells* sejam removidos durante a evolução, outros podem ser inseridos (KIM, 2019).

Na literatura, foi introduzido um conjunto inicial de 30 *test smells* (DEURSEN *et al*., 2001; MESZAROS, 2007; PERUMA *et al.*, 2019). Desse conjunto, foram selecionados 21 *test smells* para serem investigados durante a evolução do código de teste apresentada neste trabalho. Esses *test smells* foram escolhidos por terem sido formalmente validados em uma pesquisa com desenvolvedores (PERUMA *et al.*, 2019) e são apoiados pela ferramenta de detecção utilizada neste trabalho, a `JNose Test` (VIRGÍNIO *et al*., 2020). Esses *test smells* são:

a) ***Assertion Roulette*** **(AR)**. Ocorre quando asserções em um método de teste não têm explicação e não é documentado, afetando a legibilidade, a compreensibilidade e a manutenibilidade do código de teste. Se uma das *assertions* falhar, não é trivial identificar qual falhou. O método `assertThat()` é "invocado" 3 vezes (linhas 6, 9 e 10) no método de teste (Código 3.1). Cada declaração verifica uma condição diferente, mas não é fornecida uma mensagem de explicação. Portanto, se uma das *assertions* falhar, a identificação da causa da falha não é direta;

Código 3.1 - Exemplo de *Test Smell Assertion Roulette*

```
1.    @MediumTest
2.    public void testCloneNonBareRepoFromLocalTestServer() throws
Exception {
3.        Clone cloneOp = new Clone(false,
integrationGitServerURIFor("small-repo.early.git"),
helper().newFolder());
4.        Repository repo = executeAndWaitFor(cloneOp);
5.
6.        assertThat(repo,
hasGitObject("ba1f63e4430bff267d112b1e8afc1d6294db0ccc"));
7.
8.        File readmeFile = new File(repo.getWorkTree(), "README");
9.        assertThat(readmeFile, exists());
10.     assertThat(readmeFile, ofLength(12));
11.   }
```

Fonte: Adaptado de Peruma *et al*. (2019).

b) ***Conditional Test Logic*** **(CTL)**. Ocorre quando um método de teste contém expressões condicionais ou estruturas `if`, `switch`, `for` e `while` e as instruções de teste podem não ser executadas porque uma condição não foi atendida. Ou seja, as condições dentro do método de teste são capazes de modificar o comportamento do teste, pois o teste falha quando uma instrução de teste não é executada. O método de teste `testSpinner()` contém várias instruções de controle (linhas 6, 8 e 14), ou seja, o resultado do teste baseado no resultado das condições não é previsível (Código 3.2);

Código 3.2 - Exemplo de *Test Smell Conditional Test Logic*

```
1.    @Test
2.    public void testSpinner() {
3.       for (Map.Entry entry : sourcesMap.entrySet()) {
4.                String id = entry.getKey();
5.          Object resultObject = resultsMap.get(id);
6.          if (resultObject instanceof EventsModel) {
7.                EventsModel result = (EventsModel) resultObject;
8.            if (result.testSpinner.runTest) {
9.                      System.out.println("Testing " + id + "
(testSpinner)");
10.             AnswerObject answer = new
AnswerObject(entry.getValue(), "", new CookieManager(), "");
11.             EventsScraper scraper = new
EventsScraper(RuntimeEnvironment.application, answer);
12.             SpinnerAdapter spinnerAdapter =
scraper.spinnerAdapter();
13.             assertEquals(spinnerAdapter.getCount(),
result.testSpinner.data.size());
14.             for (int i = 0; i < spinnerAdapter.getCount(); i++){
15.               assertEquals(spinnerAdapter.getItem(i),
result.testSpinner.data.get(i));
16.                    }
17.               }
18.            }
19.       }
20.   }
```

Fonte: Adaptado de Peruma *et al*. (2019).

c) ***Constructor Initialization*** **(CI)**. Ocorre quando a classe de teste contém um construtor. A classe utiliza um construtor em vez de um método `setUp()` para atribuir valores iniciais aos campos. No construtor `TagEncodingTest()` (linha 6), os campos `crypto` e `tagkey` recebem valores iniciais (Código 3.3);

Código 3.3 - Exemplo de *Test Smell Constructor Initialization*

```
1.    public class TagEncodingTest extends BrambleTestCase {
2.          private final CryptoComponent crypto;
3.          private final SecretKey tagKey;
4.          private final long streamNumber = 1234567890;
5.
6.          public TagEncodingTest() {
7.                crypto = new CryptoComponentImpl(new
TestSecureRandomProvider());
8.                tagKey = TestUtils.getSecretKey();
9.          }
10.   @Test
11.         public void testKeyAffectsTag() throws Exception {
12.               Set set = new HashSet<>();
13.               for (int i = 0; i < 100; i++) {
14.                     byte[] tag = new byte[TAG_LENGTH];
15.                     SecretKey tagKey = TestUtils.getSecretKey();
16.                     crypto.encodeTag(tag, tagKey,
PROTOCOL_VERSION, streamNumber);
17.                     assertTrue(set.add(new Bytes(tag)));
18.               } }..}
```

Fonte: Adaptado de Peruma *et al*. (2019).

d) ***Default Test* (DT)**. Ocorre quando é criada uma classe de teste padrão pelo *framework* para servir como exemplo (Código 3.4). Essas classes devem ser removidas ou renomeadas, pois a sua permanência fará com que os testadores adicionem métodos de teste a essas classes, tornando-as um contêiner de todos os casos de testes;

Código 3.4 - Exemplo de *Test Smell Default Test*

```
1. public class ExampleUnitTest {
2.     @Test
3.       public void addition_isCorrect() throws Exception {
4.             assertEquals(4, 2 + 2);
5.       }
6.     @Test
7.     public void shareProblem() throws InterruptedException {
8.       .....
9.             Observable.just(200)
10.                  .subscribeOn(Schedulers.newThread())
11.                  .subscribe(begin.asAction());
12.                 begin.set(200);
13.                 Thread.sleep(1000);
14.                 assertEquals(beginTime.get(), "200");
15.           .....
16.           }
17.           .....
18.     }
```

Fonte: Adaptado de Peruma *et al*. (2019).

e) ***Dependent Test* (DepT)**. Ocorre quando o teste em execução depende do sucesso de outros testes. Como resultado, testes dependentes podem falhar pelo motivo errado. No Código 3.5, o método de teste `testEquality()` (linha 14) depende do método `verify()` a ser executado, se a condição desse método for atendida;

Código 3.5 - Exemplo de *Test Smell Dependent Test*

```
1. public class ExampleTest {
2.       @Test
3.       public void verify() {
4.            int x = 20;
5.            int y = 50;
6.
7.            if(x<y){
8.              testEquality();
9.            }else{
10.             System.out.println("False");
11.          }
12.     }
13.     @Test
14.     public void testEquality(){
15.           String expected = "Record sucessfully saved!";
16.           String obtained = "Record sucessfully saved!";
17.
18.           assertEquals(expected, obtained)
19.         }
20.     }
```

Fonte: Do autor (2021).

f) ***Duplicate Assert* (DA)**. Ocorre quando um método testa a mesma condição várias vezes. Nesse caso, se o método precisar testar a mesma condição usando valores diferentes, um novo método de teste deve ser utilizado e o nome do método deve ser uma indicação do teste em execução. As possíveis situações que poderiam dar origem a esse *test smell* seriam: i) agrupar várias condições para testar um único método; ii) atividades de depuração; e iii) uma cópia acidental de código que testa a mesma condição várias vezes em um único método de teste. O método `assertEquals()` testa as condições "`Frütz büx is invalid`", "`Spaces are valid`", "`Exclamation mark is valid`" e "`Minus is valid`" diversas vezes (Código 3.6);

Código 3.6 - Exemplo de *Test Smell Duplicate Assert*

```
1.    @Test
2.
3.    public void testXmlSanitizer() {
4.
5.        boolean valid = XmlSanitizer.isValid("Fritzbox");
6.        assertEquals("Fritzbox is valid", true, valid);
7.        System.out.println("Pure ASCII test - passed");
8.
9.        valid = XmlSanitizer.isValid("Fritz Box");
10.       assertEquals("Spaces are valid", true, valid);
11.       System.out.println("Spaces test - passed");
12.
13.       valid = XmlSanitizer.isValid("Frützbüx");
14.       assertEquals("Frützbüx is invalid", false, valid);
15.       System.out.println("No ASCII test - passed");
16.
17.       valid = XmlSanitizer.isValid("Fritz!box");
18.       assertEquals("Exclamation mark is valid", true, valid);
19.       System.out.println("Exclamation mark test - passed");
20.
21.       valid = XmlSanitizer.isValid("Fritz.box");
22.       assertEquals("Exclamation mark is valid", true, valid);
23.       System.out.println("Dot test - passed");
24.
25.       valid = XmlSanitizer.isValid("Fritz-box");
26.       assertEquals("Minus is valid", true, valid);
27.       System.out.println("Minus test - passed");
28.
29.       valid = XmlSanitizer.isValid("Fritz-box");
30.       assertEquals("Minus is valid", true, valid);
31.       System.out.println("Minus test - passed");
32.   }
```

Fonte: Adaptado de Peruma *et al*. (2019).

g) ***Eager Test* (ET)**. Ocorre quando um método de teste "invoca" vários métodos do código de produção. O teste verifica muitas funções em um único método, dificultando a compreensão e a manutenibilidade do código de teste. No Código 3.7, são apresentados três testes (linhas 7, 9 e 11) no mesmo método de teste;

Código 3.7 - Exemplo de *Test Smell Eager Test*

```
1.    @Test
2.
3.    public void NmeaSentence_GPGSA_ReadValidValues(){
4.
5.        NmeaSentence nmeaSentence = new
NmeaSentence("$GPGSA,A,3,04,05,,09,12,,,24,,,,,2.5,1.3,2.1*39");
6.
7.        assertThat("GPGSA - read PDOP",
nmeaSentence.getLatestPdop(), is("2.5"));
8.
9.        assertThat("GPGSA - read HDOP",
nmeaSentence.getLatestHdop(), is("1.3"));
10.
11.       assertThat("GPGSA - read VDOP",
nmeaSentence.getLatestVdop(), is("2.1"));
12.   }
```

Fonte: Adaptado de Peruma *et al*. (2019).

h) ***Empty Test*** **(EpT)**. Ocorre quando um método de teste não contém instruções executáveis. Esses métodos são possivelmente criados para fins de depuração e esquecidos ou contêm código comentado (Código 3.8 - linhas 3, 4 e 5). Um teste vazio pode ser considerado mais problemático do que não ter um caso de teste, uma vez que o JUnit indicará que o teste passa mesmo se não houver instruções executáveis presentes no corpo do método. Assim, os desenvolvedores que introduzirem mudanças de comportamento na classe de produção não serão notificados dos resultados alternativos, pois JUnit relatará o teste como aprovado;

Código 3.8 - Exemplo de *Test Smell Empty Test*

```
1.    public void testCredGetFullSampleV1() throws Throwable{
2.
3.    //        ScrapedCredentials credentials =
innerCredTest(FULL_SAMPLE_v1);
4.    //        assertEquals("p4ssw0rd", credentials.pass);
5.    //        assertEquals("user@example.com",credentials.user);
6.    }
```

Fonte: Adaptado de Peruma *et al*. (2019).

i) ***Exception Catching Throwing*** **(ECT)**. Ocorre quando um método de teste é explicitamente dependente do método de produção que gera uma exceção. O correto é utilizar o tratamento de exceções de JUnit para aprovar/falhar automaticamente no teste, ao invés de lançar uma exceção. Na linha 16 do Código 3.9, há falha no teste quando ocorre uma exceção específica. O ideal é dividir esse método de teste em vários métodos que (1) geram conscientemente a exceção e (2) não geram a exceção utilizando o atributo esperado `@Test` no JUnit para falhar o teste quando a exceção ocorre, em vez de capturar ou lançar explicitamente a exceção;

Código 3.9 - Exemplo de *Test Smell Exception Catching Throwing*

```
1.    @Test
2.
3.    public void realCase() {
4.
5.        Point p34 = new Point("34", 556506.667, 172513.91, 620.34,
true);
6.        Point p45 = new Point("45", 556495.16, 172493.912, 623.37,
true);
7.        Point p47 = new Point("47", 556612.21, 172489.274, 0.0,
true);
8.        Abriss a = new Abriss(p34, false);
9.        a.removeDAO(CalculationsDataSource.getInstance());
10.       a.getMeasures().add(new Measure(p45, 0.0, 91.6892, 23.277,
1.63));
11.       a.getMeasures().add(new Measure(p47, 281.3521, 100.0471,
108.384, 1.63));
12.
13.       try {
14.           a.compute();
15.       } catch (CalculationException e) {
16.           Assert.fail(e.getMessage());
17.       }
18.
19.       // test intermediate values with point 45
20.       Assert.assertEquals("233.2405",
21.
this.df4.format(a.getResults().get(0).getUnknownOrientation()));
22.       Assert.assertEquals("233.2435",
23.
this.df4.format(a.getResults().get(0).getOrientedDirection()));
24.       Assert.assertEquals("-0.1", this.df1.format(
25.           a.getResults().get(0).getErrTrans()));
26.
27.       // test intermediate values with point 47
28.       Assert.assertEquals("233.2466",
29.
this.df4.format(a.getResults().get(1).getUnknownOrientation()));
30.       Assert.assertEquals("114.5956",
31.
this.df4.format(a.getResults().get(1).getOrientedDirection()));
32.       Assert.assertEquals("0.5", this.df1.format(
33.           a.getResults().get(1).getErrTrans()));
34.
35.       // test final results
36.       Assert.assertEquals("233.2435",
this.df4.format(a.getMean()));
37.
38.       Assert.assertEquals("43", this.df0.format(a.getMSE()));
39.
40.       Assert.assertEquals("30",
this.df0.format(a.getMeanErrComp()));
41.   }
```

Fonte: Adaptado de Peruma *et al*. (2019).

j) ***General Fixture*** **(GF)**. Ocorre quando o teste é muito geral e os métodos de teste acessam apenas parte dele. Um método de configuração de teste que inicializa campos não acessados pelos métodos gera trabalho desnecessário, tornando o teste mais lento. Por

exemplo, as seis variavéis do método `setUp()` (linhas 3 a 8) (Código 3.10) são inicializadas, porém, no método de teste `testIsCA()` (linha 11), é utilizado o valor de apenas quatro variavéis;

Código 3.10 - Exemplo de *Test Smell General Fixture*

```
1.    protected void setUp() throws Exception {
2.
3.        assetManager =
getInstrumentation().getContext().getAssets();
4.        certificateFactory =
CertificateFactory.getInstance("X.509");
5.        infoDebianTestCA = loadCertificateInfo("DebianTestCA.pem");
6.        infoDebianTestNoCA =
loadCertificateInfo("DebianTestNoCA.pem");
7.        infoGTECyberTrust =
loadCertificateInfo("GTECyberTrustGlobalRoot.pem");
8.             infoMehlMX = loadCertificateInfo("mehl.mx.pem");
9.    }
10.
11.   public void testIsCA() {
12.
13.       assertTrue(infoDebianTestCA.isCA());
14.       assertFalse(infoDebianTestNoCA.isCA());
15.       assertNull(infoGTECyberTrust.isCA());
16.       assertFalse(infoMehlMX.isCA());
17.   }
```

Fonte: Adaptado de Peruma *et al*. (2019).

k) ***Ignored Test*** **(IgT)**. Ocorre quando um método de teste é suprimido da execução. JUnit fornece a capacidade de impedir a execução de métodos de teste. No entanto, esses métodos ignorados resultam em sobrecarga desnecessária em relação ao tempo de compilação e aumentam a complexidade e a compreensão do código. São os métodos que contém a anotação `@Ignore` (Código 3.11 - linha 1);

Código 3.11 - Exemplo de *Test Smell Ignored Test*

```
1.    @Ignore("disabled for now as this test is too flaky")
2.    public void peerPriority() throws Exception {
3.
4.          final List addresses = Lists.newArrayList(
5.          new InetSocketAddress("localhost", 2000),
6.                new InetSocketAddress("localhost", 2001),
7.                new InetSocketAddress("localhost", 2002)
8.          );
9.        peerGroup.addConnectedEventListener(connectedListener);
10.   }
```

Fonte: Adaptado de Peruma *et al*. (2019).

l) ***Lazy Test*** **(LT)**. Ocorre quando vários métodos de teste verificam o mesmo método de produção. Os métodos `testDecrypt()` e `testEncrypt()` (linhas 2 e 16

respectivamente) “invocam” o mesmo método `Cryptographer.decrypt()` (linha 12 e linha 19, respectivamente) (Código 3.12);

Código 3.12 - Exemplo de *Test Smell Lazy Test*

```
1.    @Test
2.    public void testDecrypt() throws Exception {
3.        FileInputStream file = new
FileInputStream(ENCRYPTED_DATA_FILE_4_14);
4.        byte[] enfileData = new byte[file.available()];
5.        FileInputStream input = new
FileInputStream(DECRYPTED_DATA_FILE_4_14);
6.        byte[] fileData = new byte[input.available()];
7.        input.read(fileData);
8.        input.close();
9.        file.read(enfileData);
10.       file.close();
11.       String expectedResult = new String(fileData, "UTF-8");
12.       assertEquals("Testing simple decrypt",expectedResult,
Cryptographer.decrypt(enfileData, "test"));
13.   }
14.
15.   @Test
16.   public void testEncrypt() throws Exception {
17.       String xml = readFileAsString(DECRYPTED_DATA_FILE_4_14);
18.       byte[] encrypted = Cryptographer.encrypt(xml, "test");
19.       String decrypt = Cryptographer.decrypt(encrypted, "test");
20.       assertEquals(xml, decrypt);
21.   }
```

Fonte: Adaptado de Peruma *et al*. (2019).

m) ***Magic Number Test*** **(MNT)**. Ocorre quando instruções contêm literais numéricos (“números mágicos”) como parâmetros. O correto é utilizar variáveis ou constantes, fornecendo um nome descritivo para a entrada. Não se sabe os significados dos literais numéricos 15 e 30 nas linhas 4 e 5, respectivamente, passados como parâmetro (Código 3.13);

Código 3.13 - Exemplo de *Test Smell Magic Number Test*

```
1.    @Test
2.    public void testGetLocalTimeAsCalendar() {
3.        Calendar localTime =
calc.getLocalTimeAsCalendar(BigDecimal.valueOf(15.5D),
Calendar.getInstance());
4.        assertEquals(15, localTime.get(Calendar.HOUR_OF_DAY));
5.        assertEquals(30, localTime.get(Calendar.MINUTE));
6.    }
```

Fonte: Adaptado de Peruma *et al*. (2019).

n) ***Mystery Guest*** **(MG)**. Ocorre quando um método de teste usa recursos externos (por exemplo, arquivos e banco de dados) e, portanto, não é independente. Logo, se esse recurso for alterado, os testes podem falhar. Consequentemente, não há informações suficientes para entender a funcionalidade testada. No método de teste `testPersistence()`, é

criado um Arquivo (`tempFile`) em um diretório específico (linha 2) e, em seguida, esse arquivo é utilizado no processo de teste (linha 6) (Código 3.14);

Código 3.14 - Exemplo de *Test Smell Mystery Guest*

```
1.    public void testPersistence() throws Exception {
2.        File tempFile = File.createTempFile("systemstate-",
".txt");
3.        try {
4.            SystemState a = new SystemState(then, 27, false,
bootTimestamp);
5.            a.addInstalledApp("a.b.c", "ABC", "1.2.3");
6.            a.writeToFile(tempFile);
7.            SystemState b = SystemState.readFromFile(tempFile);
8.            assertEquals(a, b);
9.        } finally {
10.           //noinspection ConstantConditions
11.           if (tempFile != null) {
12.               //noinspection ResultOfMethodCallIgnored
13.               tempFile.delete();
14.           }
15.       }
16.   }
```

Fonte: Adaptado de Peruma *et al*. (2019).

o) ***Print Statement* (PS)**. Ocorre quando os métodos de testes contêm "impressão" de dados, normalmente usada para depuração e rastreabilidade, e pode ser esquecida no código. O método "invoca" funções como `print()`, `println()`, `printf()` ou `write()`. O método `testTransform10mNEUAndBack()` contém uma instrução que imprime o valor de uma variável (`result`) no console (linha 5) (Código 3.15);

Código 3.15 - Exemplo de *Test Smell Print Statement*

```
1.    @Test
2.    public void testTransform10mNEUAndBack() {
3.        Leg northEastAndUp10M = new Leg(10, 45, 45);
4.        Coord3D   result   =   transformer.transform(Coord3D.ORIGIN,
northEastAndUp10M);
5.        System.out.println("result = " + result);
6.        Leg reverse = new Leg(10, 225, -45);
7.        result = transformer.transform(result, reverse);
8.        assertEquals(Coord3D.ORIGIN, result);
9.    }
```

Fonte: Adaptado de Peruma *et al*. (2019).

p) ***Redundant Assertion* (RA)**. Ocorre quando os métodos de teste contêm instruções que sempre são verdadeiras ou sempre são falsas. Os métodos podem ter sido inseridos para fins de depuração e esquecidos no código. O método de teste `testTrue()` sempre será verdadeiro, pois a instrução `assertEquals()` compara um valor booleano `true` com outro valor booleano `true` (Código 3.16);

Código 3.16 - Exemplo de *Test Smell Redundant Assertion*

```
1.    @Test
2.    public void testTrue() {
3.        assertEquals(true, true);
4.    }
```

Fonte: Adaptado de Peruma *et al*. (2019).

q) ***Resource Optimism* (RO)**. Refere-se a um método de teste que faz suposições otimistas dos recursos externos utilizados, por exemplo, não verifica se um arquivo existe. O método de teste `save.Image_noImageFile_ko()` acessa o arquivo (linha 6) sem verificar se ele existe antes de usá-lo nas operações do teste (Código 3.17);

Código 3.17 - Exemplo de *Test Smell Resource Optimism*

```
1.    @Test
2.    public void saveImage_noImageFile_ko() throws IOException {
3.
4.        File outputFile = File.createTempFile("prefix", "png", new
File("/tmp"));
5.
6.        ProductImage image = new ProductImage("01010101010101",
ProductImageField.FRONT, outputFile);
7.
8.        Response response = serviceWrite.saveImage(image.getCode(),
image.getField(), image.getImguploadFront(),
image.getImguploadIngredients(),
image.getImguploadNutrition()).execute();
9.
10.       assertTrue(response.isSuccess());
11.
12.       assertThatJson(response.body())
13.       .node("status")
14.       .isEqualTo("status not ok");
15.   ...
16.   ...
17.   ...
18.   ...
19.   }
```

Fonte: Adaptado de Peruma *et al*. (2019).

r) ***Sensitive Equality* (SE)**. Ocorre quando o método `toString()` é utilizado em métodos de teste para verificar objetos, sendo sua saída comparada a uma sequência específica. Alterações na implementação do método `toString()` podem resultar em falha. A abordagem correta pode ser a implementação de um método personalizado para executar essa comparação. Na linha 11 do Código 3.18, o valor retornado pelo método `toString()` é comparado a *string* "`/54.204.10.41`", se houver alterações na implementação dos objetos do método `toString()`, pode ocorrer falha;

Código 3.18 - Exemplo de *Test Smell Sensitive Equality*

```
1.    @Test
2.    public void test1() throws UnknownHostException {
3.        String peersPacket = "F8 4E 11 F8 4B C5 36 81 " +
4.            "CC 0A 29 82 76 5F B8 40 D8 D6 0C 25 80 FA 79 5C " +
5.            "FC 03 13 EF DE BA 86 9D 21 94 E7 9E 7C B2 B5 22 " +
6.            "F7 82 FF A0 39 2C BB AB 8D 1B AC 30 12 08 B1 37 " +
7.            "E0 DE 49 98 33 4F 3B CF 73 FA 11 7E F2 13 F8 74 " +
8.            "17 08 9F EA F8 4C 21 B0";
9.         byte[] payload = Hex.decode(peersPacket);
10.       byte[] ip = decodeIP4Bytes(payload, 5);
11.       assertEquals(InetAddress.getByAddress(ip).toString(),
("/54.204.10.41"));
12.   }
```

Fonte: Adaptado de Peruma *et al*. (2019).

s) ***Sleepy Test* (ST)**. Ocorre quando a execução de instruções em um método de teste precisa ser pausada por um período de tempo (por exemplo, simulando um evento externo) e depois, continua com a execução. Um atraso artificial é causado na execução do teste usando o método `Thread.sleep()` (linha 11) (Código 3.19);

Código 3.19 - Exemplo de *Test Smell Sleepy Test*

```
1.    public void testEdictExternSearch() throws Exception {
2.        final Intent i = new
Intent(getInstrumentation().getContext(), ResultActivity.class);
3.        i.setAction(ResultActivity.EDICT_ACTION_INTERCEPT);
4.        i.putExtra(ResultActivity.EDICT_INTENTKEY_KANJIS, "空白");
5.          tester.startActivity(i);
6.          assertTrue(tester.getText(R.id.textSelectedDictionary).co
ntains("Default"));
7.          final ListView lv = getActivity().getListView();
8.          assertEquals(1, lv.getCount());
9.          DictEntry entry = (DictEntry) lv.getItemAtPosition(0);
10.         assertEquals("Searching", entry.english);
11.         Thread.sleep(500);
12.         final Intent i2 = getStartedActivityIntent();
13.         final List result = (List)
i2.getSerializableExtra(ResultActivity.INTENTKEY_RESULT_LIST);
14.         entry = result.get(0);
15.         assertEquals("(adj-na,n,adj-no) blank
space/vacuum/space/null (NUL)/(P)", entry.english);
16.         assertEquals("空白", entry.getJapanese());
17.         assertEquals("くうはく", entry.reading);
18.         assertEquals(1, result.size());
19.   }
```

Fonte: Adaptado de Peruma *et al*. (2019).

t) ***Unknown Test* (UT)**. Ocorre quando um método de teste usa uma afirmação para uma condição esperada, o que torna difícil entender o objetivo do método de teste. No Código 3.20, falta o método de verificação, não sendo possível saber o que é testado;

Código 3.20 - Exemplo de *Test Smell Unknown Test*

```
1.    @Test
2.
3.    public void hitGetPOICategoriesApi() throws Exception {
4.
5.        POICategories poiCategories =
apiClient.getPOICategories(16);
6.
7.        for (POICategory category : poiCategories) {
8.          System.out.println(category.name() + ": " + category);
9.
10.       }
11.   }
```

Fonte: Adaptado de Peruma *et al*. (2019).

u) ***Verbose Test*** **(VT)**. Ocorre quando os testes possuem linhas de código excessivas, o código de teste não é limpo e simples, o que torna o teste difícil de entender. No Código 3.21, é apresentado o método de teste `verify()` com instruções condicionais (`if`) aninhadas (linhas 3, 4 e 5).

Código 3.21 - Exemplo de *Test Smell Verbose Test*

```
1.    @Test
2.    public void verify() {
3.        if("insert".equals(action)){
4.          if("person".equals(type)){
5.                if(identification.charArt(9) - '0' !=a[9]){
6.                      return false;
7.                }else{
8.                      this.name = name;
9.                      this.cpf = cpf;
10.                     this.personphysical = true;
11.               }
12.   ...
13.   ...
14.   ...
15.   ...
16.   ...
17.   ...
18.     }
19.    }
20.   }
```

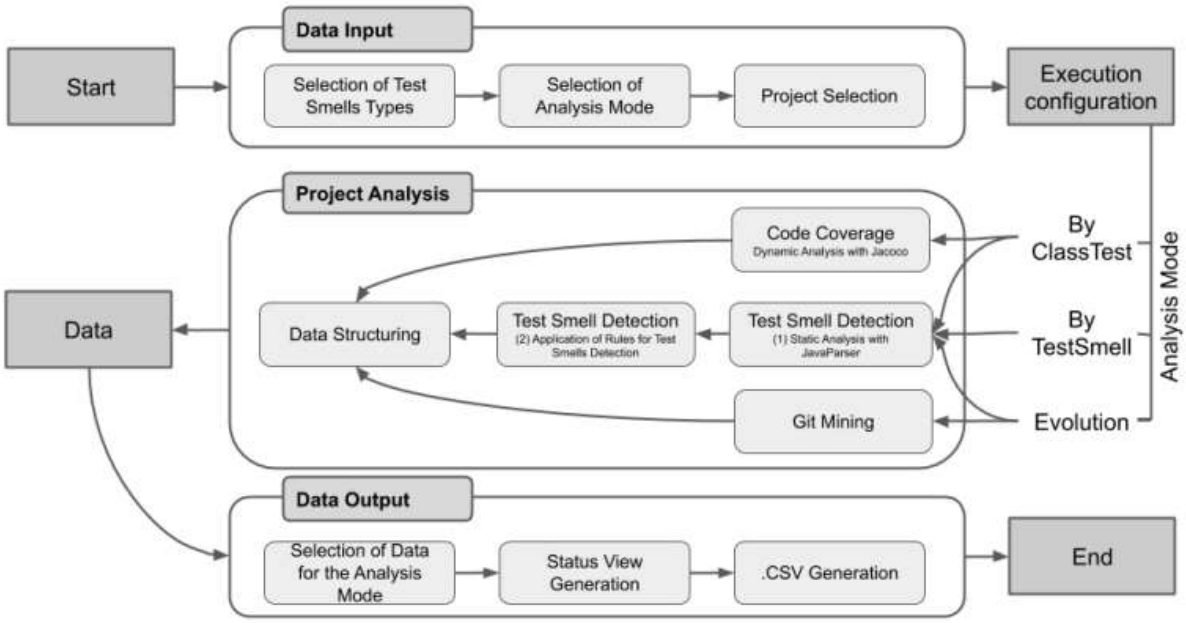
Fonte: Do autor (2021).

3.3 `JNose Test`

A ferramenta `JNose Test` apoia a análise da qualidade de casos de teste, na perspectiva de *test smells*, por meio da detecção desses *test smells* e de cálculos de cobertura de código. A `JNose Test` envolve três processos principais (Figura 3.1) (VIRGÍNIO *et al*. 2020): i) ***Data Input***, recebe as configurações para a execução da ferramenta, ou seja, a lista de *test smells*, o modo de análise (cobertura de código, *test smells* e evolução) e o projeto a ser

analisado; ii) ***Project Analysis***, realiza a análise do projeto de acordo com o modo de análise selecionado; e iii) ***Data Output***, apresenta o *status* de execução e gera um arquivo `.csv` com os resultados da análise.

Figura 3.1 - Visão Geral da Ferramenta `JNose Test` e seus principais recursos



Fonte: Virgínio *et al*. (2020).

`JNose Test` é uma ferramenta que detecta automaticamente 21 *test smells* em sistemas de software Java cuja classes de teste tenham sido geradas pelo *framework* JUnit. Além disso, coleta medidas de cobertura de teste, apresentando a quantidade de *test smells* detectados e as medidas de cobertura por classes, por meio de quatro configurações:

a) ***by TestClass.*** Detecta a quantidade de cada *test smell* nas classes de teste;
b) ***by TestSmells.*** Indica a localização (classe e método de teste) de cada *test smell*;
c) ***by TestFile.*** Detecta os *test smells* na classe selecionada e exibe a explicação do *test smells* e o código onde ocorre;
d) ***Evolution***. Coleta dados sobre versões de software (por exemplo, autores) e *test smells*.

Para o contexto deste trabalho, são explicadas as configurações: b) ***by TestSmells*** e d) ***Evolution***, todas as configurações são explicadas no trabalho de (VIRGÍNIO *et al*. 2020). Inicialmente, acessar `http://jnose.herokuapp.com/`. Na aba *Projects*, informar o caminho do projeto disponível no *GitHub* e realizar a clonagem do projeto, clicando no botão *Clone* (Figura 3.2).

Figura 3.2 - Execução da `JNose Test`

Fonte: `http://jnose.herokuapp.com/` (2021).

Deve-se selecionar a opção *Evolution* e o filtro *Commits* (Figura 3.3). Clicar em *Analyze* e, após a finalizada a execução, selecionar o primeiro relatório gerado (Figura 3.3 - opção 1 - *Evolution Report 1 - TestSmells by Commit and Class*) e clicar em *Export CSV*, será feito o *download* do arquivo de *test smells* (*resultado_evolution1.csv*). Essa análise fornece a detecção de *test smells* por classe de teste para cada versão do projeto, além de dados sobre o autor responsável pelo *test smell*. As linhas do arquivo `.csv` representam as classes de teste e as colunas os seguintes parâmetros: projeto, classe de teste, classe de produção, quantidade e nome dos *test smells*, identificação do *commit*, autoria, data e mensagem do *commit*. Para saber quem é responsável pelos *test smells*, a ferramenta calcula automaticamente a autoria de um *test smell* por culpa, ou seja, atribui a autoria para o testador que modificou o método pela última vez (e não o corrigiu), supondo que ele estava ciente da presença do *test smell*.

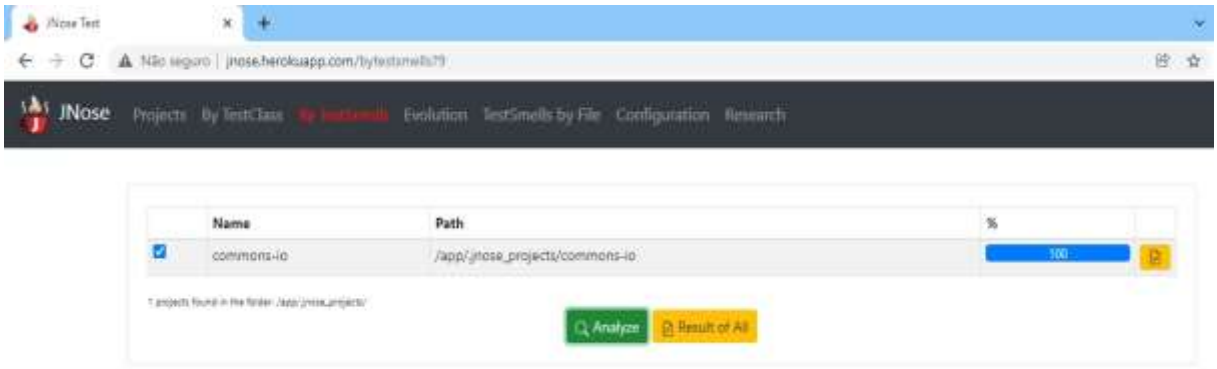
Figura 3.3 - `JNose Test` opção *Evolution*

Fonte: `http://jnose.herokuapp.com/` (2021).

Para a análise por métodos, utiliza-se a opção *by TestSmells* (*nomedoprojeto_result_byclasstest_testsmells.csv*). Após clicar em *Analyze*, deve-se clicar no

relatório gerado (Figura 3.4). Essa análise fornece o método exato onde o *test smell* está localizado, cada linha do `.csv` representa um *test smell* e as colunas representam os seguintes parâmetros: projeto, classe de teste, classe de produção, *test smell*, linha de localização do *test smell* e o método.

Figura 3.4 - `JNose` Opção *By Test Smells*



Fonte: `http://jnose.herokuapp.com/` (2021).

`JNose Test` estende a ferramenta `tsDetect` que também identifica 21 *test smells*, mas retorna um valor booleano para indicar se um *test smell* ocorre no código de teste (BAVOTA *et al.*, 2015). `JNose Test` reutilizou as regras de detecção de `tsDetect`, pois, apresenta uma precisão variando de 85% a 100% e um *recall* de 90% a 100% (PERUMA *et al.*, 2019). No entanto, estende vários aspectos, fornecendo a quantidade de ocorrências de cada test *smell*, identificando a classe de teste, o nome e a linha de método onde ocorre o *test smell*. Além disso, apoia a análise do conjunto de testes de várias versões do projeto, minerando o Git para fornecer informações sobre quando e quem são os autores responsáveis pelos *test smells* (VIRGÍNIO *et al.*, 2020).

3.4 Visualização de Software

A visualização de software é utilizada para facilitar a compreensão de sistemas de software. Consequentemente, o processo de manutenção pode se tornar menos árduo, pois, o entendimento do comportamento, das estruturas e/ou das demais características de sistemas de software pode ser facilitado por meio da representação visual.

Visualização é o processo de transformar informações em uma forma visual, permitindo aos usuários observar a informação. A visualização de software tem sido utilizada na Engenharia de Software para facilitar a tarefa de compreensão, de manutenção e de evolução de sistemas de software (DIEHL, 2007; CONCEIÇÃO *et al.*, 2012). Ao utilizar recursos

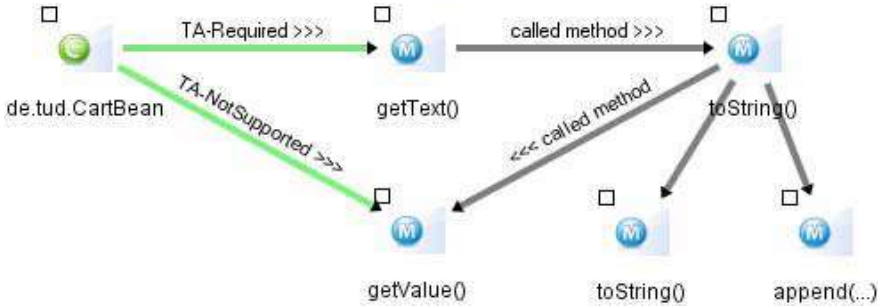
gráficos, a compreensão do sistema de software pode ser melhorada, pois torna-se mais intuitiva do que representações textuais (CASERTA; ZENDRA, 2011). Por isso, muitos pesquisadores têm apostado no valor da visualização de software para os domínios da Engenharia de Software, uma vez que, tipicamente nessa área, há grandes e complexas quantidades de dados (KOSCHE; 2003).

A visualização de software é uma área que tem sido utilizada como alternativa para reduzir a dificuldade de compreensão (NOVAIS *et al*., 2013). Essa área utiliza recursos visuais para representar a estrutura, o comportamento e a evolução de sistemas de software (CARPENDALE; GHANAM, 2008). Ao representar sistemas de software por meio de recursos visuais, sua representação mental pode ser clara, porque os seres humanos possuem mais facilidade em entender informações representadas de forma gráfica (CASERTA; ZENDRA, 2011). Existem diversas técnicas de visualização; em um estudo, 20 técnicas de visualização de software foram identificadas utilizando revisão sistemática da literatura (CRUZ *et al*., 2016). A seguir, são explicadas as três técnicas utilizadas neste trabalho:

a) ***Graphs***. São utilizados para representar sistemas de software orientados a objetos. A sua representação varia de acordo com a ferramenta em que for implementada (SULAIMAN, 2004). Mas, no geral, os grafos são compostos de arestas e vértices, sendo as arestas a ligação entre dois vértices. Os grafos são estruturas computacionais de reconhecida utilidade em Ciência da Computação pela sua capacidade de representar os relacionamentos entre componentes do sistema de software (LOZADA, 2014). Por exemplo, a ferramenta SEXTANT utiliza *graphs* para exploração do sistema de software, onde os nós do grafo representam os artefatos do software e as arestas representam o relacionamento entre os nós. Na Figura 3.5, a ferramenta está analisando um sistema de software, porque foi relatado um *bug* por causa do componente `CartBean`. A partir da visualização, é possível obter rapidamente quais classes implementam o *bean* e chegar ao resultado: classe `de.tud.CartBean` (EICHBERG *et al*., 2005). Para o contexto deste trabalho, a técnica é denominada *Graph View*;
b) ***Treemap***. É uma técnica comumente utilizada para fornecer uma visão geral da hierarquia (pacotes, classes e métodos), utilizando retângulos aninhados. Essa visualização é gerada recursivamente ao subdividir um retângulo em retângulos menores para cada nível da hierarquia (CASERTA; ZENDRA, 2011; LÜ; FOGARTY, 2008; STOREY *et al.*, 2002). A representação do código como retângulos aninhados permite ao leitor obter uma visão

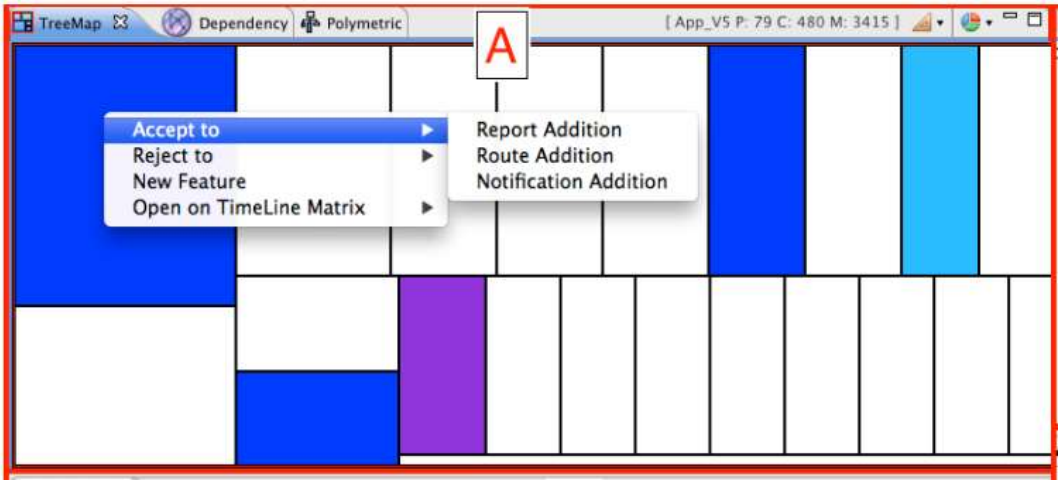
da estrutura do código (PORTO *et al.*, 2009). A ferramenta `SourceMiner`[8] (Figura 3.6) mostra uma vista panorâmica da estrutura estática do sistema de software. As cores, os tamanhos e os retângulos são associados à complexidade e ao tamanho do módulo representado. Essa visualização permite o reconhecimento de padrões da heurística adotada pelos programadores durante a execução de tarefas de manutenção (CARNEIRO *et al.*, 2009). Para o contexto deste trabalho, a técnica é denominada *Treemap View*;

Figura 3.5 - Explorando o rastreamento de erros com SEXTANT



Fonte: Eichberg *et al*. (2005).

Figura 3.6 - Visão *Source Miner*



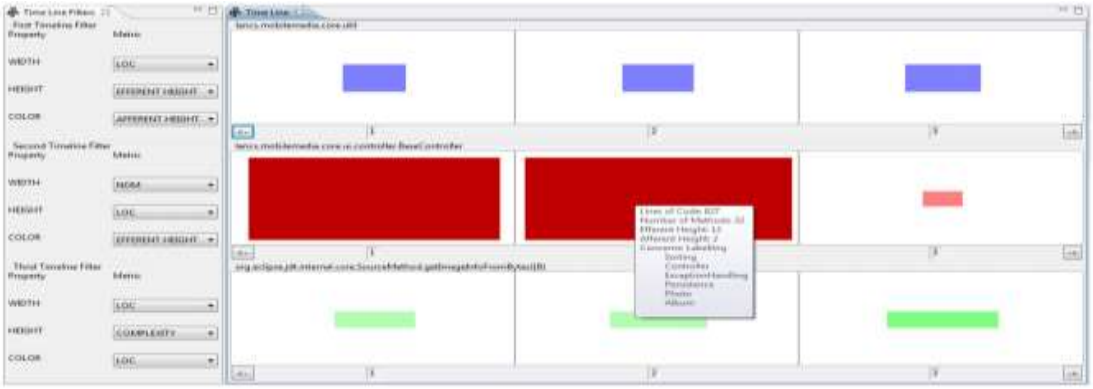
Fonte: Novais *et al*. (2013).

c) ***Timeline***. É uma técnica utilizada para representar a evolução. Por exemplo, ela foi utilizada para exibir a evolução das classes baseada no conceito de edifícios (WETTEL; LANZA, 2008). Os edifícios correspondem as classes e os tijolos correspondem aos métodos dessas classes. Cada método permanece no mesmo lugar nas versões da classe. A cor é utilizada para representar a quantidade de versões (CASERTA; ZENDRA, 2011). A *Timeline* pode ser representada na forma de uma matriz (LANZA, 2007). A ferramenta `TimeLine Matrix (TLM)` (Figura 3.7) foi projetada no formato de uma matriz 3x3,

[8] http://wiki.dcc.ufba.br/SoftVis/SME?redirectedfrom=SoftVis.Sme

em que as linhas são denominadas de *timeline* de evolução e as colunas representam uma versão do elemento de software em análise. Cada *timeline* mostra até três versões do elemento. É possível selecionar para cada *timeline* um elemento de software diferente ou o mesmo elemento de software para todas as *timelines* (NOVAIS *et al*., 2012). Para o contexto deste trabalho, a técnica será denominada *Timeline View*.

Figura 3.7 - Visão *Timeline Matrix*



Fonte: Novais *et al*., 2012

3.5 Evolução de Sistemas de Software

Um sistema de software é mantido por tempo indeterminado, ou seja, evoluções e manutenções devem ser realizadas. No processo de realizar manutenções, diferentes tipos de alterações são feitas, seja para atender novos requisitos, para corrigir *bugs* ou *design* ou para melhorar o desempenho (SULAIMAN, 2004). Como esses sistemas evoluem ao longo do tempo, assim como o código de produção, o código de teste também evolui. A evolução é fundamental para os sistemas de software atenderem as novas necessidades de seus usuários, como também para corrigir problemas no código e refatorá-lo. A compreensão dessa evolução pode ser utilizada para encontrar a origem de problemas atuais ou para obter informações que possibilitam prever características futuras desses sistemas. No entanto, o aumento das informações, das funções e da quantidade de código deixa o sistema de software mais complexo, dificultando sua compreensão. Desse modo, técnicas de visualização de software têm sido utilizadas para representar a evolução de determinados atributos do software, facilitando a compreensão de suas características evolutivas (BASTOS; COSTA, 2016).

As estratégias de análise para a visualização da evolução de um sistema de software dividem-se em 2 tipos principais (*Differential e Temporal*); e 5 subtipos (*Differential*: *Differential Relative* (DR) e *Differential Absolute* (DA); e *Temporal*: *Temporal Overview* (TO), *Temporal Snapshot* (TS) e *Temporal Accumulative Snapshot* (TA)) (NOVAIS *et al*., 2013).

Para a representação visual da evolução dos *test smells* neste trabalho, uma das técnicas utilizadas é a técnica *timeline*, pois ela segue a estratégia **Temporal** por retratar a evolução considerando várias versões disponíveis para análise. Dadas `n` versões (v_1, v_2, ... , v_n) ou um subconjunto sequencial considerável delas, essa análise leva em conta o que aconteceu da versão v_1 para a versão v_n, considerando as versões intermediárias. Isso ajuda, por exemplo, a analisar padrões de mudanças na evolução do software. Essa estratégia pode ser especializada em uma visão geral da evolução ou em um instantâneo (NOVAIS *et al*., 2013).

De acordo com NOVAIS *et al*., 2013, dentro do contexto de evolução de software, há também a divisão em perspectivas para a representação de sistemas de software. As perspectivas mais comuns são Estruturais, Dependência, Mudança, Autoria e Perspectivas de herança. A perspectiva Autoria traz à tona as atividades dos autores durante a evolução de sistemas de software, por exemplo, quantos *commits* eles realizaram ou em quais arquivos eles trabalharam. Responde questões como "Quem realizou essas modificações no código?". Neste trabalho, foi utilizada a perspectiva **Autoria** adaptada para a técnica de visualização de *graphs* para representar quem possivelmente é autor dos *test smells*.

3.6 Considerações Finais

Neste capítulo, foi apresentada uma visão geral sobre *test smells*, 21 test smells foram exemplificados, ferramenta `JNose Test`, visualização de software e evolução de software. Estudos comprovam que *test smells* afetam negativamente os testes, por exemplo, podendo torná-los mais lentos, como no *test smell Lazy Test*, em que vários métodos de teste verificam o mesmo método de produção.

Como sistemas de software estão em constante evolução, é necessário identificar de forma mais rápida problemas como *test smells*. Além disso, quanto mais fácil se dá a compreensão do sistema de software, menos árduas podem ser as manutenções, pois os envolvidos gastarão menos tempo em entender o sistema de software. Assim, como o código de produção evolui, em paralelo, o código de teste também precisam evoluir. Por isso, é importante utilizar técnicas que facilitem a visualização da evolução do software facilitando a identificação e a visualização de problemas como más escolhas de *design*.

4 ABORDAGEM PROPOSTA

4.1 Considerações Iniciais

A abordagem proposta neste trabalho visa estabelecer a relação entre *test smells*, visualização de software e evolução de software. Para isso, foram definidas estratégias de análise visual para *test smells*. Essas estratégias definem a forma como os dados são analisados e exibidos para serem utilizadas pelas técnicas de visualização. O propósito dessa abordagem é fornecer características de ocorrências e evolução de *test smells* de forma visual.

Além disso, uma ferramenta, `TSVizzEvolution`[9], foi desenvolvida para automatizar a abordagem proposta. Essa ferramenta facilita a visualização da ocorrência de *test smells* durante a evolução do código de teste e de quem assume a autoria pelo *test smells*.

Este capítulo está organizado da seguinte forma. Na Seção 4.2, é apresentada a visão geral da abordagem. Na Seção 4.3, o processo de desenvolvimento da ferramenta é explicado. Na Seção 4.4, uma visão geral da ferramenta é fornecida. Na Seção 4.5, o funcionamento da ferramenta é explicado e exemplificado.

4.2 Visão Geral da Abordagem

Estratégias visuais são utilizadas em diversos conceitos, tais como, *code smells*, evolução de software, código morto, mas não foram encontradas estratégias específicas para a representação visual de *test smells*. Na abordagem proposta, foram definidas três estratégias de análise visual para *test smells*:

a) ***TSInstant***. Considera apenas uma versão para análise. Dada uma versão V, é possível obter suas características, como as ocorrências de *test smells*. Atributos visuais, como símbolos e cores, são utilizados para representar características relacionadas a ocorrência dos *test smells*, por exemplo, a classe que ele se encontra;

b) ***TSEvolution***. Exibe a evolução de *test smells*. Dadas duas versões V_1 e V_2, é possível comparar o comportamento de *test smells*, por exemplo, se houve aumento, diminuição, propagação, remoção e inclusão. As cores são utilizadas para representar esse comportamento;

c) ***TSAuthor***. Apresenta o autor que alterou trechos de código que contém *test smells*. Dado um *test smell*, ts_1, os autores A_1, A_2, A_3 fizeram alteração naquele trecho e inseriram ou deixaram de remover o *test smell*, ele assume sua autoria.

[9] https://github.com/arieslab/TSVizzEvolution

TSInstant foi implementada na visualização *Graph View* e *Treemap View*, a análise de uma versão do código de teste permite visualizar as características do *test smell*, tais como a quantidade de ocorrências, possíveis autores, classes e métodos. A estratégia *TSEvolution* foi implementada na visualização *Timeline View*, permitindo a análise de duas versões do código de teste para visualização de suas características evolutivas, tais como se houve aumento ou diminuição na quantidade *test smells*. A estratégia *TSAuthor* foi implementada na visualização *Graph View*, onde pode-se ver os possíveis "autores" dos *test smells*.

4.3 Desenvolvimento da Ferramenta

A ferramenta `TSVizzEvolution` foi implementada seguindo o modelo de referência para a visualização de sistemas de software (CARD *et al*., 1999). No Quadro 4.1, são apresentadas as quatro etapas que compõem esse modelo: i) Fonte de Dados; ii) Processamento de Dados; iii) Mapeamento em Estruturas Visuais; e iv) Visualizações. Além disso, é descrito o funcionamento de cada etapa.

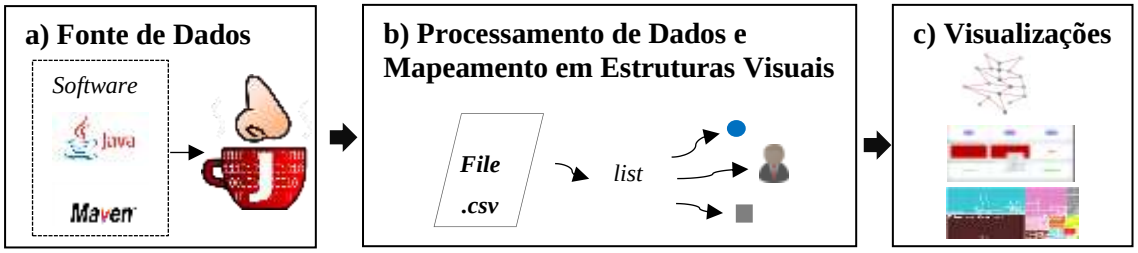
Quadro 4.1 - Modelo de Visualização da Informação

Etapas	Descrição
Fonte de Dados	Os dados a serem analisados e representados visualmente são coletados
Processamento de Dados	Os dados são processados, filtrados e transformados em representações internas apropriadas para manipulação
Mapeamento em Estruturas Visuais	As representações internas são mapeadas em estruturas de dados com as informações (cores, tamanhos e formas) necessárias para gerar as visões
Visualizações	As estruturas de dados com os atributos visuais criados são utilizadas para criar as visões

Fonte: Adaptado de Card *et al*. (1999).

O modelo de visualização da informação da ferramenta `TSVizzEvolution` é apresentado de maneira geral na Figura 4.1. As etapas de Processamento de Dados e Mapeamento em Estruturas Visuais, para o contexto, foram condensadas em uma mesma etapa (b). Cada etapa é explicada a seguir:

Figura 4.1 - Processo de Gerar a Visualização da `TSVizzEvolution`



Fonte: Do autor (2021).

a) **Fonte de Dados.** A fonte de dados é um arquivo `.csv` utilizado como entrada para a ferramenta `TSVizzEvolution`. Esse arquivo contém a detecção de até 21 *test smells* gerado pela ferramenta `JNose Test`[10];

b) **Processamento de Dados e Mapeamento em Estruturais Visuais.** De posse do arquivo `.csv` gerado, pode-se submeter o arquivo na `TSVizzEvolution`. Nesse momento, os artefatos do software e seus relacionamentos são armazenados em listas e mapeados em *Graph View*, *Treemap View* ou *Timeline View*, conforme especificação. As listas de classes, *test smells* e autores são carregadas na interface para serem selecionadas;

c) **Visualizações.** De acordo com o mapeamento anterior, é gerada a visualização dos artefatos coletados do `.csv`, podendo ser *Graph View*, *Treemap View* ou *Timeline View*.

4.3.1 Tecnologias Utilizadas

`TSVizzEvolution` foi implementada na linguagem de programação Java (Java 8) utilizando o Eclipse IDE for Java Developers, Versão 2020-12 (4.18.0)[11]. As representações visuais foram geradas utilizando os seguintes recursos:

a) **GraphStream**[12]. Biblioteca Java de manipulação de grafos que se concentra nos aspectos dinâmicos dos grafos. Seu foco principal é a modelagem de redes de interação dinâmica de vários tamanhos. Além disso, permite armazenar qualquer tipo de atributo de dados nos elementos do grafo, foi usado para criar a visualização *GraphView*;

b) **CSS (*Cascading Style Sheets*)**. Linguagem de programação para estilizar páginas HTML (*HyperText Markup Language*) (Silva, 2007). Foi utilizada para personalizar os componentes dos grafos, por exemplo, os nós;

c) **Diagramming for Java Swing**[13]. É uma biblioteca de diagramas MindFusion[14] que permite que os aplicativos criem e apresentem fluxogramas e diagramas de processo, fluxogramas de fluxo de trabalho e dados, diagramas de relacionamento de entidade de banco de dados, redes, gráficos, árvores e muito mais. Foi utilizado para criar a visualização *TreemapView*.

d) **Swing**. API (*Application Programming Interface*) que disponibiliza um conjunto de elementos gráficos para ser utilizado na plataforma Java, sendo escolhida por fornecer recursos extensíveis e configuráveis para construção de interfaces gráficas (SWING,

[10] http://jnose.herokuapp.com/
[11] https://www.eclipse.org/downloads/
[12] http://graphstream-project.org/
[13] https://jar-download.com/artifact-search/diagramming
[14] https://mindfusion.eu/

2015). Foi utilizada para a criação dos componentes, por exemplo, `JPanel`, `JLabel`, `JScrollPane` das telas de seleção dos arquivos `.csv` e para criar a visualização *Timeline View*;

e) **HTML**. Permite a escrita de documentos ou páginas web, que podem ser lidas por navegadores de Internet (Silva, 2007). Foi utilizada para fazer o *tooltip* na visualização *Timeline View*; ao passar o *mouse* sobre os elementos da visualização, são apresentadas suas características;

4.3.2 Modelagem da Ferramenta

Para modelar a ferramenta, foram elaborados o Diagrama de Casos de Uso (Seção 4.3.2.1) e o Diagrama de Classes (Seção 4.3.2.2) da `TSVizzEvolution`.

4.3.2.1 Diagrama de Casos de Uso

Na Figura 4.2, é apresentado o diagrama de casos de uso fornecendo uma visão geral do comportamento do sistema, com o ator e seus casos de uso. O ator representa os envolvidos nas atividades de teste de software, ou seja, o usuário utilizando a ferramenta `TSVizzEvolution`. O sistema de software `TSVizzEvolution` tem 6 casos de uso:

a) **Visualizar Ocorrências de *Test Smells***. O usuário deseja visualizar as ocorrências de *test smells* de uma versão do código de teste. Esse caso de uso é estendido pelos casos de uso Usar *Treemap View* e Usar *Graph View*;

b) **Visualizar Autores de *Test Smells***. O usuário deseja visualizar os possíveis autores dos *test smells*. Esse caso de uso é estendido pelo caso de uso Usar *Graph View*;

c) **Visualizar Evolução de *Test Smells***. O usuário deseja visualizar a evolução de *test smells* no código de teste. Esse caso de uso é estendido pelo caso de uso Usar *Timeline View*;

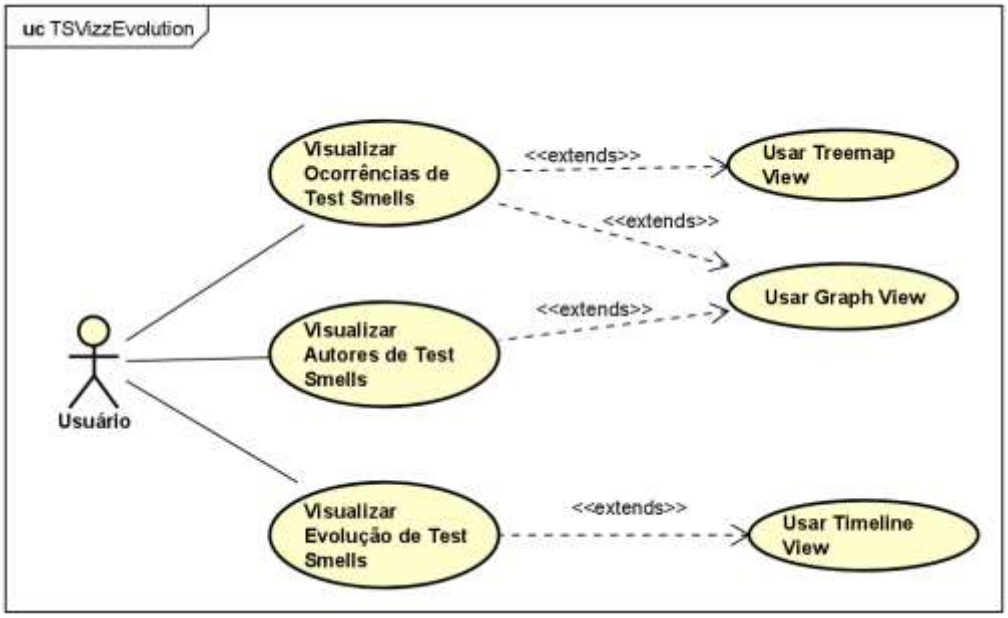
d) **Usar *Treemap View***. O usuário seleciona a técnica *Treemap View* para visualizar as ocorrências de *test smells* por meio de retângulos aninhados;

e) **Usar *Graph View***. O usuário seleciona a técnica *Graph View* para visualizar as ocorrências e/ou autores de *test smells* por meio de grafos. Nesse caso de uso, o usuário pode escolher a granularidade, sendo possível selecionar um projeto específico, todas as classes de teste, uma classe de teste específica, um *test smell* especifico, um autor específico, todos os autores ou um método específico;

f) **Usar *Timeline View***. O usuário seleciona a técnica *Timeline View* para visualizar a evolução de *test smells* perante duas versões do código de teste. Nesse caso de uso, o usuário pode escolher a granularidade, sendo possível selecionar um projeto específico,

todas as classes de teste, uma classe de teste específica, um *test smell* específico, um autor específico, todos os autores ou um método específico.

Figura 4.2 - Diagrama de Casos de Uso



Fonte: Do autor (2021).

4.3.2.2 Diagrama de Classes

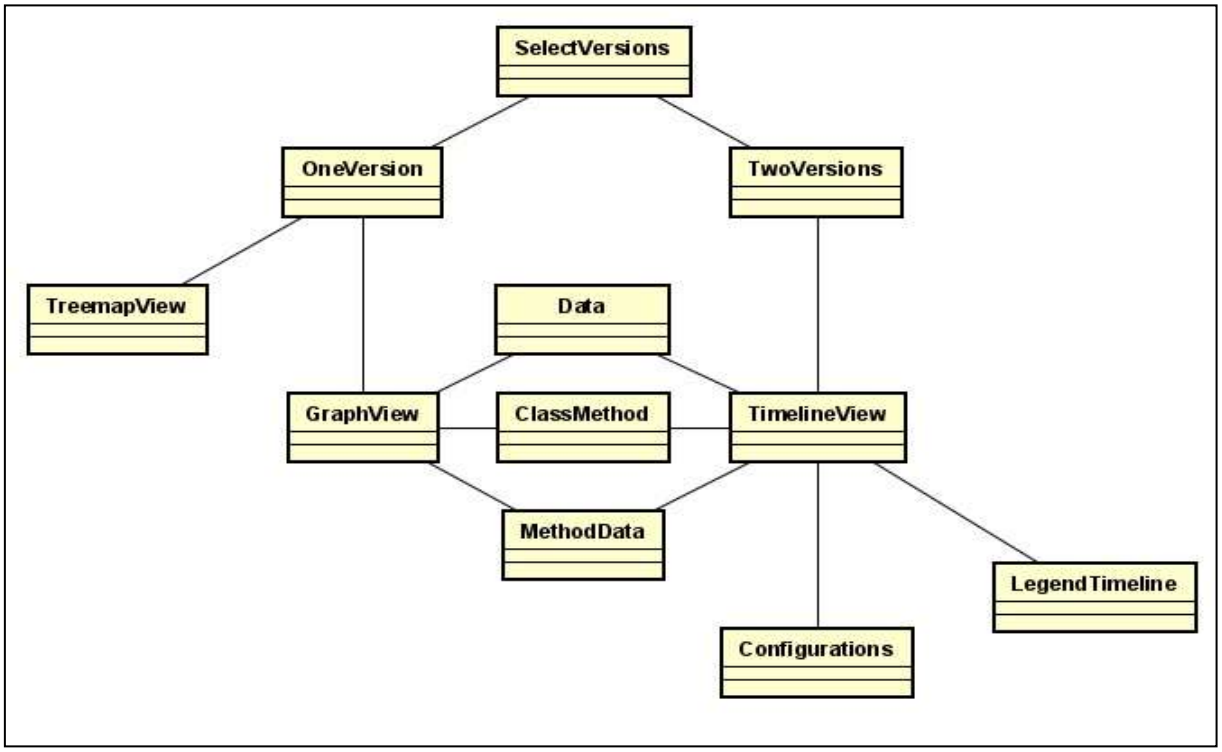
Na Figura 4.3, é apresentado o diagrama de classes simplificado (apenas identificador da classe) da `TSVizzEvolution`. A seguir, são explicadas as funções principais de cada classe:

a) ***OneVersion***: classe que fornece a tela de análise de uma versão, onde estão disponíveis para seleção do arquivo `.csv` e da técnica de visualização *Graph View* com suas granularidades e a técnica de visualização *Treemap View*;
b) ***TreemapView***: classe responsável pela geração da visualização *Treemap View*;
c) ***GraphView***: classe responsável pela geração da técnica de visualização *Graph View*;
d) ***TwoVersions***: classe que fornece a tela de análise de duas versões, onde estão disponíveis para seleção dos arquivos `.csv` e as granularidades para a visualização *Timeline View*;
e) ***TimelineView***: classe responsável pela geração da técnica de visualização *Timeline View*;
f) ***LegendTimeline***: classe responsável pela geração da legenda da *Timeline View*;
g) ***Configurations***: classe que contém as configurações para os componentes da visualização *Timeline View*, por exemplo, tamanho dos retângulos;
h) ***Data***: classe que contém as informações para gerarem as visualizações como os nomes das classes, do projeto, do autor e a quantidade de ocorrências de *test smells*;

i) ***ClassMethod***: classe que contém algumas informações referente aos métodos, como nomes da classes e nome dos métodos para gerar as visualizações *Graph View* e *Timeline View;*

j) ***MethodData***: classe que contém informações específicas dos métodos, como nomes dos métodos e as linhas de início e fim dos *test smells*, para gerar as visualizações *Graph View* e *Timeline View.*

Figura 4.3 - Diagrama de Classes `TSVizzEvolution`



Fonte: Do autor (2021).

4.3.3 Licença

A ferramenta `TSVizzEvolution` está licenciada sob a *GNU General Public License (GPLv3)*[15], por utilizar como entrada o arquivo gerado pela `JNose Test` licenciado sob a *GPLv3*. A *General Public License* permite modificar, copiar e distribuir o software, porém não permite que o código de um software seja apropriado por outros com restrições que impeçam que seja distribuído da mesma maneira que foi adquirido. Sendo assim, o código-fonte da `TSVizzEvolution` pode ser utilizado em software livre.

[15] https://www.gnu.org/licenses/quick-guide-gplv3.html

4.4 Visão Geral da Ferramenta

`TSVizzEvolution` exibe a ocorrência de *test smells* em diferentes versões do código de teste na linguagem Java para permitir identificar características, tais como, se a quantidade diminuiu ou aumentou ao longo do tempo com o surgimento das versões do código de teste e quem são os possíveis autores responsáveis por esses *test smells*. Primeiramente, `TSVizzEvolution` recebe como entrada a lista de *test smells* coletados do projeto a ser analisado (arquivo `.csv` gerado pela `JNose Test` (Seção 3.3)). Em seguida, os usuários executam três etapas para gerar a visualização de dados: i) **Seleção de Versões**, os usuários podem optar por analisar uma ou duas versões do código de teste; ii) **Granularidade da Análise**, os usuários podem selecionar a granularidade da análise para a geração das visualizações; e iii) **Técnicas de Visualização**, os usuários podem optar por visualizar os dados em *Graph View*, *Timeline View* ou *Treemap View*.

Primeiramente, a execução da ferramenta é feita com o seu `.exe` com clique duplo sobre `TSVizzEvolution.exe`[16]. Os arquivos `.csv` gerados pela `JNose Test` contendo a detecção dos *test smells* devem ser inseridos. Na Figura 4.4, é apresentado o funcionamento de `TSVizzEvolution`, com suas três etapas: i) Seleção de Versões; ii) Granularidade da Análise e iii) Técnicas de Visualização. Essas etapas são explicadas nos subtópicos a seguir.

4.4.1 Seleção de Versões

A primeira etapa para gerar as visualizações de *test smells* é **Seleção de Versões** (Figura 4.4) na qual são apresentadas aos usuários as possibilidades de (i) visualizar uma versão do código de teste (análise única) ou (ii) comparar duas versões do código de teste (análise da evolução). Para a análise única, os usuários precisam selecionar o arquivo de *test smells* (`.csv`). Para a análise de evolução, os usuários devem escolher duas versões do arquivo de *test smells* (`.csv`) gerados a partir de versões distintas do código de teste. É importante as versões serem inseridas em ordem crescente de data de criação para refletir a evolução correta, ou seja, primeiro, a versão mais antiga e, em seguida, a mais recente.

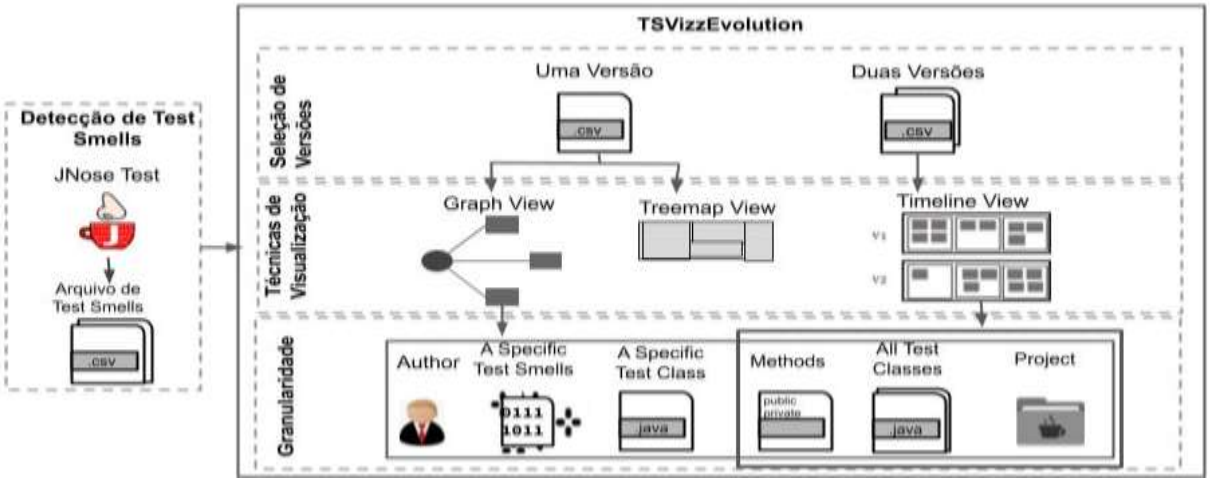
[16] https://github.com/arieslab/TSVizzEvolution/releases

4.4.2 Granularidade da Análise

Após selecionar as versões a serem analisadas, é necessário escolher o nível de granularidade no qual os dados devem ser apresentados. A abordagem estabelece um relacionamento entre artefatos de software para cada granularidade:

a) ***Project***. Exibe a relação entre todo o projeto e os *test smells* (1:N). Não são necessários filtros para executar a ferramenta com essa granularidade;

Figura 4.4 - Funcionamento `TSVizzEvolution`



Fonte: Do autor (2021).

b) ***All Test Classes***. Exibe a relação entre as classes de teste e os *test smells* (N:N). Não são necessários filtros para executar a ferramenta com essa granularidade;

c) ***A Specific Test Class***. Exibe a relação entre as classes de teste e os *test smells* (1:N). Para executar a ferramenta com essa granularidade, os usuários precisam selecionar no filtro qual classe de teste deve ser analisada;

d) ***A Specific Test Smells***. Exibe a relação entre as classes de teste e os *test smells* (N:1). Para executar a ferramenta com essa granularidade, os usuários devem selecionar no filtro qual o *test smell* deve ser analisado;

e) ***Author***. Exibe a relação entre autores com os *test smells* (N:N) e com as classes de teste (1:N). Para utilizar a ferramenta com essa granularidade, os usuários devem selecionar no filtro qual autor ou todos autores e o *test smell* a ser analisado. Lembrando que a autoria é definida “por culpa”, ou seja, se um usuário inserir um *test smell* ou modificar um método que possui *test smells*, mas não o corrige, ele “estava ciente” do problema e assumiu sua co-autoria;

f) ***Methods***. Exibe a relação entre um *test smell* específico, uma classe de teste específica e os métodos (1:N). Os usuários precisam selecionar no filtro qual *test smell* e qual classe de teste devem ser analisados.

4.4.3 Técnicas de Visualização

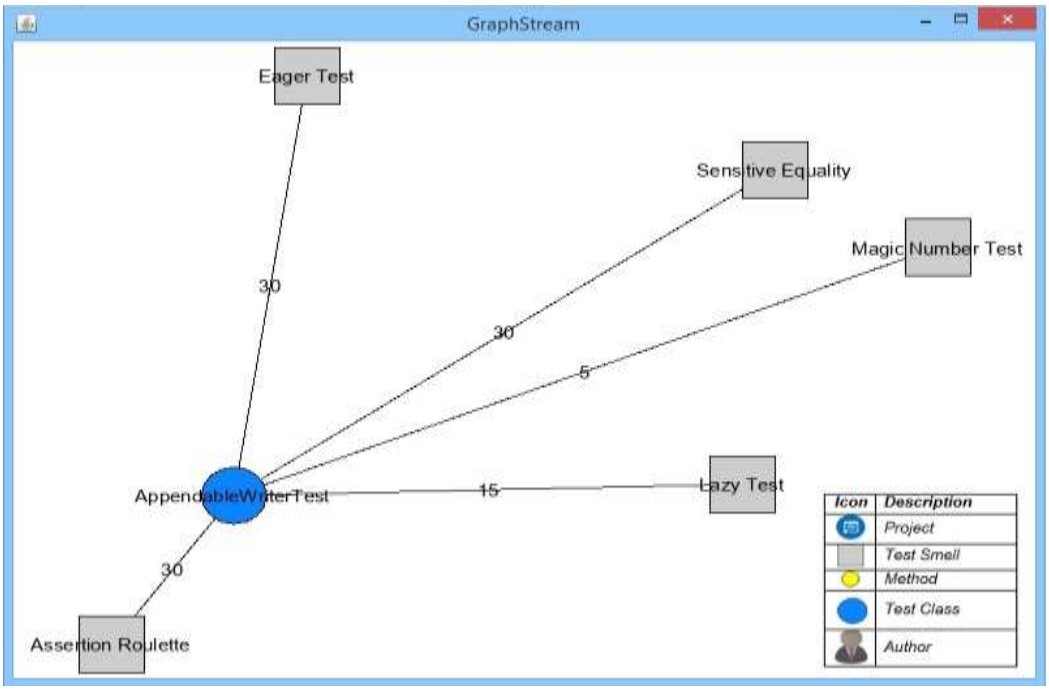
Na `TSVizzEvolution`, são implementadas três técnicas de visualização para gerar as visualizações de *test smells*: i) *Graph View;* ii) *Treemap View; e* iii) *Timeline View* (Figura 4.4). Cada técnica implementada tem o objetivo de representar de maneira visual as ocorrências e a evolução de *test smells*, para facilitar a visualização desses problemas. A seguir, são explicadas as três técnicas utilizadas:

a) ***Graph View***. Nessa visualização, pode-se analisar as ocorrências de *test smells* para uma versão do código de teste, podendo escolher entre todos os níveis de granularidade (Seção 4.4.2). Os dados retornados por `JNose Test` são estruturados para criar os grafos de visualização compostos por nós e arestas. Os componentes do grafo variam de acordo com a granularidade escolhida e podem ser arrastados conforme necessidade. Os ícones que são os nós do grafo são detalhados na Figura 4.5 na legenda disponível no canto inferior ao grafo;

b) ***Treemap View***. Nessa visualização, pode-se analisar as ocorrências de *test smells* para uma versão do código de teste. Os *test smells* são representados por retângulos, sendo que cada retângulo possui uma cor, gerada aleatoriamente. Além disso, os tamanhos dos retângulos representam as suas ocorrências, ou seja, quanto maior o retângulo mais ocorrências de *test smells* existem (Figura 4.6);

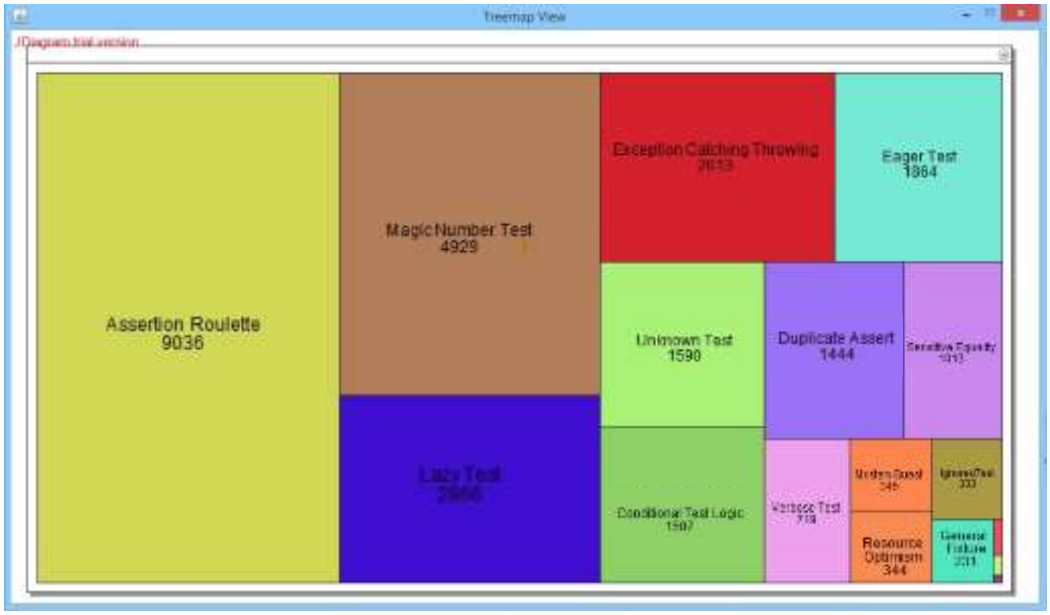
c) ***Timeline View***. Nessa visualização, são analisadas duas versões do código de teste e os usuários podem escolher entre três granularidades: i) *Project*; ii) *All Test Classes*; ou *iii) Methods*. Os artefatos (projeto, classe de teste e *test smell*) do software são representados por retângulos e as versões são apresentadas em paralelo horizontalmente, conforme ordem de inserção dos arquivos `.csv`. Conforme Figura 4.7, os retângulos são aninhados da seguinte forma: i) **retângulo externo** representa o projeto ou a classe de teste (conforme granularidade selecionada) sendo representado pela borda na cor rosa; e ii) **retângulos menores** representam cada *test smell* detectado em uma classe de teste ou projeto conforme granularidade, com as cores representando o comportamento durante a evolução, podendo ser verde, azul, cinza ou vermelho.

Figura 4.5 - *Graph View*



Fonte: Do autor (2021).

Figura 4.6 - *Treemap View*

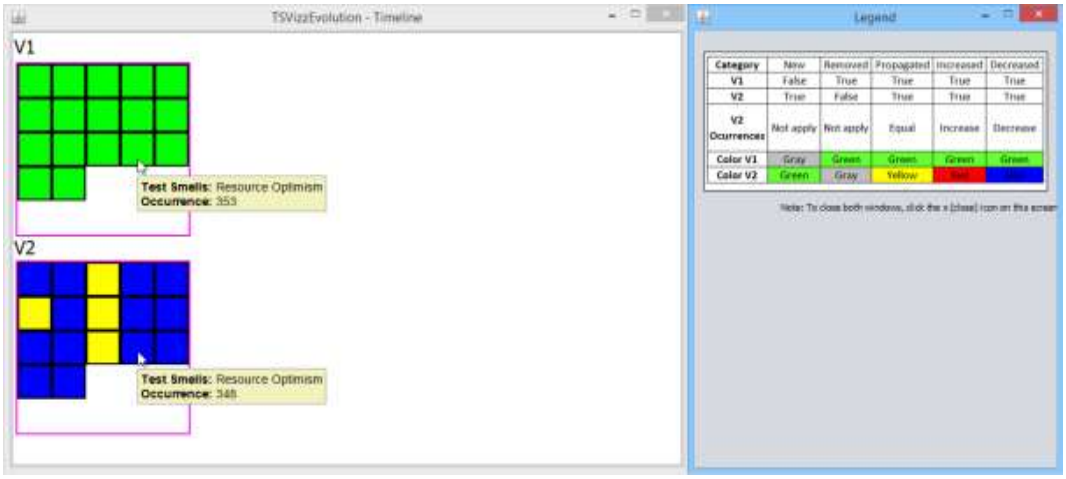


Fonte: Do autor (2021).

As posições dos retângulos nas duas versões são mantidas para facilitar a rastreabilidade entre classes de teste e *test smells*. As cores dos *test smells* são definidas conforme sua existência e ocorrências (Quadro 4.2) e apresentadas ao lado da *timeline*. Os usuários podem ver as classes de teste, projeto e o nome dos *test smells* e suas

ocorrências ao posicionar o *mouse* sobre cada retângulo (Figura 4.7). As categorias são (Versão um - V_1 e Versão dois - V_2):

Figura 4.7 - *Timeline View*



Fonte: Do autor (2021).

- **Novo**. Se V_1 não tiver o *test smell* específico (*False*) e V_2 (*True*), a categoria é "**New**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**cinza**" e "**verde**", respectivamente;
- **Removido**. Se V_1 tiver o *test smell* específico (*True*) e V_2 (*False*), a categoria é "**Removido**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**verde**" e "**cinza**", respectivamente;
- **Propagado**. Se V_1 e V_2 tiverem o *test smell* específico (*True - True*) e as quantidade de ocorrências for a mesma (*Igual*), a categoria é "**Propagado**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**verde**" e "**amarelo**", respectivamente;
- **Aumentado**. Se V_1 e V_2 tiverem um *test smell* específico (*True - True*), mas a quantidade de *test smells* aumenta em V_2 (**Aumenta**), a categoria é "**Aumentado**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem a cores "**verde**" e "**vermelho**", respectivamente;
- **Diminuído**. Se V_1 e V_2 tiverem um *test smell* específico (*True - True*), mas a quantidade de *test smells* diminui na V_2 (**Diminui**), a categoria é "**Diminuído**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem a cores "**verde**" e "**azul**".

4.5 Funcionamento da Ferramenta

Para exemplificar o funcionamento da ferramenta, foi utilizado o código de teste do sistema de software `Commons IO`[17], uma biblioteca de utilitários para ajudar no desenvolvimento de sistemas I/O[18]. Foi utilizada a versão 2.1 do código de teste desse sistema de software para a análise de uma versão (análise única) e as versões 2.1 e 2.6 foram utilizadas para a análise de duas versões (análise de evolução).

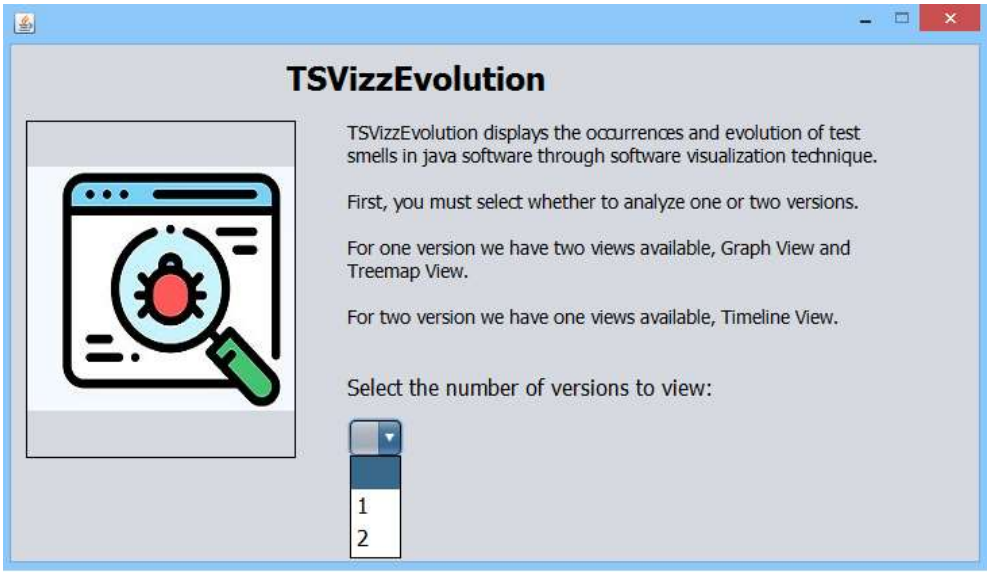
Quadro 4.2 - Classificação dos *Test Smells*

Categoria	**V_1**	**V_2**	**V_2 Ocorrências**	**Cor V_1**	**Cor V_2**
Novo	*False*	*True*	Não se aplica	Cinza	Verde
Removido	*True*	*False*	Não se aplica	Verde	Cinza
Propagado	*True*	*True*	Igual	Verde	Amarelo
Aumentado	*True*	*True*	Aumentado	Verde	Vermelho
Diminuído	*True*	*True*	Diminuído	Verde	Azul

Fonte: Do autor (2021).

Primeiramente, deve-se escolher a quantidade de versões a serem analisadas (uma ou duas) (Figura 4.8). Para uma versão, deve se escolher *Graph View* ou *Treemap View* (Figura 4.9). Para duas versões, têm-se a visualização *Timeline View* (Figura 4.10).

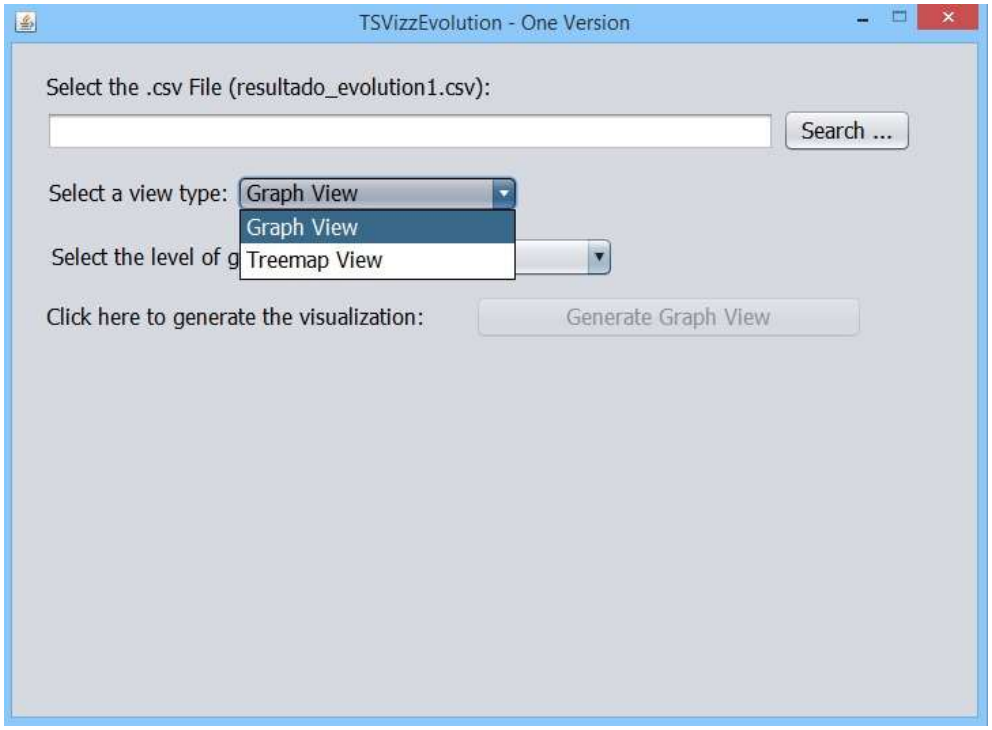
Figura 4.8 - Seleção de Quantidade de Versões



Fonte: Do autor (2021).

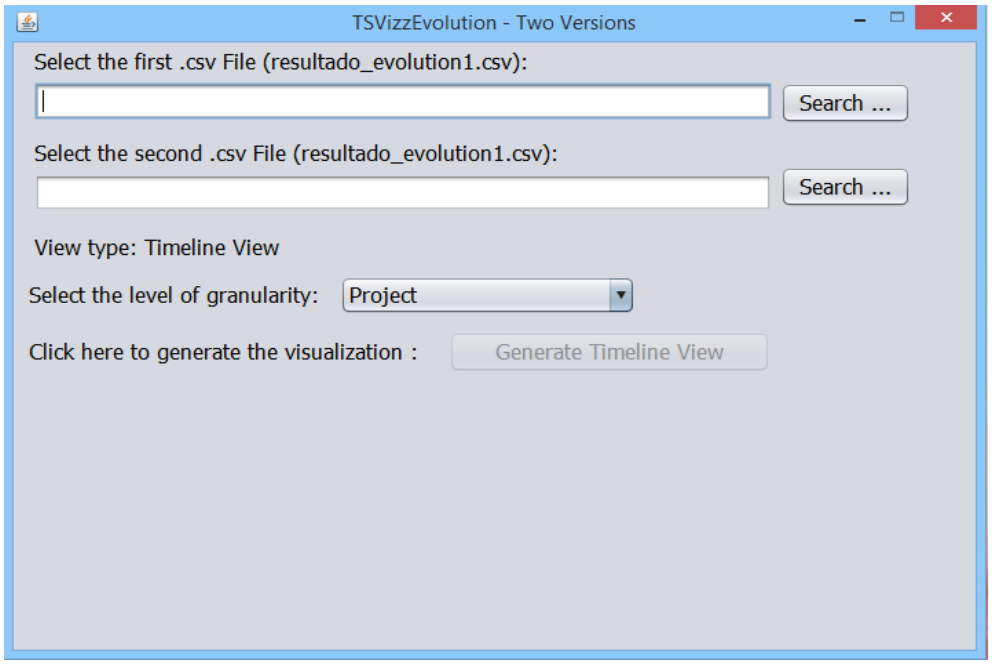
[17] https://commons.apache.org/proper/commons-io/
[18] Os sistemas I/O (*Input/Output*) têm a função de organizar e controlar operações de entrada e saída de dados.

Figura 4.9 - Seleção de Uma Versão



Fonte: Do autor (2021).

Figura 4.10 - Seleção de Duas Versões



Fonte: Do autor (2021).

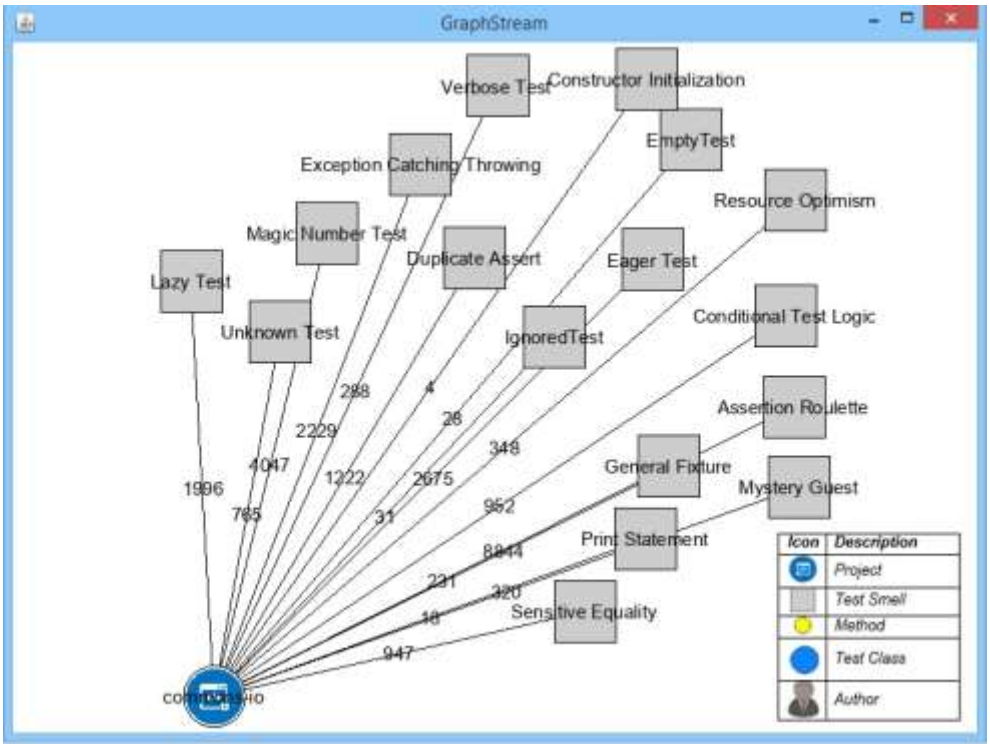
4.5.1 Análise Única

As técnicas de visualização disponíveis para essa análise são a *Graph View* e *Treemap View*. Para o exemplo utilizando a técnica de visualização *Graph View*, foi selecionada a versão

2.1 e todas as granularidades. Na Figura 4.11, é mostrada a visualização gerada após a seleção da granularidade *Project*. Essa visão permite que os usuários obtenham uma ideia geral dos *test smells* existentes no software e a quantidade de *test smells*. Nas arestas, é informada a quantidade de ocorrências.

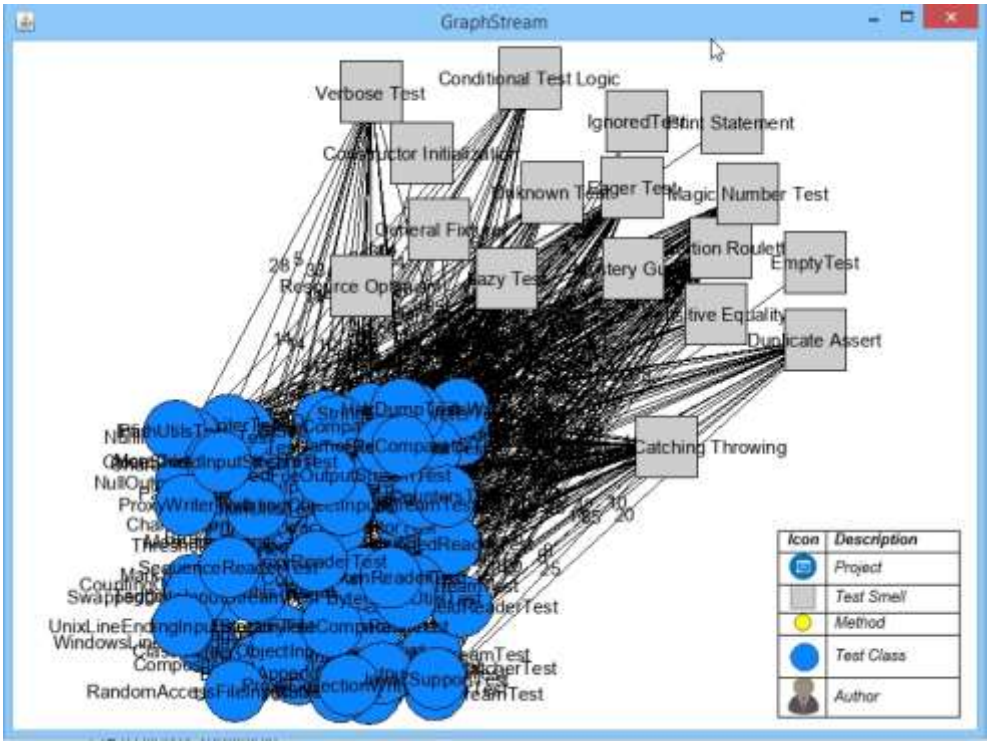
Na Figura 4.12, é mostrada a visualização para a granularidade *All Test Classes*, representando uma visão geral das classes e dos *test smells.E*ssa granularidade é a menos recomendada, pois, para um sistema de software com muitas classes de teste, a visualização pode ficar sobrecarregada. Na Figura 4.13, é mostrada a visualização para a granularidade *A Specific Test Class* selecionando a classe `AppendableOutputStreamTest`, pode-se ver que essa classe está vinculada a 5 *test smells* (*Eager Test, Sensitive Equality, Magic Number Test, Lazy Test* e *Assertoin Roulette*). Essa visão permite aos usuários escolher uma classe de teste específica para analisar em detalhes os *test smells* que a classe possui. Na Figura 4.14, é apresentado o *test smell Unknow Test* vinculado a 24 classes de teste. Os usuários podem gerar essa visualização a partir da granularidade *A Specific Test Smells* para observar todas as classes que possuem determinado *test smell*.

Figura 4.11 - Visualização *Graph View* - `Commons IO` (Granularidade: *Project*)



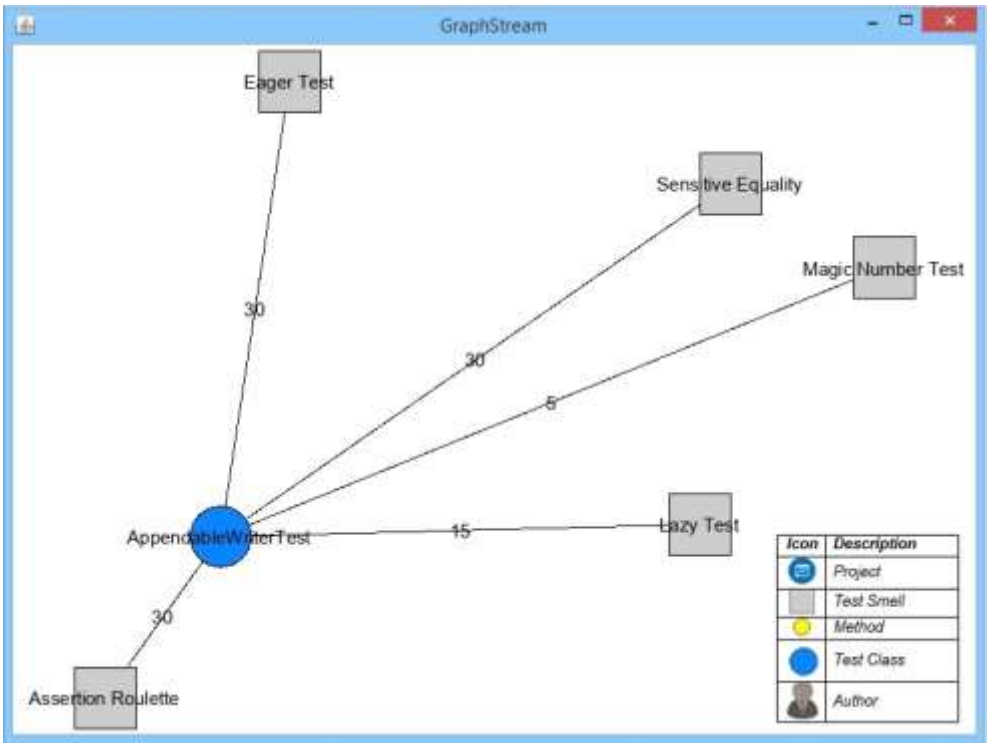
Fonte: Do autor (2021).

Figura 4.12 - Visualização *Graph View* - `Commons IO` (Granularidade: *All Test Classes*)



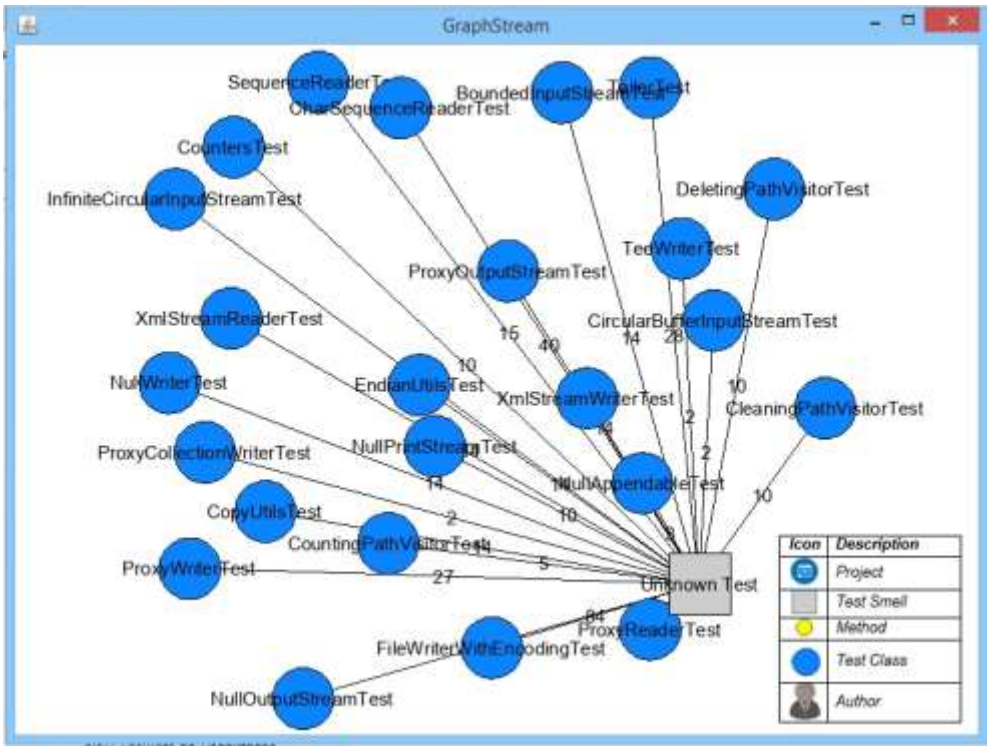
Fonte: Do autor (2021).

Figura 4.13 - Visualização *Graph View* - `Commons IO` (Granularidade: *A Specific Test Class*)



Fonte: Do autor (2021).

Figura 4.14 - Visualização *Graph View* - `Commons IO` (Granularidade: *A Specific Test Smells*)



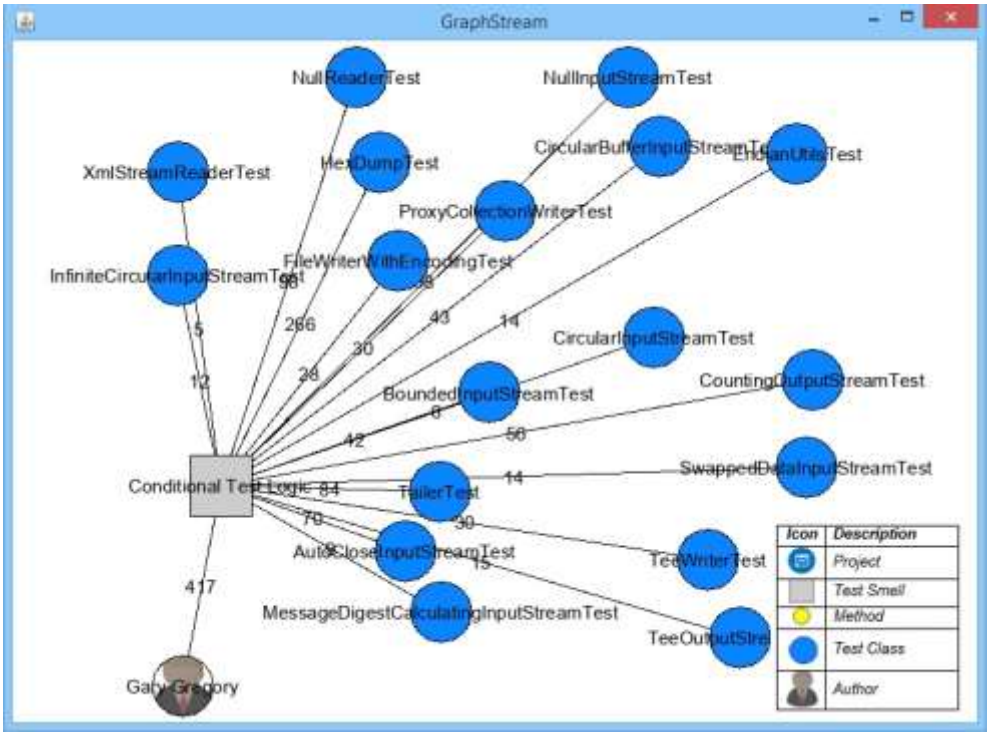
Fonte: Do autor (2021).

Na Figura 4.15, é exibida a visualização gerada utilizando a granularidade *Author*. É apresentado um autor (*Gary Gregory*) vinculado ao *test smell Conditional Test Logic* e esse *test smell* está vinculado a 18 classes de teste. Essa quantidade indica que essa pessoa é o autor ou assumiu a autoria "por culpa" desse *test smell* em 18 classes.

Na Figura 4.16, são apresentados 5 autores (*Benson Margulies, Benedikt Ritter, Kristian Rosenvold, Gary Gregory e Gary D. Gregory*) para o *test smell Magic Number Test*, assumiram a autoria "por culpa". Na Figura 4.17, é mostrada a visualização com a granularidade *Methods*, apresentando o método `testInputSize0FilterSize1` da classe `CharacterFilterReaderTest` que possui o *test smell Assertion Roulette* e ocorre na linha 37, com essa visualização pode-se localizar o método problemático na classe de teste para corrigi-lo.

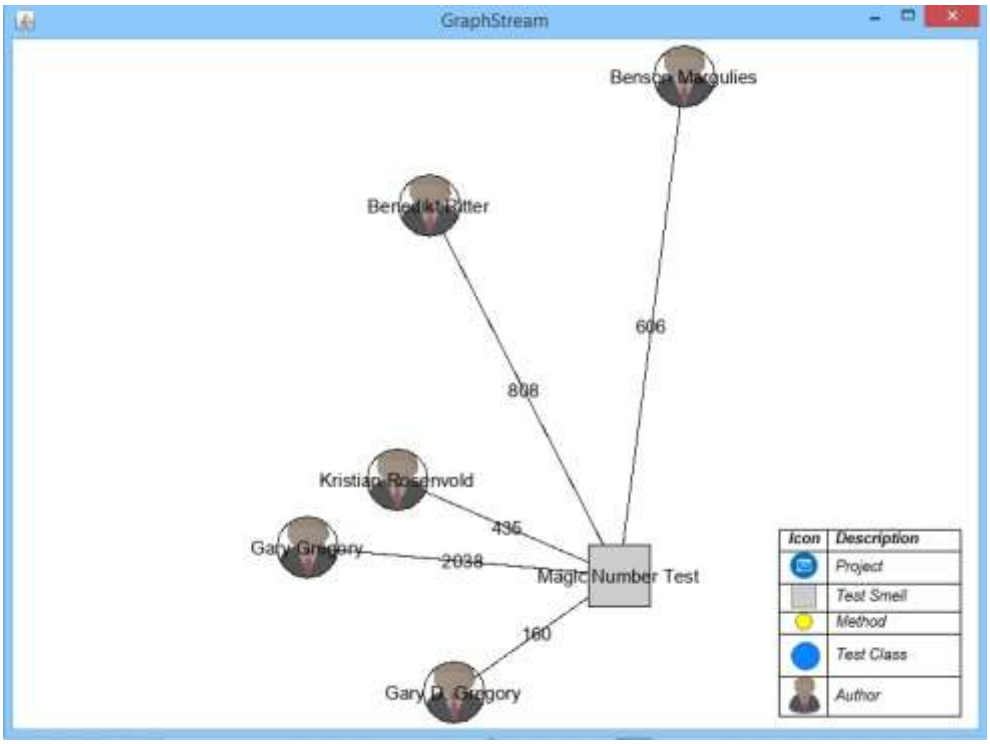
Na Figura 4.18, pode-se verificar a visualização gerada com a técnica *Treemap View*. Para gerar essa visualização, não há granularidades, pois a ideia é obter informações do projeto como um todo para verificar quais *test smells* possuem mais ocorrências.

Figura 4.15 - Visualização *Graph View* - `Commons IO` (Granularidade: *Author*)



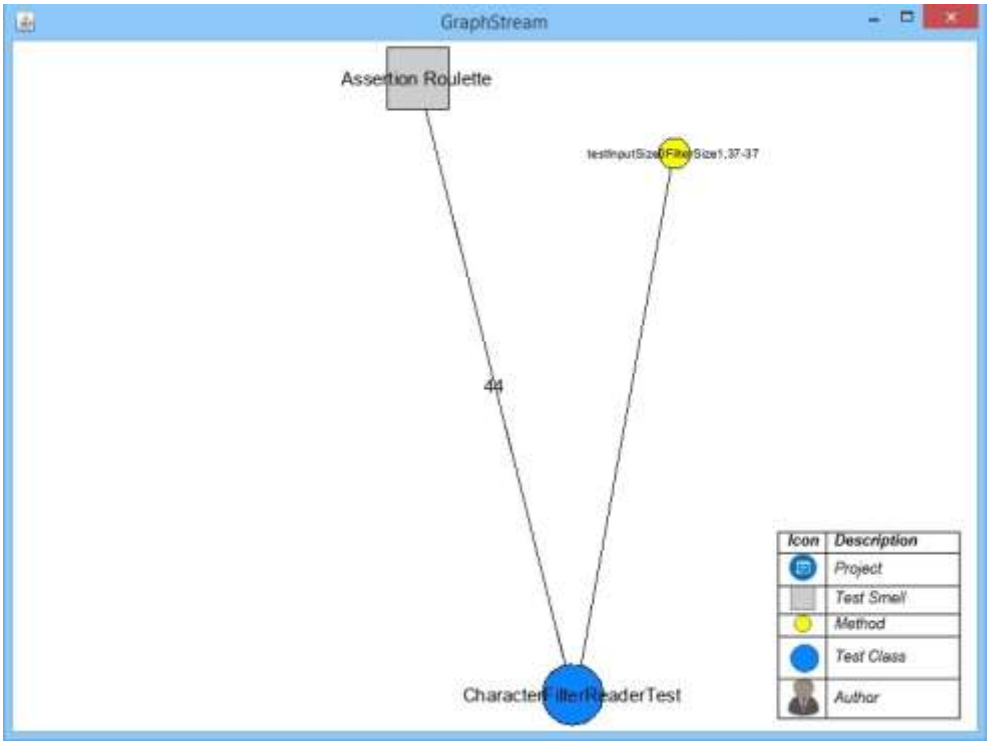
Fonte: Do autor (2021).

Figura 4.16 - Visualização *Graph View* - `Commons IO` (Granularidade: *Author - All*)



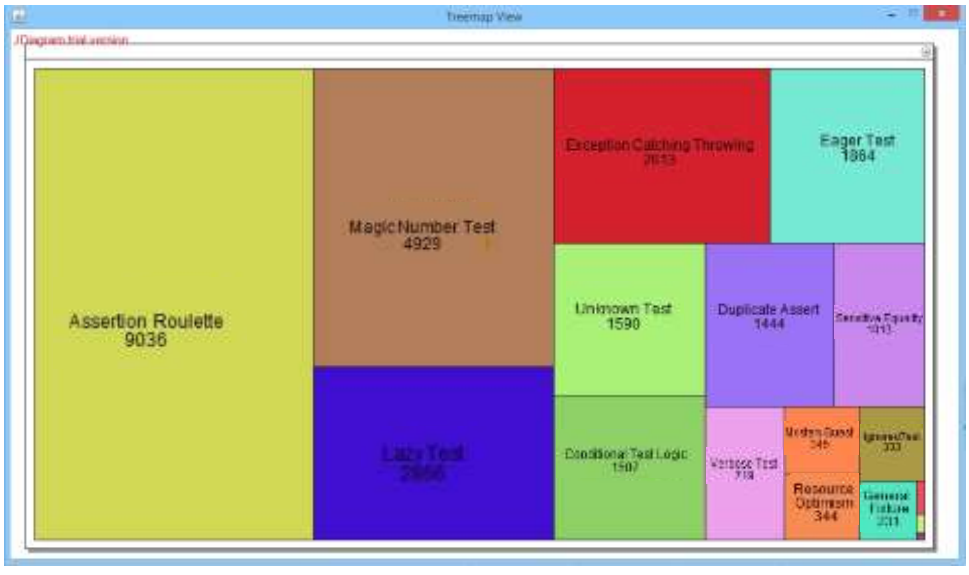
Fonte: Do autor (2021).

Figura 4.17 - Visualização *Graph View* - `Commons IO` (Granularidade: *Methods*)



Fonte: Do autor (2021).

Figura 4.18 - Visualização *Treemap View* - `Commons IO`



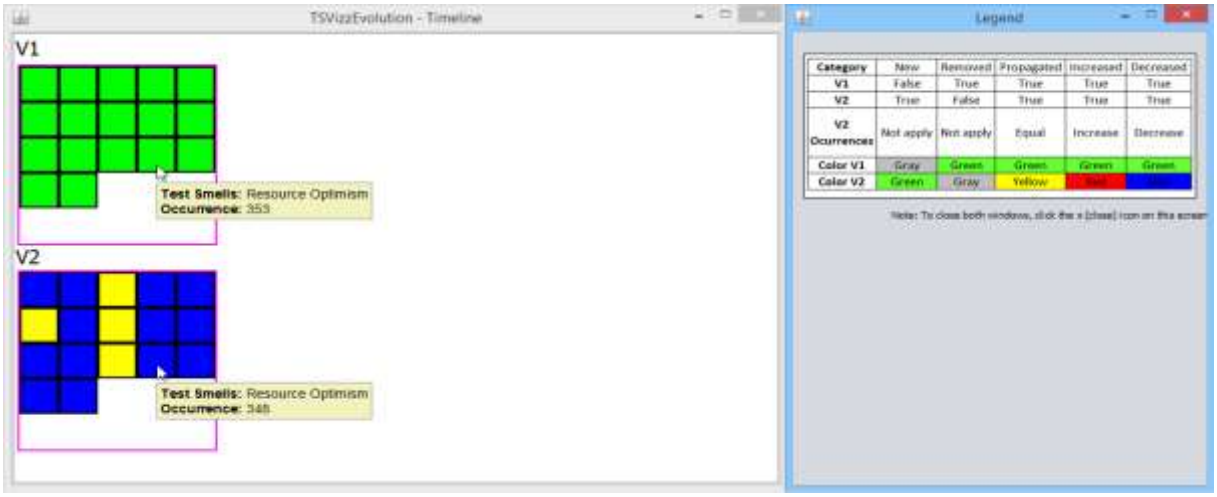
Fonte: Do autor (2021).

4.5.2 Análise de Evolução

A técnica de visualização disponível para essa análise é a *Timeline View*, que possui três granularidades (*Project*, *All Test Classes e Methods*). Na Figura 4.19, para a granularidade *Project*, as ocorrências do *test smell Resource Optimism* diminuíram em 5 ocorrências. Na Figura 4.20, para a granularidade *All Test Classes*, o *test smell Verbose Test* teve aumento de 5

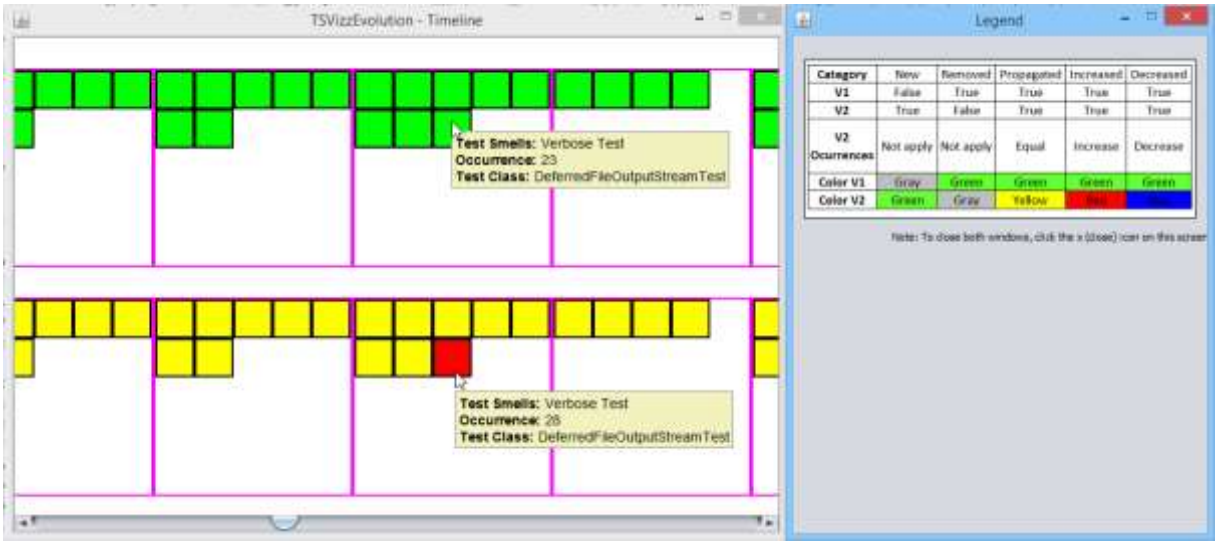
ocorrências na classe de teste `DeferredFileOutputStreamTest`. Na Figura 4.21, para a granularidade *Methods*, *o test smell Unknow Test* na versão 2.1 `Commons IO` na classe `EndianUtilsTest` ocorre no método `testCtor`, com início na linha 34 e fim na linha 38.

Figura 4.19 - Visualização *Timeline View* - `Commons IO` (Granularidade: *Project*)



Fonte: Do autor (2021).

Figura 4.20 - Visualização *Timeline View* - `Commons IO` (Granularidade: *All Test Classes*)



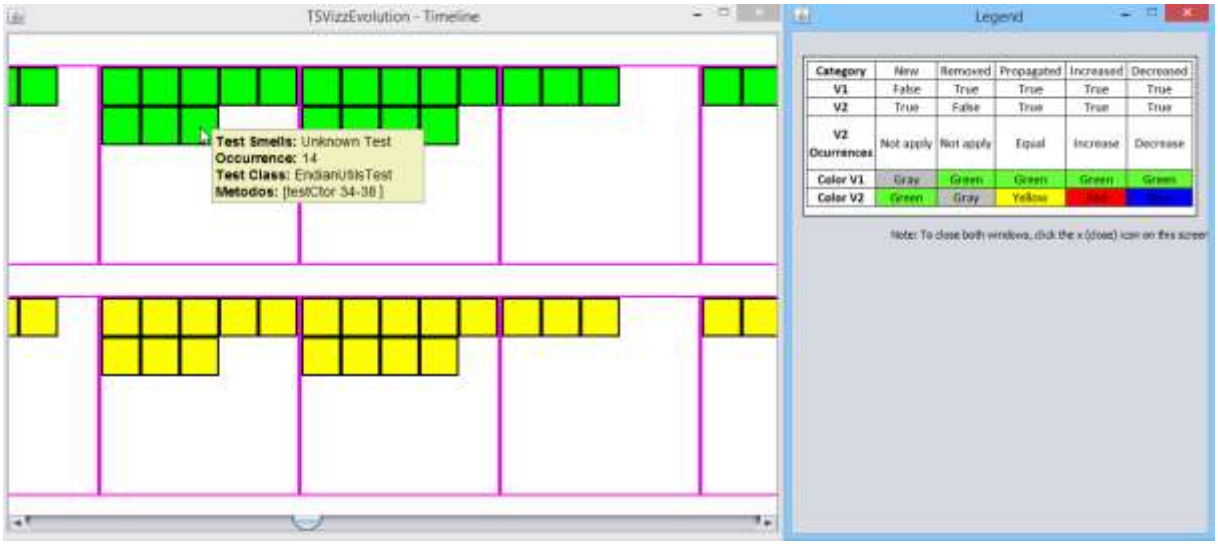
Fonte: Do autor (2021).

4.6 Considerações Finais

Neste capítulo, foi apresentada a abordagem proposta neste trabalho e a ferramenta que a implementa. A abordagem definiu estratégias de análise visuais para *test smells.* A visualização de software tem sido pouco abordada para *test smells*. Para fornecer uma ferramenta visual para *test smells*, primeiramente, foram definidas três estratégias visuais para representar os dados: *TSInstant*, *TSEvolution e TSAuthor*. De acordo com as estratégias

definidas, foi implementada uma ferramenta que fornece três técnicas de visualização de software para a representação de ocorrências, evolução e possíveis autores de *test smells*.

Figura 4.21 - Visualização *Timeline View* - `Commons IO` (Granularidade: *Methods*)



Fonte: Do autor (2021).

5 AVALIAÇÃO DA ABORDAGEM E FERRAMENTA

5.1 Considerações Iniciais

Nesse capítulo, é apresentada a avaliação da abordagem e da ferramenta realizada por meio de um experimento controlado. O experimento consistiu na execução de tarefas com o uso de duas ferramentas de visualização de *test smells*. O processo de avaliação da abordagem tem por objetivo identificar se as três estratégias de análise visual para *test smells* implementadas na ferramenta são capazes de facilitar a visualização de *test smells* e, consequentemente, auxiliar os envolvidos nas atividades de teste a melhorar a qualidade do código de teste. Neste capítulo, são apresentados os procedimentos adotados e os resultados obtidos com a realização do experimento.

Este capítulo está organizado da seguinte forma. Na Seção 5.2, é apresentado o planejamento do estudo. Na Seção 5.3, são descritos os resultados e as discussões. Na Seção 5.4, as ameaças a validade são descritas.

5.2 Planejamento

O objetivo dessa avaliação é verificar se a abordagem criada e implementada na ferramenta `TSVizzEvolution` apoia a hipótese sobre o qual esse trabalho foi baseado (Seção 1.2). Essa avaliação foi realizada por meio de um experimento controlado, cujo protocolo foi baseado no trabalho de (SANTANA, 2021) e está detalhado no APÊNDICE A.

Foram buscadas na literatura ferramentas que utilizam recursos visuais para representar as ocorrências de *test smells*, para serem utilizadas no estudo de comparação das ferramentas. As ferramentas encontradas foram `VITRuM` e `TestQ`, explicadas na Seção 6.3. Foi selecionada a ferramenta `VITRuM` por analisar *test smells* em classes de teste geradas pelo *framework* JUnit, assim como a ferramenta `JNose Test` cujo os resultados de detecção de *test smells* são utilizados como entrada para a `TSVizzEvolution`. A ferramenta `TestQ` não foi selecionada por utilizar outro *framework* de testes, o XUnit.

Foi definido o uso de 2 *test smells* no experimento, definido por meio das ferramentas utilizadas. A ferramenta `VITRuM` detecta 7 *test smells*, sendo um deles (*Indirect Testing*) não abordado pela ferramenta `JNose Test`, dentre as 6 opções restantes disponíveis, primeiramente, foi selecionado o *test smell Assertion Roulette*, pois é um dos *test smells* mais frequentes no código de teste (PALOMBA *et al.*, 2016; PERUMA, 2018). Posteriormente, foi selecionado, aleatoriamente, o *test smell Sensitive Equality*.

Os sistemas de *software* escolhidos para serem utilizados no experimento controlado foram validados em um estudo anterior da ferramenta `JNose Test` (Virgínio *et al*., 2019) e atendem os requisitos das duas ferramentas de visualização de *test smells* utilizadas, sendo eles `Commons IO`[19], `Commons Text`[20] e `Commons Email`[21]. Esses sistemas de software são explicados a seguir.

5.2.1 Caracterização dos Sistemas de Software Utilizados

Os sistemas de software selecionados estão disponíveis no *GitHub* e atendem os requisitos das ferramentas `JNose Test` e `VITRuM`, descritos a seguir: i) Linguagem Java; ii) Projetos Reais; iii) Projetos *Open Source*; iv) Projetos Maven; e v) Testes desenvolvidos com JUnit.

Os sistemas de software `Commons Email` e `Commons Text` foram selecionados para a realização do experimento e o sistema de software `Commons IO` foi utilizado para treinamento. Uma visão geral sobre os projetos é descrita no Quadro 5.1. A coluna *Star* informa a quantidade de usuários que possui o projeto como favorito. A coluna *Commits* informa a quantidade de alterações realizadas no projeto. A coluna *Último Commit* Analisado informa a data da última modificação analisada no projeto. A coluna *Contributors* informa a quantidade de usuários que alteraram o código fonte do projeto. A seguir, o detalhamento dos sistemas de software:

Quadro 5.1 - Caracterização dos Sistemas de Software Selecionados

Projeto	***Star***	***Commits***	**Último *Commit* Analisado**	***Contributors***
`Commons IO`	829	3493	27/01/2022	84
`Commons Email`	91	941	28/09/2021	23
`Commons Text`	227	1552	21/01/2022	48

Fonte: Adaptado de https://github.com/apache/commons-email, https://github.com/apache/commons-text, https://github.com/apache/commons-io (Dados atualizados em 27/01/2022)

a) **`Commons IO`.** Fornece classes utilitárias para operações do `File IO`, oferecendo recursos para as seguintes categorias:
   - Classes utilitárias. Essas classes no pacote `org.apache.commons.io` fornecem comparação de arquivos e cadeias de caracteres, sendo elas: `IOUtils`,

[19] https://github.com/apache/commons-io
[20] https://github.com/apache/commons-text
[21] https://github.com/apache/commons-email

`FilenameUtils`, `FileUtils`, `IOCase`, `FileSystemUtils` e `LineIterator`;

- Classes de filtro. As classes de filtro do pacote `org.apache.commons.io.filefilter` fornecem métodos para filtrar arquivos com base em critérios lógicos, em vez de comparações baseadas em cadeias de caracteres, sendo elas `NameFileFilter`, `WildcardFileFilter`, `SuffixFileFilter`, `PrefixFileFilter`, `OrFileFilter`, `AndFileFilter`;
- Classes do `File Monitor`. As classes do `File Monitor` do pacote `org.apache.commons.io.monitor` fornecem controle para rastrear alterações em um arquivo ou pasta específica e permitem executar ações de acordo com as alterações, sendo elas: `FileEntry`, `FileAlterationObserver` e `FileAlterationMonitor`;
- Classes comparadoras. As classes comparadoras do pacote `org.apache.commons.io.comparator` permitem comparar e classificar arquivos e diretórios facilmente, sendo elas: `NameFileComparator`, `SizeFileComparator` e `LastModifiedFileComparator`;
- Classes de fluxo. As classes de fluxo fornecem implementações do `InputStream` no pacote `org.apache.commons.io.input` e do `OutputStream` no pacote `org.apache.commons.io.output` para executar tarefas úteis nos fluxos. Sendo algumas delas: `NullOutputStream`, `eeOutputStream`, `ByteArrayOutputStream`, `CountingOutputStream`, `CountingOutputStream`, `ProxyOutputStream` e `LockableFileWriter`;

b) **`Commons Email`.** API para envio de e-mails. Algumas das classes de e-mail fornecidas são:

- `SimpleEmail`. Essa classe é usada para enviar e-mails básicos sem anexos;
- `MultiPartEmail`. Essa classe é usada para enviar mensagens com anexos;
- `HtmlEmail`. Essa classe é usada para enviar e-mails formatados em HTML. Possui todos os recursos como `MultiPartEmail` e fornece suporte a imagens incorporadas;

- `ImageHtmlEmail`. Essa classe é usada para enviar e-mails formatados em HTML com imagens incorporadas;
- `EmailAttachment`. Essa classe é usada para facilitar o manuseio de anexos, utilizando as instâncias de `MultiPartEmail` e `HtmlEmail`;

c) **`Commons Text`.** Fornece um conjunto de ferramentas para o processamento de texto. Anteriormente, essas ferramentas estavam disponíveis em um projeto denominado `Commons Lang`[22], porém foi adicionado ao `Commons Text` para aprimoramento das funções, algumas classes foram substituídas e outras adicionadas. As classes substituídas tiveram o prefixo `Str` ao invés do antigo `String` para garantir não colisão com classes Java padrão atuais ou futuras. Algumas das classes fornecidas são:

- `StrBuilder`. Essa classe é uma substituição para `StringBuffer`, sendo o seu comportamente semelhante, porém fornece métodos adicionais. Permite a construção de *string* a partir de partes constituintes, sendo mais flexível que o `StringBuffer`. Alguns dos métodos adicionais fornecidos são: i) `toCharArray/getChars`. Maneira mais simples de obter um intervalo da matriz de caracteres; ii) `delete`. Exclusão de *char* ou *string*; iii) `replace`. Pesquisa e substitui *caracter* ou *string*;
- `StrSubstitutor`. Essa classe permite substituir variáveis dentro de uma *string*;
- `StrTokenizer`. Essa classe permite dividir as *strings* em *strings* menores (*tokens*), sendo uma substituição da classe `StringTokenizer`. Fornece além das funções existentes, outras funções para comparar similaridade e diferenças entres textos, permitindo mais controle e flexibilidade por meio das funções: i) `StringEscapeUtils`. Permite gerar partes de texto, por exemplo, pode ser usado para a geração de senhas; ii) `Similarity e Distance`. Permite comparar a semelhança e distância de *strings* usando medidas. Para semelhança, fornece medidas como a semelhança de cosseno, semelhança de *jaccard*, entre outras. Para a comparação da distância, fornece medidas como a distância do cosseno, a distância *Levenshtein*, entre outras.

5.2.2 Condução do Experimento

Para evitar o efeito de aprendizado dos sistemas de software, foi definido o uso do *design* quadrado latino, cujas unidades experimentais, ou seja, o que está sendo utilizado no

[22] https://commons.apache.org/proper/commons-lang/

experimento, devem receber os tratamentos agrupados de duas maneiras diferentes (linhas e colunas), a quantidade de linhas, colunas e tratamentos devem ser o mesmo (CALEGARE, 2009). Assim, seguindo o quadrado latino, os sistemas de software e as ferramentas foram organizados (Quadro 5.2). Cada linha do quadro representa um grupo. Cada grupo iniciou com uma ferramenta e/ou sistema de software distinto, pode-se ver que os grupos 1 e 3 começaram o experimento utilizando a mesma ferramenta, porém o sistema foi distinto, o mesmo ocorre com os grupos 2 e 4, garantindo que cada grupo executou as atividades em ordens distintas.

Para contemplar o *design* quadrado latino, a quantidade de participantes deve ser múltipla da quantidade de linhas e colunas definidas. Conforme Quadro 5.2, tem-se 4 linhas e 4 colunas (refere a ferramenta e aos sistemas de software, o id do grupo não é considerado), logo, a quantidade de participantes deve ser múltipla de 4. Foi definido que o experimento seria composto de duas etapas:

a) **Avaliação Piloto**. Avaliação com quantidade reduzida de participantes, para padronizar como o treinamento seria realizado e fazer ajustes se necessários. Os resultados não foram utilizados para a análise dos dados. Foi definida a quantidade de 4 participantes e cada participante foi destinado há um grupo;
b) **Avaliação Final**. Avaliação aprimorada após a avaliação piloto, com quantidade maior de participantes. Os resultados foram utilizados para análise dos dados. Foi definida a quantidade de 16 participantes na avaliação final, sendo 4 participantes em cada grupo. Cada participante foi destinado há um grupo sequencialmente, participante 1 no grupo 1 e assim sucessivamente.

Quadro 5.2 - *Design* Quadrado Latino do Experimento

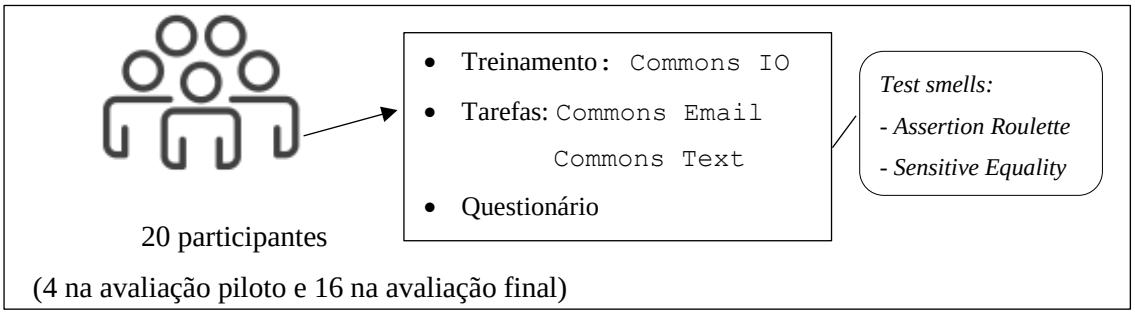
Grupo	**1ª Ferramenta**	**1º Projeto**	**2ª Ferramenta**	**2º Projeto**
#1	TSVizzEvolution	Commons Email	VITRuM	Commons Text
#2	VITRuM	Commons Text	TSVizzEvolution	Commons Email
#3	TSVizzEvolution	Commons Text	VITRuM	Commons Email
#4	VITRuM	Commons Email	TSVizzEvolution	Commons Text

Fonte: Do autor (2021).

Na Figura 5.1, é apresentada uma ideia geral do experimento, pode-se ver a quantidade de participantes nas avaliações piloto e final, os sistemas de software utilizados e os *test smells* abordados. Os convites foram enviados individualmente para cada participante por e-mail e o experimento também foi realizado individualmente. No estudo de (MARSHALL *et al.*, 2013),

recomenda-se que apesar da quantidade de participantes em estudos pode ser arbitrária, o ideal é entre 15 a 25 participantes. Sendo assim, o experimento está de acordo com as recomendações. Primeiramente, foi explicado o objetivo geral do experimento. Em seguida, foram abordados os tópicos (nas letras c, d, e, f a ferramenta e o sistema de software foram definidos de acordo com o grupo que o participante pertencia (Quadro 5.2)):

Figura 5.1 - Funcionamento Geral do Experimento



Fonte: Do autor (2021).

a) ***Test Smells*:** Foram explicados os conceitos e exemplificados os *test smells*: *Assertion Roulette* e *Sensitive Equality;*
b) **Ferramentas para a Visualização de *Test Smells*:** Foram explicadas as ferramentas para a visualização de *test smells* `TSVizzEvolution` e `VITRuM`;
c) **Treinamento 1.** Realização de treinamento, com atividades semelhantes a Tarefa 1, com condução pelo pesquisador usando a ferramenta `TSVizzEvolution` ou `VITRuM - Commons IO`;
d) **Tarefa 1.** `TSVizzEvolution` ou `VITRuM - Commons Email` ou `Commons Text`;
e) **Treinamento 2.** Realização de treinamento, com atividades semelhantes a Tarefa 2, com condução pelo pesquisador usando a ferramenta `TSVizzEvolution` ou `VITRuM - Commons IO`;
f) **Tarefa 2.** `TSVizzEvolution` ou `VITRuM - Commons Email` ou `Commons Text`;
g) **Questionário.** Questionário para caracterização dos participantes e *feedback* do experimento (APÊNDICE J).

5.2.3 Avaliação Piloto

Antes da avaliação final, foi realizada uma avaliação piloto com 4 participantes, sendo todos profissionais da indústria, um deles é estudante de graduação. Esse estudo foi fundamental para revisar os conceitos, fazer ajustes, padronizar como o treinamento seria

realizado. Além disso, serviu para avaliar se os participantes foram capazes de entender as tarefas. Para a análise de dados, a avaliação piloto não foi considerada.

Na avaliação piloto, os participantes utilizaram as ferramentas em seus computadores. Dessa forma, foi necessária a instalação dos programas requisitos das ferramentas. Para a execução da `TSVizzEvolution`, foi necessária a instalação do *Java 8*. Para a execução da ferramenta `VITRuM`, foi necessária a instalação do *Jdk 11*, da IDE IntelliJ[23] e do *plug-in* `VITRuM` na IDE. Foi observado que a execução das atividades estava dispendiosa e gastando tempo desnecessário, pois o foco era o funcionamento das ferramentas. Por isso, para a avaliação final, foi feito o ajuste na condução do experimento para o participante acessar o computador do pesquisador de forma remota.

O protocolo foi lido pelo pesquisador e a comunicação foi mantida durante o experimento utilizando do *google meet*[24], o *link* para cada reunião foi enviado por e-mail no agendamento para cada participante individualmente. A sequência das atividades foi seguida conforme no APÊNDICE B, no APÊNDICE D, no APÊNDICE F e no APÊNDICE H de acordo com o grupo a que o participante pertencia.

Durante a execução das tarefas, os participantes responderam 11 perguntas em um *google docs*, todas as questões eram abertas. A utilização de questões abertas demandava tempo desnecessário para os participantes escrever nome das classes, *test smells*, autores, projetos, métodos, entre outros. Nesse quesito, também foi feito o ajuste da criação de um formulário para a avaliação final. No Quadro 5.3, pode-se ver o tempo de cada participante na avaliação piloto. A média dos tempos foi 2 h e 18 min, o que tornou o estudo exaustivo. Os participantes são referenciados pelo identificador Ppn, sendo n um número sequencial de 1 a 4.

5.2.4 Avaliação Final

Na avaliação final, as tarefas ocorreram de forma remota e os participantes controlaram o computador do pesquisador utilizando o software *AnyDesk*[25]. O protocolo foi lido pelo pesquisador e a comunicação foi mantida durante o experimento utilizando do *google meet*, o *link* para cada reunião foi enviado por e-mail no agendamento para cada participante individualmente. A sequência das atividades foi seguida conforme no APÊNDICE B, no APÊNDICE D, no APÊNDICE F e no APÊNDICE H de acordo com o grupo a que o participante pertencia.

[23] https://www.jetbrains.com/pt-br/idea/
[24] https://meet.google.com/
[25] https://anydesk.com/pt

Quadro 5.3 - Tempo Total Por Participante na Avaliação Piloto

Id	Duração Total
Pp1	02:08:00
Pp2	02:47:00
Pp3	01:55:00
Pp4	02:23:00

Fonte: Do autor (2021).

Durante a execução das tarefas, os participantes responderam a 11 perguntas utilizando um formulário disponibilizado no *google forms*[26], com questões de múltipla escolha para as questões que possuíam respostas objetivas e algumas questões abertas àquelas referentes as opiniões dos participantes. As perguntas também estão disponíveis no APÊNDICE C, no APÊNDICE E, no APÊNDICE G e no APÊNDICE I. No Quadro 5.4, pode-se ver as atividades de cada grupo e os apêndices que correspondem a cada formulário que os participantes responderam no *forms*. Ao final das tarefas, os participantes responderam um questionário de caracterização dos participantes e *feedback* do experimento, também disponibilizado no *google forms*, o *link* foi fornecido aos participantes no protocolo (`https://forms.gle/1mis7iVRHrbBSvws5`). Esse questionário está disponível no APÊNDICE J. Os participantes são referenciados pelo identificador Prn, sendo n um número sequencial de 1 a 16.

Quadro 5.4 - Formulários das Tarefas Realizadas Durante o Experimento

Grupo	Atividades	Formulário
G1	APÊNDICE B - Atividades do Estudo Experimental com Ferramentas para Visualização de Test Smells - Grupo 1	APÊNDICE C - Perguntas Experimento Controlado G1
G2	APÊNDICE D - Atividades do Estudo Experimental com Ferramentas para Visualização de Test Smells - Grupo 2	APÊNDICE E - Perguntas Experimento Controlado G2
G3	APÊNDICE F - Atividades do Estudo Experimental com Ferramentas para Visualização de Test Smells - Grupo 3	APÊNDICE G - Perguntas Experimento Controlado G3
G4	APÊNDICE H - Atividades do Estudo Experimental com Ferramentas para Visualização de Test Smells - Grupo 4	APÊNDICE I - Perguntas Experimento Controlado G4

Fonte: Do autor (2021).

[26] https://docs.google.com/forms/u/0/

5.3 Resultados e Discussões

Nesta seção, os resultados são discutidos e apresentados em 4 subseções. Na subseção 5.4.1, é feita a caracterização dos participantes. Na subseção 5.3.2, é apresentada a resposta a primeira questão de pesquisa. Na subseção 5.3.3, é apresentada a resposta a segunda questão de pesquisa. Na subseção 5.3.4, a questão de pequisa principal é respondida. Na subseção 5.3.5, são abordadas as considerações dos participantes sobre o experimento.

5.3.1 Caracterização dos Participantes

Os participantes foram caracterizados por meio de um questionário (APÊNDICE J). Dos 16 participantes recrutados para o experimento, 5 participantes (31,3%) atuam na indústria, 5 participantes (31,3%) atuam na academia e 6 participantes (37,5%) atuam em ambos. No Quadro 5.5, são apresentadas as titulações e as experiências dos participantes. Pode-se ver que 6 participantes (37,50%) são estudantes de graduação, 4 participantes (25%) possuem gradução completa, 4 participantes (25%) são estudantes de mestrado e 2 participantes (12,5%) são estudantes de doutorado.

Quadro 5.5 - Titulação e Experiência dos Participantes

Grupo	Id	Escolaridade	Teste de Software Profissional	Teste de Software Acadêmico
G1	Pr1	Estudante de Graduação	Não possuo	< 1 ano
G2	Pr2	Graduação Completa	>= 1 ano e < 5 anos	>= 1 ano e < 5 anos
G3	Pr3	Estudante de Graduação	< 1 ano	< 1 ano
G4	Pr4	Estudante de Graduação	< 1 ano	< 1 ano
G1	Pr5	Estudante de Mestrado	>= 10 anos	>= 1 ano e < 5 anos
G2	Pr6	Estudante de Graduação	< 1 ano	Não possuo
G3	Pr7	Estudante de Graduação	Não possuo	< 1 ano
G4	Pr8	Graduação completa	Não possuo	< 1 ano
G1	Pr9	Estudante de Mestrado	>= 1 ano e < 5 anos	>= 1 ano e < 5 anos
G2	Pr10	Graduação completa	>= 1 ano e < 5 anos	Não possuo
G3	Pr11	Estudante de Graduação	< 1 ano	< 1 ano
G4	Pr12	Estudante de Mestrado	>= 1 ano e < 5 anos	>= 1 ano e < 5 anos
G1	Pr13	Estudante de Doutorado	Não possuo	>= 1 ano e < 5 anos
G2	Pr14	Estudante de Mestrado	Não possuo	< 1 ano
G3	Pr15	Estudante de Doutorado	Não possuo	>= 1 ano e < 5 anos
G4	Pr16	Graduação completa	Não possuo	< 1 ano

Fonte: Do autor (2021).

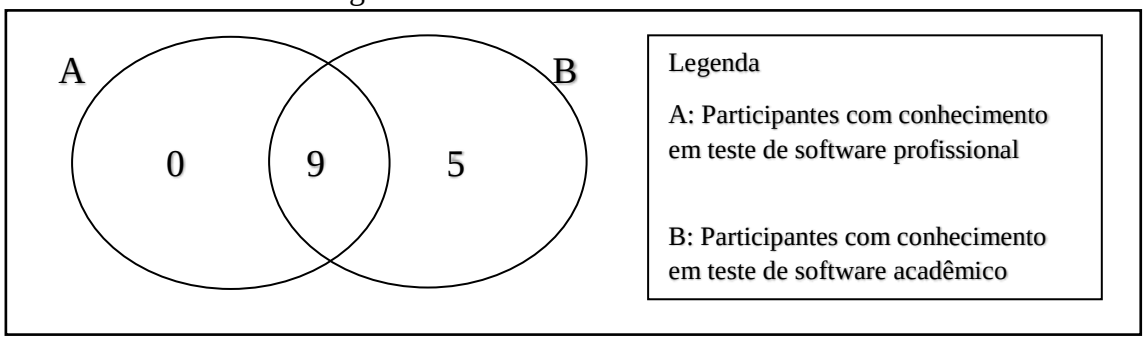
Quanto à experiência em teste de software, eles foram questionados sobre o tempo em anos na área de teste de software profissional e/ou acadêmica, sendo profissional referente a

atividades na indústria e acadêmica a atividades em instituições de ensino. Todos possuem algum conhecimento em teste de software (Figura 5.2), 9 participantes (56,3%) possuía conhecimento em teste de software profissional e 14 participantes (87,5%) possuía conhecimento em teste de software acadêmico. Vale ressaltar que alguns participantes possuíam conhecimento profissional concomitante ao conhecimento acadêmico.

Quanto ao conhecimento referente aos *test smells* (Figura 5.3), 6 participantes (37,5%) sabem o que são *test smells* mas nunca trabalharam com *test smells*, 4 participantes (25%) são pesquisadores de tópicos relacionados a *test smells*, 3 participantes (18,8%) sabem o que são *test smells* e 3 participantes (18,8%) nunca ouviram falar.

Quanto ao conhecimento referente à visualização de software (Figura 5.4), 9 participantes (56,3%) sabem o que é visualização de software, 4 participantes (25%) sabem o que é visualização de software, mas nunca trabalharam com visualização, 2 participantes (12,5%) nunca ouviram falar e 1 (6,3%) participante trabalha com tópicos relacionados a *test smells*, mas desconhece visualização. O conhecimento dos participantes destaca a importância de realizar treinamentos, pois alguns não possuíam conhecimentos desses tópicos.

Figura 5.2 - Conhecimento em Teste de Software



Fonte: Do autor (2021).

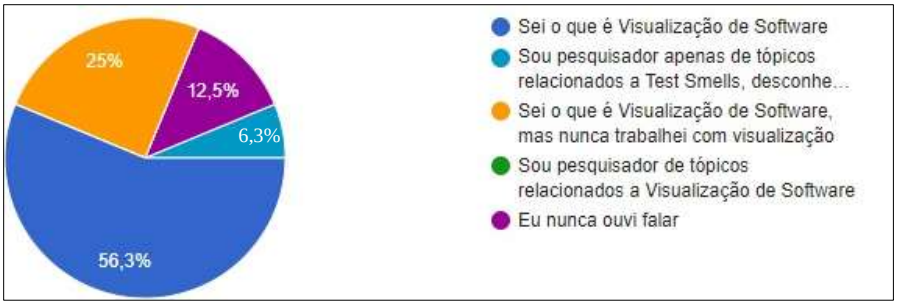
Pode-se concluir que a caracterização dos participantes contém fatores que podem influenciar a sua compreensão durante o experimento. Esses fatores são:

a) Diferentes níveis de experiência dos participantes;
b) Conhecimento prévio de *test smells;*
c) Conhecimento prévio de visualização de software;
d) Conhecimento na área de teste de software.

Figura 5.3 - Conhecimento dos Participantes Referente aos *Test Smells*

Fonte: Do autor (2021).

Figura 5.4 - Conhecimento dos Participantes Referente a Visualização de *Software*



Fonte: Do autor (2021).

5.3.2 QP1: Como a ferramenta `TSVizzEvolution` pode facilitar as decisões de projeto e os processos de tomada de decisão?

Para responder QP1, os participantes foram questionados durante as tarefas realizadas no experimento sobre suas percepções quanto à visualização dos possíveis autores dos *test smells* se auxiliaria nas decisões de projeto e nos processos de tomada de decisão, conforme pergunta a seguir. As perguntas são referenciadas pelo identificador Pn, sendo n um número sequencial de 1 a 11.

> "P5: Você acha que a informação do possível autor do *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique."

As respostas de todos os participantes estão disponíveis no APÊNDICE K. Pode-se ver que todos os participantes disseram que ao visualizar os autores pelo *test smells* ajuda nas decisões de projeto e nos processos de tomada de decisão, por exemplo, para remanejamento de pessoal e fornecimento de treinamentos. Pode-se destacar a resposta do Pr9: "Sim, pois permite que os gestores do projeto consigam pensar em formar de organizar e treinar a equipe, para tratar as ocorrências dos TestSmells e aumentar a manutenção do código" e do Pr14: "Sim. Para buscar capacitar e orientar o testador para evitar a geração de novos test smells.".

Além disso, foi solicitado aos participantes visualizarem qual *test smell* com mais ocorrências no código de teste utilizados no experimento e foram questionados se a informação fornecida era útil, conforme pergunta a seguir:

> "P8: Qual *test smell* com mais ocorrências? Essa informação pode ser útil para o projeto? Explique."

Todos os participantes conseguiram identificar com o uso da `TSVizzEvolution` qual *test smell* com mais ocorrências, que se tratava do *Assertion Roulette*. Eles destacaram ser importante saber essa informação para adotar estratégias durante o desenvolvimento de teste, por exemplo, a resposta do Pr11: "Assertion Roulette. Sim. Essa informação pode ajudar a identificar os test smells que mais ocorrem no projeto o que pode servir para promover uma conscientização aos desenvolvedores sobre os problemas que estão sendo inseridos. Além disso, é possível identificar os pontos que precisam de melhoria.". Pode-se verificar as respostas de todos os participantes no APÊNDICE L.

De acordo com as respostas dos participantes, pode-se responder a QP1. Conclui-se que a ferramenta `TSVizzEvolution` auxilia nos processos de tomada de decisão ao informar os possíveis autores dos *test smells* e ao exibir quais *test smells* com mais ocorrências no projeto, o que não é abrangido pela outra ferramenta abordada no estudo.

5.3.3 QP2: Como a ferramenta `TSVizzEvolution` facilita a visualização de *test smells* em comparação com a ferramenta `VITRuM`?

Primeiramente, para responder a QP2, foram medidos os tempos para realizar as atividades de visualização com ambas ferramentas utilizadas no estudo (`TSVizzEvolution` e `VITRuM`). No Quadro 5.6, pode-se verificar o tempo gasto pelos participantes no uso de cada ferramenta e também o tempo total. O tempo total inclui as explicações, o treinamento e a execução das tarefas e o maior tempo toal foi do participante Pr7 com 1 h e 18 minutos. Apenas dois participantes realizaram as atividades da ferramenta `TSVizzEvolution` em tempo menor do que utilizando a `VITRuM`, sendo eles os participantes Pr6 e Pr13. Os demais 14 participantes gastaram em média 9 minutos a mais para realizarem as atividades utilizando a `TSVizzEvolution` do que a `VITRuM`. Mas, isso justifica-se pelo fato de que a ferramenta `VITRuM` não realiza todas as análises que a `TSVizzEvolution` realiza, por exemplo:

a) Possíveis autores dos *test smells*;
b) Comparação de duas versões do código de teste;
c) Três técnicas de visualização;

d) Diversas granularidades;

e) Exibição da quantidade de ocorrências por *test smells* e autores;

f) Detalhamento de método e linha onde ocorre os *test smells*.

Quadro 5.6 - Tempo Gasto na Execução das Tarefas

Id	TSVizzEvolution	VITRuM	Tempo Total
Pr1	00:25:00	00:16:00	01:03:00
Pr2	00:20:00	00:15:00	00:55:00
Pr3	00:31:00	00:19:00	01:15:00
Pr4	00:23:00	00:13:00	00:56:00
Pr5	00:21:00	00:08:00	00:52:00
Pr6	00:14:00	00:16:00	00:55:00
Pr7	00:16:00	00:07:00	01:18:00
Pr8	00:21:00	00:07:00	01:01:00
Pr9	00:19:00	00:09:00	00:55:00
Pr10	00:23:00	00:17:00	01:05:00
Pr11	00:21:00	00:08:00	00:54:00
Pr12	00:25:00	00:17:00	01:12:00
Pr13	00:18:00	00:19:00	01:10:00
Pr14	00:12:00	00:09:00	00:57:00
Pr15	00:17:00	00:08:00	00:54:00
Pr16	00:17:00	00:06:00	00:54:00

Fonte: Do autor (2021).

Além disso, ainda para responder a QP2, foram feitas perguntas no questionário ao final do experimento para comparar `TSVizzEvolution` e `VITRuM` em relação a como exibem os *test smells* e a facilidade em que a visualização proporciona para corrigir os problemas no código de teste. Os participantes foram questionados sobre qual ferramenta utilizariam, as respostas foram:

a) 12 dos participantes (75%) usariam o `TSVizzEvolution`, as justificativas de suas escolhas estão disponíveis no APÊNDICE M. O participante Pr8 justificou que: "Considerando apenas o tópico Test Smells, eu usaria a TSVizz porque apresenta diversas informações como a autoria dos test smells e a sua localização de maneira precisa. Além disso, a TSVizz suporta a visualização de 21 test smells, enquanto que a Vitrum suporta apenas 7. Outro ponto, a TSVizz é uma ferramenta standalone e a Vitrum é um plugin do IntelliJ. Considerando que não preciso que os dados sobre test smells sejam atualizados com frequência (e.g., a cada modificação do código), a escolha pela ferramenta standalone parece mais acertada, pois é necessário menos processamento e independe de ambiente de programação.";

b) 3 participantes (18,8%) usariam ambas as ferramentas, as justificativas estão disponíveis no APÊNDICE N. O participante Pr14 destacou que: “Vitrum tem como vantagem ser integrado com IDE e TSVizzEvolution tem como vantagens poder visualizar várias informações a respeito da evolução de test smells em projetos.”;
c) 1 participante (6,3%) não usaria nenhuma das ferramentas, porém não justificou a sua resposta; e
d) nenhum participante escolheu a ferramenta `VITRuM`.

Quando os participantes foram questionados sobre qual ferramenta representa visualmente melhor as ocorrências de *test smells*, todos os participantes responderam que é a ferramenta `TSVizzEvolution`. Todas as justificativas estão no APÊNDICE O, algumas justificativas são descritas a seguir:

a) Pr3: “Tanto as abordagens gráficas quando na riqueza de detalhes e informações, permite com que o usuário tenha uma análise mais abrangente que a outra.”;
b) Pr4: “Clareza e detalhe nas informações fornecidas:
   - Quantidade de cada test smell
   - Localização de cada test smell
   - Responsável por cada test smell
   - Comparação de duas versões do projeto Isso destaca como as três técnicas de visualização de software escolhidas representam adequadamente a ocorrências dos *test smells*.”

Os participantes também foram questionados sobre qual ferramenta exibe melhor a visualização da evolução de *test smells*. Todos os participantes informaram a ferramenta `TSVizzEvolution`, as justificativas estão disponíveis no APÊNDICE P. O participante Pr1, assim como outros destacaram que: “A ferramenta VITRuM nem chega a fornecer dados sobre a evolução dos smells”.

Os participantes foram orientados a localizar um *test smell* e informar a linha e o método em que ocorriam. Após, foram questionados se com as informações que a ferramenta forneceu se conseguiriam corrigi-los (P11). Com o uso da `TSVizzEvolution` por meio da visualização *Timeline View*, todos os participantes informaram que conseguiriam corrigir os *test smells* (as justificativas estão no APÊNDICE Q). O participante Pr15 destacou que: “Sim. A ferramenta apontou para a linha exata em que o problema ocorre, bastaria adicionar uma explicação sobre essa asserção para resolver o problema.”. Porém, a ferramenta `VITRuM` não fornece essa informação.

De acordo com as respostas dos participantes e o tempo para realizar as tarefas pode-se responder a QP2. A `TSVizzEvolution` facilita a visualização dos *test smells* por meio das três técnicas de visualização, sendo que a maioria dos participantes informou que usaria a `TSVizzEvolution` para exibir as ocorrências de *test smells*. Além disso, ela representa a evolução de *test smells* permitindo a comparação de duas versões e informa exatamente o método e a linha onde ocorre o *test smell*, permitindo aos usuários encontrarem facilmente e corrigi-lo, possivelmente melhorando a qualidade do código de teste, o que não é abrangido pela outra ferramenta. Em relação ao tempo para realizar as atividades, pode-se verificar que a diferença no tempo foi irrelevante diante das funcionalidades fornecidas.

5.3.4 QP: Como a abordagem, por meio da ferramenta `TSVizzEvolution`, auxilia os usuários a visualizar *test smells* e tomar decisões de projeto?

De acordo com as respostas a QP1 e a QP2, pode-se responder a QP, lembrando que a abordagem definiu três estratégias visuais. Conforme coletado no experimento, a `TSVizzEvolution` auxilia no processo de tomada de decisão (QP1) ao exibir quem são os possíveis autores de *test smells* (estratégia *TSAuthor*) e ao exibir qual *test smell* com mais ocorrências (estratégia *TSInstant*) permitindo aos envolvidos identificar qual o problema que mais ocorre e orientar a equipe para que prestem atenção aquele *test smell* para corrigi-lo e evitar futuras inserções.

Na QP2, foi possível obter que a `TSVizzEvolution` representa melhor a visualização de *test smells* ao ser comparada com outra ferramenta, utilizando três técnicas de visualização e exibindo detalhamentos não contemplados pela outra ferramenta. Dentre eles, a comparação da evolução e comportamento dos *test smells* (estratégia *TSEvolution*) de duas versões do código de teste e ao fornecer a localização no código onde o *test smell* ocorre.

Sendo assim, a abordagem criada nesse trabalho auxilia os usuários a visualizar *test smells* de forma efetiva e intuitiva.

5.3.5 Considerações dos Participantes

Na execução das tarefas durante ao experimento com ambas ferramentas, todos participantes responderam corretamente as questões de P1 a P11 (no APÊNDICE B, no APÊNDICE D, no APÊNDICE F e no APÊNDICE H). Vale ressaltar que as perguntas P5, P8 e P11 eram discursivas e envolviam a opinião do participante, não tendo resposta correta.

No Quadro 5.7, é apresentada uma visão geral das vantagens e desvantagens das ferramentas `TSVizzEvolution` e `VITRuM`, todos os pontos levantados pelos participantes estão disponíveis no APÊNDICE R e no APÊNDICE S, respectivamente.

Os participantes também tiveram a oportunidade de informaram o que acharam do experimento (APÊNDICE T), a maioria dos participantes informaram que o experimento foi abrangente e bem conduzido. O participante Pr13, por exemplo, disse: "Interessante por mostrar informações relevantes para visualizar os resultados e evolução de um teste.".

Quadro 5.7 - Vantagens e Desvantagens `TSVizzEvolution` e `VITRuM`

Ferramenta	Vantagens	Desvantagens
`TSVizzEvolution`	- Granularidades fornecidas; - Detalhamentos de informações; - Independência de uma IDE.	- Dependência de outra ferramenta (`JNose Test`); - Visualizações da técnica de grafos para a exibição de muitos dados podem ficar poluída; - Usabilidade de ter que selecionar os arquivos.
`VITRuM`	- Simplicidade da interface; - Apresentação das classes em listas simples; - Integrada a uma IDE sendo interessante para quem a utiliza.	- Dependência de uma IDE; - Falta de detalhamento nas análises como informar linhas e métodos; - Não abordar os autores e a evolução de *test smells*.

Fonte: Do autor (2021).

5.4 Ameaças à Validade

Em geral, as ameaças à validade são organizadas em quatro tipos: i) Validade Interna; ii) Validade Externa; iii) Validade de Construção; e iv) Validade de Conclusão (TRAVASSOS *et al.*, 2002):

a) **Validade Interna**. Prejudica o conhecimento do grau em que os resultados da pesquisa refletem a realidade observada. Neste estudo, as ameaças internas são:
   - A forma de elaboração das questões pode levar a diferentes interpretações, interferindo nos resultados obtidos no estudo experimental. Apesar de alguns pesquisadores terem revisado os formulários utilizados no estudo, não é possível garantir que os participantes tiveram a mesma interpretação das questões;
   - As perguntas realizadas no experimento controlado são pontuais, ou seja, foram questionadas informações de *test smells*, classe de teste, método e autores específicos. Essas questões foram definidas para reduzir o esforço dos participantes na execução do experimento. Dessa forma, não é possível garantir que na prática,

seriam obtidos resultados semelhantes. Para contornar essa questão, seria necessário aplicar estudos mais elaborados, em que o participante teria mais tempo para analisar inteiramente as versões do código de teste;

- No experimento controlado duas ferramentas foram utilizadas, não foi informado quem era o autor da ferramenta, porém, caso o participante soubesse poderia comprometer as opiniões dos participantes e o resultado do estudo;

b) **Validade Externa**. Limita a capacidade de generalização dos resultados para contextos fora do ambiente avaliado. Neste estudo, as ameaças externas são:

- Alguns participantes (pesquisadores e estudantes de mestrado/doutorado) do experimento controlado podem não representar adequadamente a população de possíveis usuários reais da abordagem, comprometendo a generalização dos resultados. A aplicação do estudo com diferentes tipos de profissionais pode amenizar essa ameaça;
- A quantidade limitada de questões definidas nos questionários pode não abordar todos os assuntos necessários para avaliação da abordagem;
- Apesar da quantidade total de participantes na avaliação final estar de acordo com (MARSHALL *et al.*, 2013), sendo 16 participantes, a quantidade por grupo de 4 participantes pode ser pequena;

c) **Validade de Construção**. Está relacionada à validação do instrumento de pesquisa, definindo se o tratamento reflete a causa e se os resultados refletem o efeito. Por isso, durante a avaliação da validade de construção, os fatores humanos envolvidos na pesquisa devem ser avaliados e controlados para não interferirem nos resultados obtidos. Neste estudo, as ameaças de construção são:

- O processamento dos dados foi realizado e maneira automatizada através das ferramentas utilizadas no experimento e independente de fatores humanos;
- Podem haver outras perguntas e perspectivas a serem abordados no experimento controlado;
- O experimento controlado foi realizado através dos participantes acessando remotamente o computador do pesquisador e sob leitura do protocolo, podendo haver influência por parte do pesquisador na condução do experimento;
- Para pessoas com deficiências visuais, por exemplo, daltonismo, o entendimento da visualização devido ao uso das cores pode ficar comprometido;

d) **Validade de Conclusão**. Afeta a habilidade de chegar a uma conclusão correta a respeito dos relacionamentos entre o tratamento e o resultado do experimento. Neste estudo, as ameaças de conclusão são:
   - A falta de testes por outras pessoas compromete a eficácia da ferramenta;
   - A quantidade limitada de participantes envolvidos no experimento controlado pode comprometer a obtenção de conclusões significativas. Para contornar essa questão, pode-se repetir o estudo experimental com maior quantidade de pessoas;

5.5 Considerações Finais

Neste capítulo, foi apresentada a avaliação da abordagem utilizando a ferramenta proposta neste estudo denominada `TSVizzEvolution`. A avaliação foi realizada por meio de um estudo experimental. Inicialmente, foi realizada uma avaliação piloto com 4 participantes para ajustes e melhorias para a realização da avaliação final. Na avalição final, 16 participantes integraram o estudo, onde realizaram tarefas e responderam 11 questões para cada uma das duas ferramentas utilizadas no estudo, `TSVizzEvolution` e `VITRuM`. Ao final do experimento, os participantes responderam um questionário de *feedback*.

Os resultados mostram que no geral a ferramenta `TSVizzEvolution` apresentou facilidades na visualização de *test smells*, ao fornecer detalhamentos que não são contemplados na outra ferramenta do estado da arte. Assim, a abordagem crida pode ajudar nas decisões de projeto e nos processos de tomada de decisão, conforme afirmado pelos participantes.

6 TRABALHOS RELACIONADOS

Alguns trabalhos realizam a detecção de *test smells*, mas não utilizam recursos visuais. Outros trabalhos utilizam a visualização de software para evolução de software, porém não no contexto de *test smells*. Foram encontrados apenas dois trabalhos que utilizam recursos visuais para *test smells*.

6.1 Detecção e Visualização de *Test Smells*

A detecção de *test smells* foi abordada em diversos estudos. Por exemplo, em um estudo (TUFANO *et al*., 2016), foram analisados 5 *test smells* perante 19 desenvolvedores para identificar quando eles foram introduzidos, quanto tempo permanecem e se eles estão relacionados ao *smells* do código de produção. Em um estudo (SANTANA *et al.,* 2020), foi apresentada a ferramenta RAIDE, um *plug-in* para a IDE Eclipse. A RAIDE detecta dois *test smells* (*Assertion Roulette* e *Duplicate Assert*) e permite a refatoração automatizada para software Java que tenha suas classes de teste geradas com o *framework* com JUnit.

ALJEDAANI *et al.,* 2021 realizaram um mapeamento sistemático de ferramentas para a detecção de *test smells*. Realizaram uma análise comparativa com os *test smells* detectados por cada ferramenta, linguagem de programação, estratégia de detecção e outros recursos, foram identificadas 22 ferramentas.

No trabalho de CAMPOS *et al.,* 2021, foi analisado como os desenvolvedores percebem a gravidade dos *test smells* inseridos por eles. A gravidade refere-se ao grau em que um *test smell* pode afetar negativamente o código de teste. Foram selecionados 6 projetos de software de código aberto do *GitHub* e seus desenvolvedores foram entrevistados. Embora a maioria dos desenvolvedores entrevistados considerou os *test smells* como tendo baixa severidade, eles indicaram que podem impactar negativamente o projeto, particularmente na manutenção e na evolução do código de teste.

GREILER *et al*., 2013 propuseram uma ferramenta (`TestHound`) que fornece um relatório de *test smells* e recomendações de refatoração para resolvê-los. Em outro trabalho (BAVOTA *et al*., 2012), a distribuição de *test smells* no código de teste foi verificada. Foram analisados 18 projetos de software (2 industriais e 16 abertos). Os resultados mostraram que apenas 18% das classes de testes não foram afetadas, ou seja, 82% foram afetadas por, pelo menos, um *test smell*.

Em outro trabalho (GAROUSI *et al*., 2018), foi realizada uma pesquisa na literatura sobre *test smells* e sobre ferramentas para detectá-los. Duas das 10 ferramentas encontradas

utilizam recursos visuais (TecReVis e TestQ). O foco de TecReVis é mostrar a cobertura e a redundância de teste, gerando uma matriz de correlação para comparar os casos de teste e essa visualização basicamente é usada para comparar casos de teste entre si. TestQ apresenta duas formas visuais para a existência de 12 *test smells* (Breugelmans; ROMPAEY, 2008).

SANTANA *et al.*, 2021 realizaram um estudo com 87 testadores para entender suas visões sobre 8 *test smells*. Além de terem capturado as visões dos testadores sobre os *test smells*, foi identificado que a maioria dos participantes utiliza estratégias manuais para criar e manter os casos de teste. Esse estudo contribui com possíveis direções e tratamentos para analisar os *test smells*, buscando explicar como os *test smells* afetam o trabalho dos testadores.

Em uma outra abordagem (ROMPAEY *et al*., 2006), foi verificada, com estudo de caso de um sistema de código aberto, a significância relativa de 2 *test smells* de acordo com critérios de teste de unidade e propuseram uma abordagem heurística baseada em medidas para classificar os *test smells*. MARTINS *et al.*, 2021 realizaram um estudo baseado em técnicas de aprendizado de máquina para dar sugestões de refatoração de *test smells*. Como resultado, propuseram um *framework* que auxilia desenvolvedores na tomada de decisão quanto à refatoração *de test smells.*

De modo geral, os estudos e as ferramentas focam na detecção e na identificação da natureza dos *test smells*. A ferramenta que utiliza recurso visual para *test smells* não apresenta dados referente a evolução e autoria (TestQ), além disso abrange 7 *test smells*. O diferencial da proposta é abranger a visualização de 21 *test smells* e apoiar a visualização da evolução e autoria de *test smells*.

6.2 Visualização e Evolução de Software

Em um trabalho (LUNGU; LANZA, 2007) foi apresentada a ferramenta Softwarenaut, que utiliza uma técnica de visualização denominada *Edgen Evolution Fimstrip*, apresentando a evolução das relações intermódulo por meio de múltiplas versões do sistema. Em outro estudo (BASTOS *et al.*, 2016), foi apresentado um *plug-in* para a IDE Eclipse denominado *DCEVizz*. Essa ferramenta utiliza uma técnica de visualização concebida com base nas técnicas *Treemap* e *Matriz de Evolução* para representar código morto na evolução de sistemas de software Java. A ferramenta detecta o código morto (métodos não invocados) e representa-os na forma de retângulos aninhados ou na forma de um gráfico de linha, não há limite quanto à quantidade de versões que podem ser analisadas.

Outros trabalhos utilizaram metáforas do mundo real para representar visualmente a evolução do software, como as ferramentas CodeForest e CodeCity. CodeForest

utiliza uma floresta para representar a estrutura, o comportamento e a evolução de sistemas de software desenvolvidos em Java. Cada árvore representa uma classe no sistema de software, cada galho de uma árvore representa um método e a quantidade de galhos é igual a quantidade de métodos da classe. Essa ferramenta apoia o cálculo de 17 medidas que podem ser mapeadas pelo usuário, por exemplo, altura do tronco pode representar NOC (*Number of Classes*) (MARUYAMA *et al*., 2014). `CodeCity` representa o software em forma de cidade. Os edifícios representam interfaces e azulejos representam os pacotes. A saturação de cor é proporcional ao nível de sobreposição dos pacotes correspondentes. A altura dos edifícios representa a quantidade de métodos, enquanto a largura e o comprimento representam a quantidade de atributos (WETTEL; LANZA, 2007; CASERTA; ZENDRA, 2011).

Em outro estudo (CASERTA; ZENDRA, 2011), foi proposta a ferramenta `RelVis` que exibe a evolução do sistema de software no que diz respeito as medidas de software. Essa técnica utiliza gráficos de radar para ilustrar os valores dessas medidas. A visualização mostra as medidas de várias versões do sistema de software e diferentes cores são utilizadas para representar as versões. As larguras das arestas representam o acoplamento entre duas classes. Essa visualização contém vasta quantidade de informação e pode ser útil para a identificação de acoplamentos.

Em outro trabalho (PECORELLI *et al*., 2020), foi apresentado um *plug-in* para o IntelliJ denominado `VITRuM` que fornece aos desenvolvedores uma interface relacionada ao código de teste, medidas estruturais, *test smells* e medidas dinâmicas, como indicadores de cobertura de código. Os dados são apresentados em um gráfico para acompanhar a evolução.

De modo geral, os estudos e as ferramentas focam na exibição visual da evolução do software de código de produção. A ferramenta que utiliza recurso visual para evolução, apresenta a evolução das medidas e não dos *test smells*, além de não abordar a autoria (`VITRuM`) e abranger 12 *test smells*. O diferencial da proposta é abranger a visualização de 21 *test smells* e exibir a evolução e autores de *test smells*.

6.3 Ferramentas Relacionadas

Para analisar a eficácia de uma ferramenta de visualização, é necessário verificar sua forma de representar as informações (Carneiro *et al.,* 2008), mas, também pode-se compará-la a outras ferramentas disponíveis. Por esse motivo, no Quadro 6.1, são apresentadas 14 ferramentas que refere-se a testes, com nome, descrição e endereço eletrônico. Apenas as duas ferramentas assinaladas com "***" utilizam recursos visuais para *test smells*. Os campos preenchidos com "-" indicam que a informação não foi disponibilizada pelos autores.

Muitas ferramentas lidam com a visualização de informações para diferentes propósitos, por exemplo, visualizar a evolução do código morto (BASTOS *et al*., 2016), mostrando uma estrutura de software (CASERTA; ZENDRA, 2011) e software de teste (GAROUSI *et al*., 2018). No entanto, para representação de *test smells*, a literatura ainda apresenta poucas propostas.

Quadro 6.1 - Ferramentas para *Test Smells*

Ferramenta	Descrição	Endereço Eletrônico
TRex	Ferramenta para detecção e refatoração de *smells* para *TTCN*[1]	www.trex.informatik.uni-goettingen.de/trac
TeReDetect	Ferramenta para detecção de redundância de teste para o JUnit	www.codecover.org
TestLint	Detecção automática de 49 *test smells* para o NUnit[2]	www.typemock.com/test-lint
TestHound	Detecção automática de *smells* de *suítes* de teste para JUnit	https://github.com/SERG-Delft/TestHound
OraclePolish	Ferramenta para melhorar a qualidade do oráculo, detectando afirmações frágeis	www.bitbucket.org/udse
TestLint	Detecção baseada em regras de *test smells* para o JUnit	scg.unibe.ch/wiki/alumni/stefanreichhart/testsmells
TecReVis	Ferramenta para visualização de cobertura e redundância de teste	www.codecover.org
Unnamed	Detecção de teste duplicado	www.bitbucket.org/felixfangzh/similartestanalysis
**TestQ	Apresenta a associação entre *test smells* e os métodos das classes de teste usando diferentes visualizações	code.google.com/archive/p/tsmells
An extension to TRex	Detectar *smells* em *suítes* de teste *TTCN-3*[3]	www.trex.informatik.uni-goettingen.de/trac
**VITRuM	Apresenta uma correlação entre os *test smells* e métricas de código	https://plugins.jetbrains.com/plugin/14160-vitrum
tsDetect	Detecção de 21 *test smells*	https://testsmells.github.io/index.html
JNose Test	Estende tsDetect, realiza a detecção de 21 *test smells* apresenta os resultados em arquivos csv	https://github.com/ariesla b/jnose
RAIDE	Refatoração automática de 2 *test smells*	https://github.com/ariesla b/raide
TestEvoHound	Melhoramento da ferramenta TestHound	-

(continua)

[1] Testing and Test Control Notation: linguagem de programação usada para testar protocolos de comunicação e serviços da web.
[2] https://nunit.org/
[3] Testing and Test Control Notation Version 3 (http://www.ttcn-3.org/)

Quadro 6.1 - Ferramentas para *Test Smells* (continuação)

Ferramenta	Descrição	Endereço Eletrônico
Taste	Detecção de *smells* durante a execução do teste (por exemplo, teste de leitura/escrita de recursos compartilhados)	-
DTDetector	Detecção do *test smell Dependent Test* para o JUnit	ttps://github.com/winglam/dtdetector
EletricTest	Detecção de *test smell Dependent Test* para o JUnit, melhoramento da DTDetector	-
PraDet	Detecta redundância de testes	https://github.com/gmu-swe/pradet-replication
SoCRATES	Detecta a presença de 6 *test smells* na linguagem Scala usando análise estática	https://github.com/jonas-db/socrates
TEDD	Detecta rdedundância de teste para sistemas *web*	https://github.com/matteobiagiola/FSE19-submission-material-TEDD
DARTS	*Plugin* que utiliza a recuperação de informações para detectar 3 *test smells*	https://github.com/StefanoLambiase/DARTS
PolDeT	Detecção de *smells* no código de teste	-
RTj	Detecção e refatoração do *smell Rotten Green Test*[1]	https://github.com/UPHF/RTj
DrTest	Detecção do *smell Rotten Green Test*	https://github.com/juliendelplanque/DrTests

Fonte: Adaptado Garousi *et al*. (2018), Aljedaani *et al*. (2021)

6.3.1 TestQ

A ferramenta TestQ apoia várias maneiras de visualizar dados para explorar a estrutura do código de teste e analisar/quantificar a existência de 12 *test smells* para a linguagem C++ (BREUGELMANS; ROMPAEY, 2008). Essa ferramenta apresenta dois recursos:

a) ***Test Suite Topology***: permite explorar a estrutura dos casos de teste, utilizando três visualizações (Figura 6.1):
   - *Expandable Suite Tree*: exibe os casos e métodos de teste utilizando uma árvore vertical;
   - *Polymetric Suite View*: representa os casos de teste, utilizando três medidas representadas respectivamente pela altura, largura e coloração: i) quantidade de

[1] *Test smell* que refere a testes que foram planejados para executar algumas asserções, mas que na verdade não o fazem (Delplanque *et al.*, 2019).

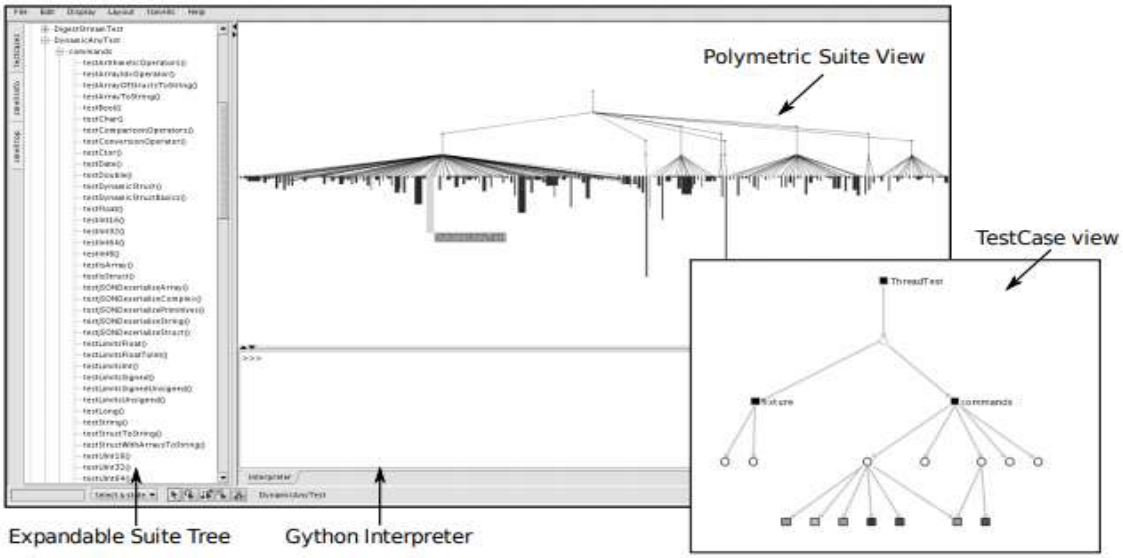
comandos no caso de teste; ii) SLOC (*Source lines of code*) dividido pela quantidade de comandos; e iii) presença de métodos auxiliares;

- *Test Case View*: representação hierárquica de um único caso de teste. Os *test smells* são mostrados como nós separados, ligado ao método de origem ou caso de teste e coloridos de acordo com seu tipo. Ao passar o *mouse* sobre o nó, são apresentadas algumas informações adicionais como valores as medidas;

b) ***Test Smells Detection***: Esse conjunto de visualizações revela *test smells* em nível de código. Cada *test smell* é representado como um nó, conectado a entidade de teste impactada. Ao passar o *mouse* sobre o nó, informações detalhadas são fornecidas (nome da entidade que o possui, quantidade da linha e medidas):

- *Smell Flower View*: coleção de grafos que mostram os casos de teste separados por "*flower*". O nó central da flor representa o próprio caso de teste, cercado por seus métodos de testes e as instâncias de *test smells*. Por padrão, o conjunto completo e todos os 12 *test smells* são mostrados, o que resulta em uma visão "cheia", as opções interatividade, como o *zoom*, permitem alternar entre os casos de teste, geral ou específico;
- *Smell Pie*: mostra a proporção dos *test smells* em um gráfico de pizza. A seleção de um *test smell* no gráfico destaca todas as ocorrências.

Figura 6.1 - Visualização com `TestQ` opção "*Test Suite Topology*"



Fonte: Breugelmans; Rompaey (2008).

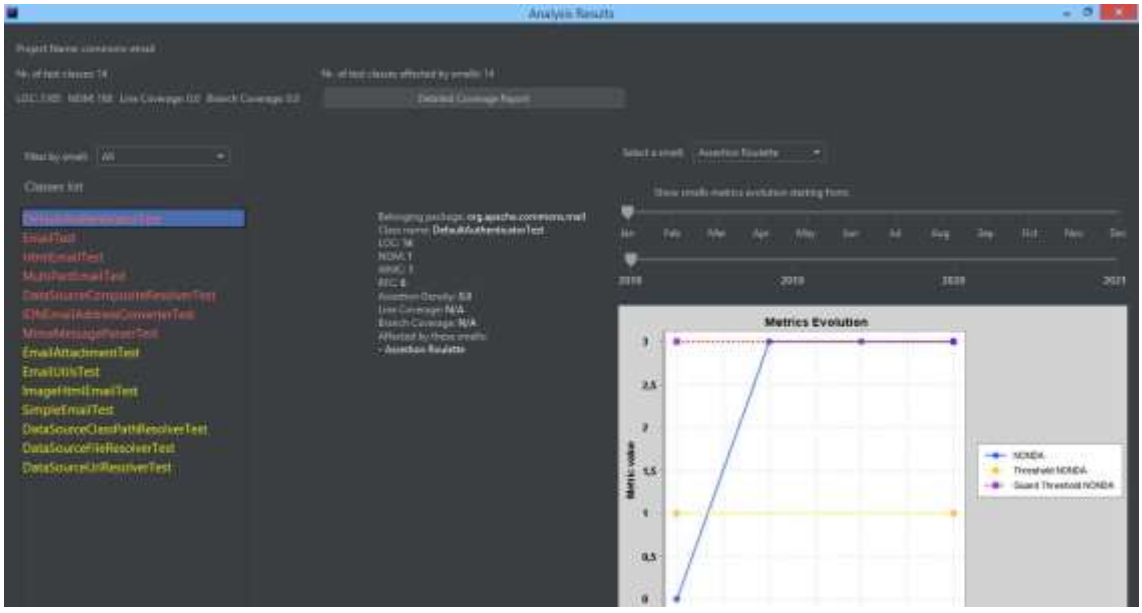
6.3.2 VITRuM

A ferramenta VITRuM (*VIsualization of Test-Related Metrics*) é um *plug-in* para o IntelliJ para código Java que inclui o cálculo de medidas de código de teste e detecta 7 *test smells (*PECORELLI *et al*. 2020). As medidas dividem-se em dois tipos: i) estruturais, referem ao tamanho, à coesão, ao acoplamento e à complexidade; e ii) dinâmicas, exibem informações sobre a eficácia dos testes e cobertura de testes. Os *test smells* são exibidos por meio de uma lista com as classes de teste em cores diferentes: i) vermelho, para classes de teste afetadas por *test smells*, com valores de medidas de código de teste acima dos limites definidos; ii) amarelo, para classes de teste afetadas por *test smells*, com valores de medidas de código de teste dentro dos limites definidos; e iii) cinza, para classes de teste sem *test smells*.

Além disso, os usuários podem filtrar os resultados por *test smell* para exibir sua evolução em relação as medidas em um gráfico durante o desenvolvimento do projeto (Figura 6.2). A seguir são detalhadas as medidas e os *test smells* que a ferramenta abrange:

a) **Medidas estruturais:** LOC (*Line of Code*), AD (*Assertion Density*) e WMC (*Weigthed Methods Per Class*);
b) **Medidas dinâmicas:** *Line Coverage, Branch Coverage, Mutation Coverage e Flaky Tests;*
c) ***Test Smells:*** *Assertion Roulette, Eager Test, General Fixture, Indirect Testing, Sensitive Equality, Mystery Guest e Resource Optimism.*

Figura 6.2 - VITRuM Painel Principal



Fonte: Pecorelli *et al*. (2020).

No Quadro 6.2, são apresentadas as ferramentas que utilizam recursos visuais para *test smells*. São exibidos os recursos que fornecem referente a visualização de *test smells* e a

quantidade de *test smells* abordados. Os campos assinalados com “x” representam que atende ao recurso e aqueles assinalados com “-” não atendem.

Quadro 6.2 - Ferramentas para Visualização de *Test Smells*

Recursos	**`TestQ`**	**`TSVizzEvolution`**	**`VITRuM`**
Visualização de ocorrências de *test smells*	x	x	x
Visualização de possíveis autores de *test smells*	-	x	-
Visualização da evolução de *test smells*	-	x	-
Quantidade de *test smell*s abordados	12	21	7

Fonte: Do Autor (2022).

De modo geral, há diversas ferramentas que representam características de forma visual do código de produção, pois a utilização de recursos visuais, no geral, facilita o entendimento. Apenas duas ferramentas utilizam a visualização para *test smells*. Porém, nenhuma das ferramentas encontradas abordam a evolução e autoria de *test smells*, além de abrangerem uma quantidade menor de *test smells* do que a ferramenta proposta nesse tabalho.

7 CONSIDERAÇÕES FINAIS

Test smells podem tornar a execução dos testes mais dispendiosa e afetar a sua qualidade. Ao longo da evolução do código de teste, novos *test smells* podem surgir e o aumento dessse problemas podem dificultar a manutenção do código de teste. Desse modo, ferramentas de detecção de *test smells* foram propostas, algumas ferramentas utilizam recursos visuais para facilitar a compreensão desses problemas. No entanto, existe carência para a análise visual das ocorrências de *test smells* durante a evolução do software e seus possíveis autores.

Com base nesses fatores, neste trabalho, foi proposta uma abordagem para a representação de *test smells* por meio de diferentes técnicas de visualização. Essa abordagem representa características dos *test smells*, como a quantidade de ocorrências, a classe e o método onde ocorrem e seus possíveis autores durante a evolução na análise de até duas versões do código de teste. A abordagem foi implementada em uma ferramenta denominada `TSVizzEvolution`. De modo geral, o objetivo foi facilitar a visualização dos *test smells* para proporcionar entendimento mais rápido das suas características motivando os envolvidos a possivelmente removê-los.

7.1 Conclusão

A abordagem proposta neste estudo foi avaliada com um estudo empírico e os resultados foram apresentados. No estudo final, 16 participantes integraram o estudo experimental, onde realizaram tarefas e responderam 11 questões para cada ferramenta utilizada, sendo `TSVizzEvolution` e `VITRuM`. Além disso, os participantes responderam a um questionário final para caracterização dos participantes e *feedback* do experimento.

Os resultados obtidos com as opiniões dos participantes ressaltam que a `TSVizzEvolution` exibe melhor detalhamento que a `VITRuM`. Além disso, pode auxiliar nas decisões de projeto e nos processos de tomada de decisão ao proporcionar informações sobre os possíveis autores dos *test smells*. No geral, a abordagem e a sua implementação na ferramenta mostraram abrangentes e suficientes para facilitar a visualização dos *test smells*, servindo de auxílio para os envolvidos nas atividades de teste a visualizar e possivelmente corrigir os *test smells* para consequentemente melhorar a qualidade do código de teste.

7.2 Contribuições

A condução desta pesquisa contribuiu para a evolução do conhecimento na área de Ciência da Computação, especificamente na área de Engenharia de Software. As principais contribuições proporcionadas foram:

a) Classificação de três estratégias visuais para análise e visualização de *test smells*;
b) A ferramenta `TSVizzEvolution.exe` automatiza a visualização de *test smells;*
c) Identificação da necessidade de ferramentas que auxiliem nas decisões de projeto e nos processos de tomada de decisão quanto a *test smells.*

7.3 Perspectivas Futuras

Algumas sugestões de trabalhos futuros, que podem contribuir no aprimoramento, evolução e na avaliação da abordagem proposta nesse trabalho:

a) Acompanhar a evolução da saída da ferramenta `JNose Test` para a ferramenta `TSVizzEvolution`.
b) Aumentar a abrangência da avaliação para melhorar os dados que representam a experiência do usuário;
c) Realizar estudos empíricos para entender o comportamento dos *test smells* e seus malefícios durante a evolução de software;
d) Acoplar a ferramenta `TSVizzEvolution` a ferramenta `JNose Test`, convertendo-a para *web*.

7.4 Publicações

Trabalhos publicados durante essa pesquisa de mestrado:

a) Trabalho publicado como primeira autora:
   - CRUZ, A; COSTA, H. Uma Abordagem Visual para Evolução de Test Smells em Sistemas de Software Java. In: Workshop de Teses e Dissertações do CBSoft. Anais Estendidos do XI Congresso Brasileiro de Software: Teoria e Prática. p. 63-69. 2020.
b) Trabalhos publicados como co-autora:
   - SANTANA, R. S.; MARTINS, L. A.; VIRGINIO, T. G. A.; CRUZ, A. P. S.; SOARES, L. R.; Costa, H. A. X.; MACHADO, I. C. RAIDE: a tool for Assertion Roulette and Duplicate Assert identification and refactoring. In: Simpósio Brasileiro de Engenharia de Software. p. 374-379, 2020.

- VIRGINIO, T. G. A.; MARTINS, L. A.; SOARES, L. R.; SANTANA, R. S.; CRUZ, A. P. S.; COSTA, H. A. X.; MACHADO, I. C. JNose: Java Test Smell Detector. In: Simpósio Brasileiro de Engenharia de Software. p. 564-569, 2020.
- VIRGINIO, T., MARTINS, L., SANTANA, R., CRUZ, A., ROCHA, L., COSTA, H., MACHADO, I. On the test smells detection: an empirical study on the jnose test accuracy. Journal of Software Engineering Research and Development. V9. p. 8-1, 2021.

REFERÊNCIAS

ALJEDAANI, W.; PERUMA, A.; ALJOHANI, A.; ALOTAIBI, M.; MKAOUER, M. W.; OUNI, A.; LUDI, S. **Test smell detection tools: A systematic mapping study**. In: Evaluation and Assessment in Software Engineering. p.170-180, 2021.

BASTOS, C.; AFONSO, P. J.; COSTA, H. **DCEVizz: Uma Ferramenta para Visualização de Código Morto na Evolução de Sistemas de Software Java**. In: Simpósio Brasileiro de Engenharia de Software, 2016.

BASTOS, C.; COSTA, H. **Uma Abordagem para Visualização da Evolução de Código Morto em Sistemas de Software Orientados a Objetos**. In: Workshop de Teses e Dissertações em Qualidade de Software. p.33, 2016.

BAVOTA, G.; QUSEF, A.; OLIVETO, R.; DE LUCIA, A.; BINKLEY, D. **An empirical analysis of the distribution of unit test smells and their impact on software maintenance**. In: International Conference on Software Maintenance. p. 56-65, 2012.

BAVOTA, G.; QUSEF, A.; OLIVETO, R.; DE LUCIA, A.; BINKLEY, D. **Are test smells really harmful? an empirical study**. In: Empirical Software Engineering. p. 1052-1094, 2015.

BREUGELMANS, M.; VAN ROMPAEY, B. **Testq: Exploring structural and maintenance characteristics of unit test suites**. In: International Workshop on Advanced Software Development Tools and Techniques, 2008.

CAMPOS, D.; ROCHA, L.; MACHADO, I. **Developers perception on the severity of test smells: an empirical study**. arXiv preprint arXiv:2107.13902. 2021.

CARD, S. K.; MACKINLAY, J. D.; SHNEIDERMAN, B. **Readings in information visualization: using vision to think**. 671p. Ed. Morgan Kaufmann, 1999.

CARNEIRO, G.; MENDONCA, M.; MAGNAVITA, R. **An experimental platform to characterize software comprehension activities supported by visualization**. In: International Conference on Software Engineering. p. 441-442, 2009.

CALEGARE, A. J. **Introdução ao delineamento de experimentos**. Editora Blucher, 2009.

CARPENDALE, S.; GHANAM, Y.; **A Survey Paper on Software Architecture Visualization**, University of Calgary. 2008.

CASERTA, P.; ZENDRA, O. **Visualization of the Static Aspects of Software: A Survey**. Revista Visualization and Computer Graphics, v. 17, n. 7, p. 913-933, 2011.

CONCEIÇÃO, C. F. R; CARNEIRO, G.; DAVID, J. M. N. **Usando Recursos de Visualização Enriquecidos com Elementos de Percepção para a Compreensão de Software em um Ambiente de Desenvolvimento Distribuído**. In: Workshop Brasileiro de Visualização de Software. p. 41-48, 2012.

CRESPO, A. N.; SILVA, O. J.; BORGES, C. A.; SALVIANO, C. F.; ARGOLLO, M.; JINO, M. **Uma metodologia para teste de Software no Contexto da Melhoria de Processo. Simpósio Brasileiro de Qualidade de Software**. p. 271-285. 2004.

CRUZ, A.; BASTOS, C.; AFONSO, P.; COSTA, H. **Software visualization tools and techniques: A systematic review of the literature**. In: International Conference of the Chilean Computer Science Society. p. 1-12, 2016.

DIEHL, S. **Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software**. Berlim: Springer. 186p. 2007

EICHBERG, M.; HAUPT, M.; MEZINI, M.; SCHAFER, T. **Comprehensive software understanding with SEXTANT**. In: International Conference on Software Maintenance. p. 315-324, 2005.

GAROUSI, V.; KUCUK, B.; FELDERER, M. **What we know about smells in software test code**. In: IEEE Software. p. 61-73, 2018.

GRECHANIK, M; XIE, Q; FU, C. **Maintaining and evolving GUI-directed test scripts**. In: International Conference on Software Engineering. p. 408-418, 2009.

GREILER, M.; VAN DEURSEN, A.; STOREY, M. A. **Automated detection of test fixture strategies and smells**. In: International Conference on Software Testing, Verification and Validation. p. 322-331, 2013.

JUNIOR, N. S.; MARTINS, L.; COSTA, H.; MACHADO, I. **How are test smells treated in the wild? A tale of two empirical studies**. In: Journal of Software Engineering, v.9, p. 9, 2021.

JUNIOR, N.S.; ROCHA, L.; MARTINS, L.A; MACHADO, I. **A survey on test practitioners' awareness of test smells**. arXiv preprint arXiv:2003.05613. 2020.

KAPEC, P. **Visualizing software artifacts using hypergraphs**. In: Spring Conference on Computer Graphics. p. 27-32, 2010.

KIM, D. J. **An Empirical Study on the Evolution of Test Smell**. 2019. Disponível em: https://djaekim.github.io/djae.io/img/EvolutionOfTestSmell.pdf. Acesso em: 02 de abril de 2020.

KOOCHAKZADEH, N.; GAROUSI, V. **TecReVis: a tool for test coverage and test redundancy visualization.** In: International Academic and Industrial Conference on Practice and Research Techniques. p. 129-136, 2010.

KOSCHKE, R. **Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey**. Journal of Software Maintenance and Evolution: Research and Practice. p. 87-109, 2003.

LOZADA, L. **A-Graph: Uma ferramenta computacional de suporte para o ensino-aprendizado da disciplina Teoria dos Grafos e seus Algoritmos**. In: Workshop do Congresso Brasileiro de Informática na Educação. v. 3, p. 61, 2014.

LÜ, H.; FOGARTY, J. **Cascaded TreeMaps: Examining the Visibility and Stability of Structure in TreeMaps**. In: Conference of Graphics Interface. p. 259-266, 2008.

LUNGU, M.; LANZA, M. **Exploring Inter-Module Relationships in Evolving Software Systems**. In: European Conference on Software Maintenance and Reengineering. p. 91-102, 2007.

MARSHALL, B. *et al.* **Does Sample Size Matter in Qualitative Research?: A Review of Qualitative Interviews in is Research**. Journal of Computer Information Systems, v. 54, n.1, p. 11-22, 2013.

MARTINS, L.; BEZERRA, C.; COSTA, H.; MACHADO, I. **Smart prediction for refactorings in the software test code**. In: Brazilian Symposium on Software Engineering. p. 115-120, 2021.

MARUYAMA, K.; OMORI, T.; HAYASHI, S. A. **Visualization Tool Recording Historical Data of Program Comprehension Tasks**. In: International Conference on Program Comprehension. p. 207-211, 2014.

MESZAROS, G. **xUnit test patterns: Refactoring test code**. Pearson Education, 2007.

NOVAIS, L. R.; NUNES, C.; GARCIA, A.; MENDONÇA, M. **Sourceminer Evolution: A Tool for Supporting Feature Evolution Comprehension**. In: International Conference on Software Maintenance. p. 508-511, 2013.

NOVAIS, R.; JUNIOR, P. S.; MENDONÇA, M. **Timeline matrix: an on demand view for software evolution analysis**. In Proc. SV Workshop, 2012.

PALOMBA, F.; BAVOTA, G.; DI PENTA, M.; OLIVETO, R.; DE LUCIA, A.; **Poshyvanyk, D. Detecting bad smells in source code using change history information**. In: International Conference on Automated Software Engineering. p. 268-278. 2013.

PALOMBA, F.; DI NUCCI, D.; PANICHELLA, A.; OLIVETO, R.; LUCIA, A. **On the diffusion of test smells in automatically generated test code: An empirical study**. In: International Workshop On Search-Based Software Testing. p. 5–14. 2016.

PALOMBA, F.; ZAIDMAN, A. **Does refactoring of test smells induce fixing flaky tests?. In: International conference on software maintenance and evolution.** p. 1-12, 2017.

PALOMBA, F.; ZAIDMAN, A.; DE LUCIA, A. **Automatic test smell detection using information retrieval techniques**. In: International Conference on Software Maintenance and Evolution. p. 311-322, 2018.

PECORELLI, F.; LILLO, G.; PALOMBA, F.; LUCIA, A.; **VITRuM-A Plug-In for the Visualization of Test-Related Metrics**. In: International Conference on Advanced Visual Interfaces. p. 1-3, 2020.

PERUMA, A.; ALMALKI, K.; NEWMAN, C. D.; Mkaouer, M. W.; Ouni, A.; Palomba, F. **On the distribution of test smells in open source Android applications: an exploratory**

study. In: International Conference on Computer Science and Software Engineering. p. 193-202, 2019.

PERUMA, A.; **What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications**. Ph.D. Thesis. Rochester Institute of Technology, Rochester, New York. 2018.

PORTO, D.; MENDONCA, M.; FABBRI, S. **CRISTA: A tool to support code comprehension based on visualization and reading technique**. In: International Conference on Program Comprehension. p. 285-286, 2009.

PRODANOV, C. C.; DE FREITAS, E. C. **Metodologia do Trabalho Científico: Métodos e Técnicas da Pesquisa e do Trabalho Acadêmico** - 2ª Edição. Editora Feevale, 2013.

SANTANA, R. S.; MARTINS, L. A.; VIRGINIO, T. G. A.; CRUZ, A. P. S.; Soares, L. R.; COSTA, H. A. X.; MACHADO, I. C. **RAIDE: a tool for Assertion Roulette and Duplicate Assert identification and refactoring**. In: Simpósio Brasileiro de Engenharia de Software. p. 374-379, 2020.

SANTANA, R. S.; **RAIDE: uma abordagem semi-automatizada para identificação e refatoração de test smells**. Disponível em: http://repositorio.ufba.br/ri/handle/ri/33621. 2021.

SANTANA, R.; FERNANDES, D.; CAMPOS, D.; SOARES, L.; MACIEL, R.; MACHADO, I. **Understanding practitioners' strategies to handle test smells: a multi-method study**. In: Brazilian Symposium on Software Engineering. p. 49-53, 2021.

SILVA, M. S. **Construindo sites com CSS e (X) HTML: sites controlados por folhas de estilo em cascata**. Novatec Editora, 2007.

SILVA, R. O.; MACHADO, G. B. G.; VIANA, G. B.; DOS SANTOS SILVA, J. S. **O Processo de Teste de Software**. In: Tecnologias em Projeção. v. 7, 2016.

SPADINI, D.; PALOMBA, F.; ZAIDMAN, A.; BRUNTINK, M.; BACCHELLI, A. **On the relation of test smells to software code quality**. In: International Conference on Software Maintenance and Evolution. p. 1-12, 2018.

STOREY, M.A.; BEST, C.; MICHAUD, J.; RAYSIDE, D.; LITOIU, M.; MUSEN, M**. SHriMP views: An interactive environment for information visualization and navigation**. Conference on Human Factors in Computing Systems. p. 520 - 521, 2002.

SULAIMAN, S. **Viewing Software Artifacts for Different Software Maintenance Categories Using Graph Representations**. Journal of Science. v. 17, p. 55-67, 2004.

SWING. **Swing (Java™ Foundation Classes)**. Disponível em: <http://docs.oracle.com/javase/7/docs/technotes/guides/swing/>. Acesso em: Junho 2020.

TAHIR, A.; COUNSELL, S.; MACDONELL, S. G. **An empirical study into the relationship between class features and test smells**. In: Asia-Pacific Software Engineering Conference. p. 137-144, 2016.

TRAVASSOS, G. H.; GUROV, D.; AMARAL, E. A. G. **Introdução à Engenharia de Software Experimental**. 2002. Relatório Técnico - PESC-COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2002.

TUFANO, M.; PALOMBA, F.; BAVOTA, G.; DI PENTA, M.; OLIVETO, R.; DE LUCIA, A.; POSHYVANYK, D. **An empirical investigation into the nature of test smells**. In: International Conference on Automated Software Engineering, p. 4-15, 2016.

VAN DEURSEN, A.; MOONEN, L.; VAN DEN BERGH, A.; KOK, G. **Refactoring test code. In Conference on extreme programming and flexible processes in software engineering**. p. 92-95, 2001.

VAN ROMPAEY, B.; DU BOIS, B.; DEMEYER, S. **Characterizing the relative significance of a test smell**. In: International Conference on Software Maintenance p. 391-400, 2006.

VIRGINIO, T. G. A.; MARTINS, L. A.; SOARES, L. R.; SANTANA, R. S.; CRUZ, A. P. S.; COSTA, H. A. X.; MACHADO, I. C. **JNose: Java Test Smell Detector**. In: Simpósio Brasileiro de Engenharia de Software. p. 564-569, 2020.

VIRGÍNIO, T.; SANTANA, R.; MARTINS, L. A.; SOARES, L. R.; COSTA, H.; MACHADO, I. **On the influence of Test Smells on Test Coverage**. In: Brazilian Symposium on Software Engineering. p. 467-471, 2019.

WETTEL, R.; LANZA, M. **Visual exploration of large-scale system evolution**. In: Working Conference on Reverse Engineering. p. 219-228, 2008.

APÊNDICE A

ADRIANA PRISCILA SANTOS CRUZ

Protocolo do Estudo Experimental com Ferramentas para Visualização de *Test Smells*

LAVRAS-MG
2021

Sumário

1 EXPERIMENTO CONTROLADO

As informações são lidas, com a finalidade de manter o padrão, pois trata-se de um estudo experimental e o treinamento com todos os participantes deve ser o mais semelhante. São explicados os conceitos de *test smells* e as ferramentas para a visualização de *test smells* utilizadas nesse estudo. Posteriormente, são realizadas as tarefas propostas nesse experimento.

2 *DESIGN* DO ESTUDO

O objetivo desse estudo é visualizar a ocorrência e a evolução de *test smells*. Sendo conduzido na seguinte ordem:

a) ***Test Smells*:** São explicados os conceitos e exemplificados dois *test smells*;
b) **Ferramentas para a Visualização de *Test Smells*:** São explicadas sobre duas ferramentas para a visualização de *test smells;*
c) **Treinamento 1.** `TSVizzEvolution` ou `VITRuM – Commons IO;`
d) **Tarefa 1.** `TSVizzEvolution` ou `VITRuM – Commons Email ou Commons Text;`
e) **Treinamento 2.** `TSVizzEvolution` ou `VITRuM - Commons IO;`
f) **Tarefa 2.** `TSVizzEvolution` ou `VITRuM - Commons Text` ou `Commons Email`;
g) **Aplicar questionário.** Questionário para caracterização dos participantes e *feedback* do experimento.

3 *TEST SMELLS*

Os *test smells* são considerados decisões ou práticas ruins de design no código de teste, os quais diminuem a qualidade dos testes e tornam os testes mais difíceis de serem compreendidos e mantidos durante a evolução dos sistemas de software. Dois *test smells* são explicados a seguir:

a) ***Assertion Roulette* (AR)**. Ocorre quando asserções em um método de teste não têm explicação, não é documentado e afeta a legibilidade, a compreensibilidade e a manutenção. Se uma das afirmações falhar, é difícil identificar qual falhou. O método *assertThat*() (linhas 7, 10 e 11) é "invocado" 3 vezes no método de teste. Em cada declaração, é verificada uma condição diferente, mas não é fornecida uma mensagem de explicação para cada declaração. Portanto, se uma das declarações falhar, a identificação da causa da falha não é direta.

Código 3.1 - *Exemplo de Test Smell Assertion Roulette*

```
1   @MediumTest
2   public void testCloneNonBareRepoFromLocalTestServer() throws
    Exception {
3       Clone cloneOp = new Clone(false,
    integrationGitServerURIFor("small-repo.early.git"),
    helper().newFolder());
4
5       Repository repo = executeAndWaitFor(cloneOp);
6
7       assertThat(repo,
    hasGitObject("ba1f63e4430bff267d112b1e8afc1d6294db0ccc"));
8
9       File readmeFile = new File(repo.getWorkTree(), "README");
10      assertThat(readmeFile, exists());
11      assertThat(readmeFile, ofLength(12));
12  }
```

Fonte: Adaptado de Peruma *et al.* (2019).

b) ***Sensitive Equality* (SE)**. Ocorre quando o método *toString*() é usado em métodos de teste, "invocando-o" para verificar objetos, para comparar a sua saída com a sequência específica. Alterações na implementação do método *toString*() podem resultar em falha. Na linha 15 do exemplo, o valor retornado pelo método *toString*() é comparado a *string* "/54.204.10.41", se houver alterações na implementação dos objetos do método *toString*(), pode ocorrer falha.

Código 3.2 - *Exemplo de Test Smell Sensitive Equality*

```
1. @Test
2. public void test1() throws UnknownHostException {
3.
4.      String peersPacket = "F8 4E 11 F8 4B C5 36 81 " +
5.          "CC 0A 29 82 76 5F B8 40 D8 D6 0C 25 80 FA 79 5C " +
6.          "FC 03 13 EF DE BA 86 9D 21 94 E7 9E 7C B2 B5 22 " +
7.          "F7 82 FF A0 39 2C BB AB 8D 1B AC 30 12 08 B1 37 " +
8.          "E0 DE 49 98 33 4F 3B CF 73 FA 11 7E F2 13 F8 74 " +
9.          "17 08 9F EA F8 4C 21 B0";
10.
11.       byte[] payload = Hex.decode(peersPacket);
12.
13.       byte[] ip = decodeIP4Bytes(payload, 5);
14.
15.       assertEquals(InetAddress.getByAddress(ip).toString(),
    ("/54.204.10.41"));
16.   }
```

Fonte: Adaptado de Peruma *et al.* (2019).

4 FERRAMENTAS PARA VISUALIZAÇÃO DE *TEST SMELLS*

Neste experimento, as ferramentas `TsVizzEvolution`[1] e `VITRuM`[2] são submetidas a um estudo comparativo. Esse estudo visa analisar fatores correspondentes a eficácia, e usabilidade que podem auxiliar *testers*, gerente de testes e demais envolvidos nas atividades de teste de software a visualizar mais agilmente os *test smells*, para corrigi-los e melhorar a qualidade do código de teste.

4.1. `TSVizzEvolution` ou `VITRuM`

4.1.1 `TSVizzEvolution`

A ferramenta `TSVizzEvolution` utiliza três técnicas de visualização para exibir a ocorrência e evolução de 21 *test smells* (Palomba *et al.*, 2013; Peruma *et al.*, 2019; Deursen *et al.*, 2001). Esses *test smells* são detectados pela ferramenta `JNose Test`[3]: *Assertion Roulette, Conditional Test Logic, Constructor Initialization, Default Test, Dependent Test, Duplicate Assert, Eager Test, Empty Test, Exception Catching Throwing, General Fixture, Ignored Test, Lazy Test, Magic Number Test, Mystery Guest, Print Statement, Redundant Assertion, Resource Optimism, Sensitive Equality, Sleepy Test, Unknown Test, Verbose Test* e armazenados em um arquivo `.csv`. A ferramenta `TSVizzEvolution` estrutura os dados do arquivo `.csv` conforme visualização escolhida, a análise de *test smells* pode ser realizada para uma versão (Análise Única) ou para duas versões (Análise de Evolução).

4.1.1.1 Análise Única

Para a análise das ocorrências de *test smells* para uma versão do código de teste estão disponíveis duas técnicas: *Graph View* e *Treemap View*, essas técnicas são explicadas a seguir:

a) ***Graph View***. Os dados retornados pela `JNose` são estruturados em grafos. Os componentes do grafo variam de acordo com a granularidade escolhida, tem-se seis granularidades disponíveis (Quadro 4.1). Na Figura 4.1 verifica-se um grafo e na Figura 4.2 o detalhamento da legenda, que exibe o significado dos ícones utilizados como nós no grafo. Os componentes podem ser arrastados para ajustar a visualização conforme necessidade.

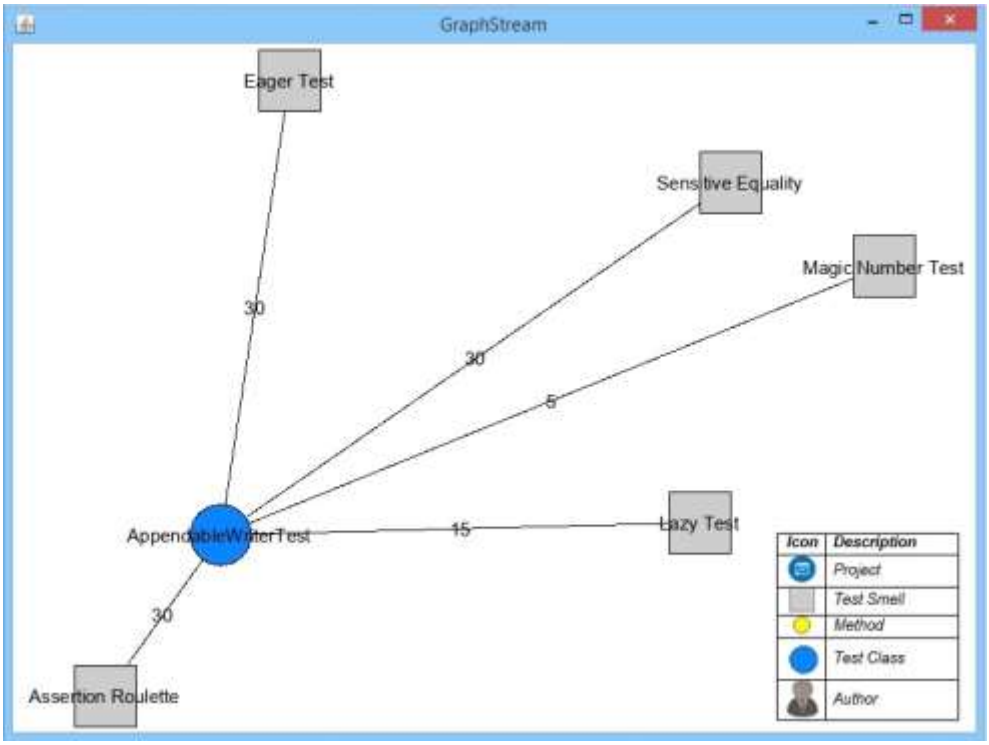
[1] https://github.com/arieslab/TSVizzEvolution
[2] https://plugins.jetbrains.com/plugin/14160-vitrum
[3] http://jnose.herokuapp.com/

Quadro 4.1 - Granularidade da Ferramenta

Granularidade	**Descrição**
Project	Exibe a relação entre todo o projeto e os *test smells* (1:N).
All Test Classes	Exibe a relação entre as classes de teste e os *test smells* (N:N).
A Specific Test Class	Exibe a relação entre as classes de teste e os *test smells* (1:N). Para executar a ferramenta com essa granularidade, os usuários precisam selecionar uma classe de teste.
A Specific Test Smells	Exibe a relação entre as classes de teste e os *test smells* (N:1). Para executar a ferramenta com essa granularidade, os usuários devem selecionar um *test smell*.
Author	Exibe a relação entre os autores, os *test smells* (N:N) e as classes de teste (1: N). Para utilizar a ferramenta com essa granularidade, os usuários devem selecionar o autor (um específico ou todos autores) e um *test smell*.
Methods	Exibe a relação entre o *test smell*, a classe de teste e os métodos (1:N). Os usuários precisam selecionar um *test smell* e uma classe de teste.

Fonte: Do autor (2021).

Figura 4.1 - *Graph View*



Fonte: Do autor (2021).

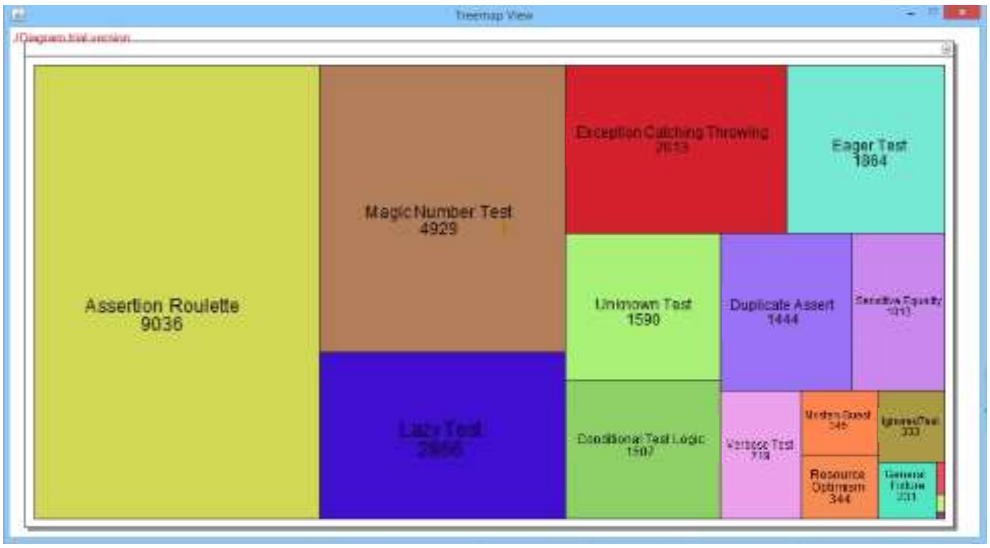
Figura 4.2 - Legenda *Graph View*

Icon	Description
	Project
	Test Smell
	Method
	Test Class
	Author

Fonte: Do autor (2021).

b) ***Treemap View***. Os *test smells* são representados por retângulos, onde cada retângulo possui uma cor, gerada aleatoriamente. Além disso, os tamanhos dos retângulos representam as suas quantidades de ocorrências, ou seja, quanto maior o retângulo mais quantidade de ocorrências de *test smells* (Figura 4.3).

Figura 4.3 - *Treemap View*



Fonte: Do autor (2021).

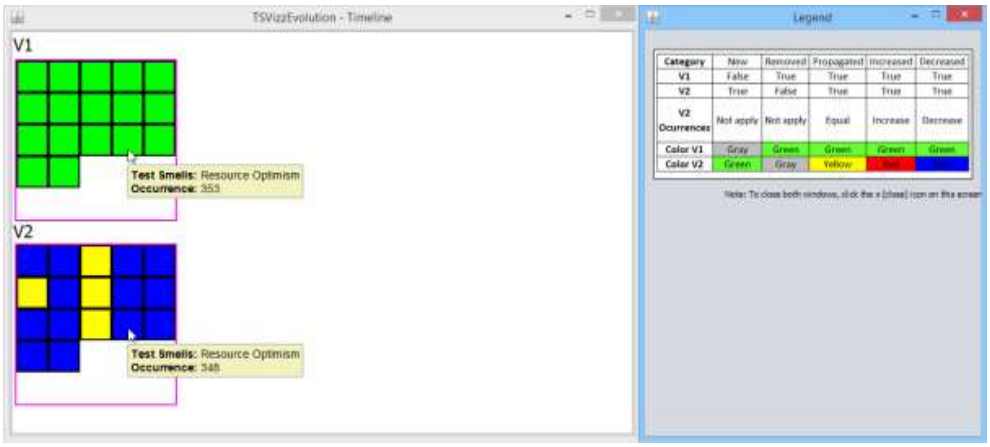
4.1.1.2 Análise de Evolução

Para a análise das ocorrências de *test smells* para duas versões é utilizada a técnica *Timeline View*, essa técnica é explicada a seguir.

a) ***Timeline View***. Os usuários podem escolher entre três granularidades: i) *Project*; ii) *All Test Classes;* ou *iii) Methods*. Os artefatos (projeto, classe de teste e *test smells*) do código de teste são representados por retângulos e as versões são apresentadas em paralelo horizontalmente, conforme ordem de inserção dos arquivos `.csv`. Os retângulos são aninhados da seguinte forma: i) **retângulo externo** representa o projeto ou a classe de teste

(conforme granularidade selecionada) possui borda na **cor rosa**; e ii) **retângulos menores** representam cada *test smell* detectado em uma classe de teste ou projeto, com as cores conforme comportamento durante a evolução, podendo ser verde, azul, cinza ou vermelho.

Figura 4.4 - *Timeline View*



Fonte: Do autor (2021).

As posições dos retângulos nas duas versões são mantidas para facilitar a rastreabilidade entre classes de teste e *test smells*. As cores dos *test smells* são categorizados conforme sua existência e quantidade de ocorrências (Quadro 4.2) e apresentadas ao lado da *timeline* (Figura 4.4). Os usuários podem ver os nomes das classes de teste, projeto e *test smells* e suas quantidade de ocorrências ao posicionar o *mouse* sobre cada retângulo. As categorias são (Versão um - V_1 e Versão dois - V2):

- **Novo**. Se V_1 não tiver o *test smell* específico (*False*) e V_2 (*True*), a categoria é "**New**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**cinza**" e "**verde**", respectivamente;
- **Removido**. Se V_1 tiver o *test smell* específico (*True*) e V_2 (*False*), a categoria é "**Removido**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**verde**" e "**cinza**", respectivamente;
- **Propagado**. Se V_1 e V_2 tiverem o *test smell* específico (*True* - *True*) e a quantidade for a mesma (*Igual*), a categoria é "**Propagado**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**verde**" e "**amarelo**", respectivamente;
- **Aumentado**. Se V_1 e V_2 tiverem um *test smell* específico (*True* - *True*), mas a quantidade de *test smells* aumenta em V_2 (**Aumenta**), a categoria é "**Aumentado**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem a cores "**verde**" e "**vermelho**", respectivamente;

- **Diminuído**. Se V_1 e V_2 tiverem um *test smell* específico (*True* - *True*), mas a quantidade de *test smells* diminui na V_2 (**Diminui**), a categoria é "**Diminuído**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem a cores "**verde**" e "**azul**".

Quadro 4.2 - Classificação dos *Test Smells*

Categoria	**V_1**	**V_2**	**V_2 Ocorrências**	**Cor V_1**	**Cor V_2**
Novo	*False*	*True*	Não se aplica	Cinza	Verde
Removido	*True*	*False*	Não se aplica	Verde	Cinza
Propagado	*True*	*True*	Igual	Verde	Amarelo
Aumentado	*True*	*True*	Aumentado	Verde	Vermelho
Diminuído	*True*	*True*	Diminuído	Verde	Azul

Fonte: Do autor (2021).

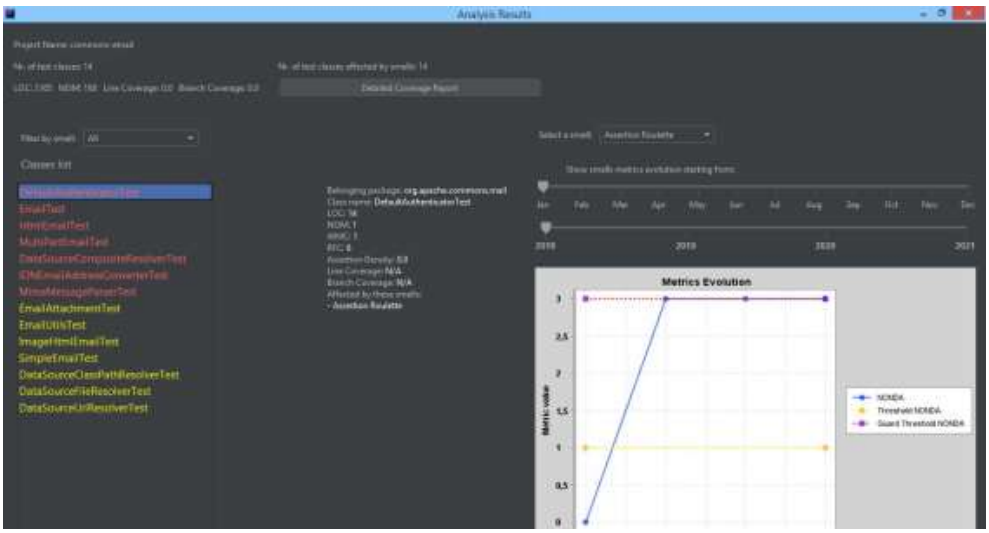
4.1.2 `VITRuM`

A ferramenta `VITRuM` (*VIsualization of Test-Related Metrics*) é um *plug-in* para o IntelliJ[1] para código Java que inclui o cálculo de medidas de código de teste e detecta 7 *test smells.* As medidas se dividem em dois tipos (i) estruturais referem ao tamanho, à coesão, ao acoplamento e à complexidade e (ii) dinâmicas exibem informações sobre a eficácia dos testes e cobertura de testes. Os *test smells* são exibidos através de uma lista com as classes de teste em cores diferentes (i) vermelho para classes de teste afetadas por *test smells*, com valores de medidas de código de teste acima dos limites definidos; (ii) amarelo para classes de teste afetadas por *test smells*, com valores de medidas de código de teste dentro dos limites definidos; e (iii) cinza para classes de teste sem *test smells*. Além disso, os usuários podem filtrar os resultados por *test smell* para exibir sua evolução em um gráfico durante o desenvolvimento do projeto (Figura 4.5). A seguir são detalhados as medidas e os *test smells* que a ferramenta abrange:

a) **Medidas estruturais:** LOC (*Line of Code*), AD (*Assertion Density*) e WMC (*Weigthed Methods Per Class*);
b) **Medidas dinâmicas:** *Line Coverage, Branch Coverage, Mutation Coverage e Flaky Tests.*
c) ***Test Smells:*** *Assertion Roulette, Eager Test, General Fixture, Indirect Testing, Sensitive Equality, Mystery Guest e Resource Optimism.*

[1] https://www.jetbrains.com/pt-br/idea/

Figura 4.5 - Painel Principal `VITRuM`



Fonte: Do autor (2021).

4.2. Treinamento – `TSVizzEvolution` ou `VITRuM` – `Commons IO`

4.2.1 Treinamento – `TSVizzEvolution` – `Commons IO`

`TSVizzEvolution`	- Selecionar 1 versão - Carregar o arquivo "*resultado_evolution1.csv*" da pasta *commons-io-2.6* - Escolher a técnica *Graph View* - Selecionar a granularidade *A Specific Test Class* - Selecionar a classe de teste *BrokenOutputStreamTest* e clicar em *Generate Graph View* **P1: Quais são os *test smells* que essa classe possui?** - Fechar a tela do grafo gerado e selecionar outra classe de teste que desejar e clicar em *Generate Graph View* **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Selecionar a granularidade *A Specific Test Smells* - Selecionar o *test smell Sensitive Equality* e clicar em *Generate Graph View* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** - Fechar a janela do grafo e selecionar a Granularidade *Author* - Selecionar o *Author: All*, que corresponde a todos autores - Selecionar o *test smell Sensitive Equality* **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação dos possíveis autores pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** - Fechar a janela do grafo - Selecionar o *Author Benson Margulies* **P7: Quais são os *test smells* que esse autor é responsável?** - Selecionar a *View Type: Treemap View* e clicar em *Generate Treemap View* **P8: Qual *test smell com* mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** - Fechar a janela da *Treemap*

	- Fechar a janela *TSVizzEvolution - One Version* - Selecionar 2 versões - Carregar o arquivo "*resultado_evolution1.csv*" da pasta *commons-io-2.1* em "*Select the first .csv*" - Carregar o arquivo "*resultado_evolution1.csv"* da pasta *commons-io-2.6* em "*Select the second .csv*" - Selecionar *Methods* em "*Select the level of granularity"* - Carregar o arquivo fornecido "*commons-io_result_byclasstest_testsmells.csv*" da pasta *commons-io-2.1* em "*Select the first .csv File*" - Carregar o arquivo "*commons-io_result_byclasstest_testsmells.csv"* da pasta *commons-io-2.6 "Select the second .csv File" e clicar em Generate Timeline View* - Passar o mouse sobre a primeira classe: *AppendableOutputStreamTest* **P9: Qual o comportamento do *test smell Assertion Roulette* da classe *AppendableOutputStreamTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais método? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-io/blob/master/src/test/java/org/apache/commons/io/output/AppendableOutputStreamTest.java* e procurar o método *testWriteInt* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.** - Clicar em fechar ao na janela de título *Legend*

4.2.2 Treinamento – VITRuM – Commons IO

VITRuM	- Acessar *File - Open* e carregar a pasta do projeto *Commons IO* - Acessar *Tools* - VITRuM - *Calculate Test Factors* - Deixar selecionado somente *Test Smells (default)* - Clicar em *Star Analysis* - Na janela aberta, procurar a classe *BrokenOutputStreamTest* **P1: Quais são os *test smells* que essa classe possui?** - Selecionar outra classe de teste que desejar **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Selecionar o *test smell Sensitive Equality* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** **P7: Quais são os *test smells* que esse autor é responsável?** **P8: Qual *test smell com* mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** **P9: Qual o comportamento do *test smell Assertion Roulette* da classe *AppendableOutputStreamTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais métodos? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-io/blob/master/src/test/java/org/apache/commons/io/output/AppendableOutputStreamTest.java* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.**

4.3. Tarefa - TSVizzEvolution ou VITRuM - Commons Email ou Commons Text

4.3.1 Tarefa - TSVizzEvolution - Commons Email

TSVizzEvolution	- Selecionar 1 versão - Carregar o arquivo "*resultado_evolution1.csv*" da pasta *commons-email-1.9* - Escolher a técnica *Graph View* - Selecionar a granularidade *A Specific Test Class* - Selecionar a classe de teste *IDNEmailAddressConvertTest* e clicar em *Generate Graph View* **P1: Quais são os *test smells* que essa classe possui?** - Fechar a tela do grafo gerado e selecionar outra classe de teste que desejar e clicar em *Generate Graph View* **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Fechar a janela do grafo e selecionar a Granularidade *A Specific Test Smells* - Selecionar o *test smell Sensitive Equality* e clicar em *Generate Graph View* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** - Fechar a janela do grafo e selecione a Granularidade *Author* - Selecionar o *Author: All*, que corresponde a todos autores - Selecionar o *test smells Sensitive Equality* **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** - Fechar a janela do grafo - Selecionar o *Author Stefan Bodewig* **P7: Quais são os *test smells* que esse autor é responsável?** - Selecionar a *View Type*: *Treemap View* e clicar em *Generate Treemap View* **P8: Qual *test smell* com mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** - Fechar a janela da *Treemap* - Fechar a janela *TSVizzEvolution - One Version* - Selecionar 2 versões - Carregar o arquivo "resultado_evolution1.csv" da pasta *commons-email-1.0* em "*Select the first .csv*" - Carregar o arquivo "*resultado_evolution1.csv"* da pasta *commons-email-1.9* em "*Select the second .csv*" - Selecionar *Methods* em "*Select the level of granularity"* - Carregar o arquivo fornecido "*commons-email_result_byclasstest_testsmells.csv*" da pasta *commons-email-1.0* em "*Select the first .csv File*" - Carregar o arquivo "*commons-email_result_byclasstest_testsmells.csv"* da pasta *commons-email-1.9 "Select the second .csv File"* e clicar em *Generate Timeline View* - Passar o mouse sobre a quinta classe: *DefaultAuthenticatorTest* **P9: Qual o comportamento do *test smell Assertion Roulette* da classe *DefaultAuthenticatorTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais método? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-email/blob/master/src/test/java/org/apache/commons/mail/DefaultAuthenticatorTest.java* e procurar o método *testDefaultAuthenticatorConstructor*

	P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.

4.3.2 Tarefa - `TSVizzEvolution - Commons Text`

`TSVizzEvolution`	- Selecionar 1 versão - Carregar o arquivo "*resultado_evolution1.csv*" da pasta *commons-text-1.9* - Escolher a técnica *Graph View* - Selecionar a granularidade *A Specific Test Class* - Selecionar a classe de teste *BiFunctionStringLookupTest* e clicar em *Generate Graph View* **P1: Quais são os *test smells* que essa classe possui?** - Fechar a tela do grafo gerado e selecionar outra classe de teste que desejar e clicar em *Generate Graph View* **P2: Qual classe foi escolhida? Qual *test smell* com maior ocorrência e a quantidade?** - Fechar a janela do grafo e selecionar a granularidade *A Specific Test Smells* - Selecionar o *test smell Sensitive Equality* e clicar em *Generate Graph View* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** - Fechar a janela do grafo e selecionar a Granularidade *Author* - Selecionar o *Author: All*, que corresponde a todos autores - Selecionar o *test smell Sensitive Equality* **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os responsáveis pelo *test smell Sensitive Equality*?** - Fechar a janela do grafo - Selecionar o *Author Gary Gregory* **P7: Quais são os test smells que esse autor é responsável?** - Selecionar a *View Type*: *Treemap View* e clicar em *Generate Treemap View* **P8: Qual *test smell com* maior ocorrência? Essa informação pode ser útil para o projeto? Explique.** - Fechar a janela da *Treemap* - Fechar a janela *TSVizzEvolution - One Version* - Selecionar 2 versões - Carregar o arquivo *"resultado_evolution1.csv"* da pasta *commons-text-1.0* em "*Select the first .csv*" - Carregar o arquivo "*resultado_evolution1.csv"* da pasta *commons-text-1.9* em "*Select the second .csv*" - Selecionar *Methods* em "*Select the level of granularity"* - Carregar o arquivo fornecido "*commons-text_result_byclasstest_testsmells.csv*" da pasta *commons-text-1.0* em "*Select the first .csv File*" - Carregar o arquivo "*commons-text_result_byclasstest_testsmells.csv"* da pasta *commons-text-1.9 "Select the second .csv File"* e clicar *em Generate Timeline View* - Passar o mouse sobre a primeira classe: *AggregateTranslatorTest* **P9: Qual o comportamento *do test smell Sensitive Equality* da classe *AggregateTranslatorTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais métodos? E em quais linhas?** - Abrir a classe no *Git Hub*: *https://github.com/apache/commons-text/blob/master/src/test/java/org/apache/commons/text/translate/AggregateTranslatorTest.java* e procurar o método *testNonNull* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.**

	- Clicar em fechar ao lado do título *Legend*

4.3.3 Tarefa - `VITRuM - Commons Email`

VITRuM	- Acessar *File* - *Open* e carregar a pasta do projeto *Commons Email* - Acessar *Tools* - `VITRuM` - *Calculate Test Factors* - Deixar selecionado somente *Test Smells (default)* - Clicar em *Star Analysis* - Na janela aberta, procurar a classe *IDNEmailAddressConverterTest* **P1: Quais são os *test smells* que essa classe possui?** - Selecionar outra classe de teste que desejar **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Selecionar o *test smell Sensitive Equality* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** **P7: Quais são os *test smells* que esse autor é responsável?** **P8: Qual *test smell* com mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** **P9: Qual o comportamento do *test smell Assertion Roulette* da classe *EmailTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais métodos? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-email/blob/master/src/test/java/org/apache/commons/mail/DefaultAuthenticatorTest.java* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.**

4.3.4 Tarefa - `VITRuM - Commons Text`

VITRuM	- Acessar *File* - *Open* e carregar a pasta do projeto *Commons Text* - Acessar *Tools* - `VITRuM` - *Calculate Test Factors* - Deixar selecionado somente *Test Smells (default)* - Clicar em *Star Analysis* - Na janela aberta, procurar a classe *BiFunctionStringLookupTest* **P1: Quais são os *test smells* que essa classe possui?** - Selecionar outra classe de teste que desejar **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Selecionar o *test smell Sensitive Equality* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** **P7: Quais são os *test smells* que esse autor é responsável?** **P8: Qual *test smell com* mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** **P9: Qual o comportamento *do test smell Assertion Roulette* da classe *AggregateTranslatorTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais métodos? E em quais linhas?**

	- Abrir a classe no *GitHub* : *https://github.com/apache/commons-text/blob/master/src/test/java/org/apache/commons/text/translate/AggregateTranslatorTest.java* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.**

4.4 Questionário

Disponível em*: https://forms.gle/1mis7iVRHrbBSvws5*

REFERÊNCIAS

CALEGARE, A. J. **Introdução ao delineamento de experimentos**. Editora Blucher, 2009.

PALOMBA, F., BAVOTA, G., DI PENTA, M., OLIVETO, R., DE LUCIA, A., POSHYVANYK, D. **Detecting bad smells in source code using change history information**. In: International Conference on Automated Software Engineering. p. 268-278. 2013.

PECORELLI, F.; LILLO, G.; PALOMBA, F.; LUCIA, A. **VITRuM-A Plug-In for the Visualization of Test-Related Metrics**. In: International Conference on Advanced Visual Interfaces. p.1-3, 2020.

PERUMA, A.; ALMALKI, K.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A.; PALOMBA, F. **On the distribution of test smells in open source Android applications: an exploratory study**. In: International Conference on Computer Science and Software Engineering. p. 193-202, 2019.

VAN DEURSEN, A.; MOONEN, L.; VAN DEN BERGH, A.; KOK, G. **Refactoring test code**. In: Conference on extreme programming and flexible processes in software engineering. p. 92-95, 2001

VIRGÍNIO, T.; SANTANA, R.; MARTINS, L. A.; SOARES, L. R.; COSTA, H.; MACHADO, I. **On the influence of Test Smells on Test Coverage**. In: Brazilian Symposium on Software Engineering. p. 467-471, 2019.

APÊNDICE B

ADRIANA PRISCILA SANTOS CRUZ

Atividades do Estudo Experimental com Ferramentas para Visualização de *Test Smells* - Grupo 1

LAVRAS-MG
2021

Sumário

1 EXPERIMENTO CONTROLADO

As informações são lidas, com a finalidade de manter o padrão, pois trata-se de um estudo experimental e o treinamento com todos os participantes deve ser o mais semelhante. São explicados os conceitos de *test smells* e as ferramentas para a visualização de *test smells* utilizadas nesse estudo. Posteriormente, são realizadas as tarefas propostas nesse experimento.

2 *DESIGN* DO ESTUDO

O objetivo desse estudo é visualizar a ocorrência e a evolução de *test smells*. Sendo conduzido na seguinte ordem:

a) ***Test Smells*:** São explicados os conceitos e exemplificados dois *test smells*;
b) **Ferramentas para a Visualização de *Test Smells*:** São explicadas sobre duas ferramentas para a visualização de *test smells;*
c) **`TSVizzEvolution`** é a primeira ferramenta detalhada no estudo;
d) **Treinamento 1.** `TSVizzEvolution - Commons IO;`
e) **Tarefa 1.** `TSVizzEvolution – Commons Email;`
f) **`VITRuM`** é a segunda ferramenta detalhada no estudo;
g) **Treinamento 2.** `VITRuM - Commons IO;`
h) **Tarefa 2.** `VITRuM - Commons Text;`
i) **Aplicar questionário.** Questionário para caracterização dos participantes e *feedback* do experimento.

3 *TEST SMELLS*

Os *test smells* são considerados decisões ou práticas ruins de design no código de teste, os quais diminuem a qualidade dos testes e tornam os testes mais difíceis de serem compreendidos e mantidos durante a evolução dos sistemas de software. Dois *test smells* são explicados a seguir:

a) ***Assertion Roulette* (AR)**. Ocorre quando asserções em um método de teste não têm explicação, não é documentado e afeta a legibilidade, a compreensibilidade e a manutenção. Se uma das afirmações falhar, é difícil identificar qual falhou. O método *assertThat*() (linhas 7, 10 e 11) é "invocado" 3 vezes no método de teste. Em cada declaração, é verificada uma condição diferente, mas não é fornecida uma mensagem de explicação para cada declaração. Portanto, se uma das declarações falhar, a identificação da causa da falha não é direta.

Código 3.1 - *Exemplo de Test Smell Assertion Roulette*

```
13  @MediumTest
14  public void testCloneNonBareRepoFromLocalTestServer() throws
    Exception {
15      Clone cloneOp = new Clone(false,
    integrationGitServerURIFor("small-repo.early.git"),
    helper().newFolder());
16
17      Repository repo = executeAndWaitFor(cloneOp);
18
19      assertThat(repo,
    hasGitObject("ba1f63e4430bff267d112b1e8afc1d6294db0ccc"));
20
21      File readmeFile = new File(repo.getWorkTree(), "README");
22      assertThat(readmeFile, exists());
23      assertThat(readmeFile, ofLength(12));
24  }
```

Fonte: Adaptado de Peruma *et al.* (2019).

b) ***Sensitive Equality* (SE)**. Ocorre quando o método *toString*() é usado em métodos de teste, "invocando-o" para verificar objetos, para comparar a sua saída com a sequência específica. Alterações na implementação do método *toString*() podem resultar em falha. Na linha 15 do exemplo, o valor retornado pelo método *toString*() é comparado a *string* "/54.204.10.41", se houver alterações na implementação dos objetos do método *toString*(), pode ocorrer falha.

Código 3.2 - *Exemplo de Test Smell Sensitive Equality*

```
17.   @Test
18.   public void test1() throws UnknownHostException {
19.
20.       String peersPacket = "F8 4E 11 F8 4B C5 36 81 " +
21.           "CC 0A 29 82 76 5F B8 40 D8 D6 0C 25 80 FA 79 5C " +
22.           "FC 03 13 EF DE BA 86 9D 21 94 E7 9E 7C B2 B5 22 " +
23.           "F7 82 FF A0 39 2C BB AB 8D 1B AC 30 12 08 B1 37 " +
24.           "E0 DE 49 98 33 4F 3B CF 73 FA 11 7E F2 13 F8 74 " +
25.           "17 08 9F EA F8 4C 21 B0";
26.
27.       byte[] payload = Hex.decode(peersPacket);
28.
29.       byte[] ip = decodeIP4Bytes(payload, 5);
30.
31.       assertEquals(InetAddress.getByAddress(ip).toString(),
    ("/54.204.10.41"));
32.   }
```

Fonte: Adaptado de Peruma *et al.* (2019).

4 FERRAMENTAS PARA VISUALIZAÇÃO DE *TEST SMELLS*

Neste experimento, as ferramentas `TsVizzEvolution`[1] e `VITRuM`[2] são submetidas a um estudo comparativo. Esse estudo visa analisar fatores correspondentes a eficácia, e usabilidade que podem auxiliar *testers*, gerente de testes e demais envolvidos nas atividades de teste de software a visualizar mais agilmente os *test smells*, para corrigi-los e melhorar a qualidade do código de teste.

4.1 `TSVizzEvolution`

A ferramenta `TSVizzEvolution` utiliza três técnicas de visualização para exibir a ocorrência e evolução de 21 *test smells* (Palomba *et al.*, 2013; Peruma *et al.*, 2019; Deursen *et al.*, 2001). Esses *test smells* são detectados pela ferramenta `JNose Test`[3]: *Assertion Roulette, Conditional Test Logic, Constructor Initialization, Default Test, Dependent Test, Duplicate Assert, Eager Test, Empty Test, Exception Catching Throwing, General Fixture, Ignored Test, Lazy Test, Magic Number Test, Mystery Guest, Print Statement, Redundant Assertion, Resource Optimism, Sensitive Equality, Sleepy Test, Unknown Test, Verbose Test* e armazenados em um arquivo `.csv`. A ferramenta `TSVizzEvolution` estrutura os dados do arquivo `.csv` conforme visualização escolhida, a análise de *test smells* pode ser realizada para uma versão do código de teste (Análise Única) ou para duas versões (Análise de Evolução).

4.1.1 Análise Única

Para a análise das ocorrências de *test smells* para uma versão do código de teste estão disponíveis duas técnicas: *Graph View* e *Treemap View*, essas técnicas são explicadas a seguir:

a) ***Graph View***. Os dados retornados pela `JNose` são estruturados em grafos. Os componentes do grafo variam de acordo com a granularidade escolhida, tem-se seis granularidades disponíveis (Quadro 4.1). Na Figura 4.1 verifica-se um grafo e na Figura 4.2 o detalhamento da legenda, que exibe o significado dos ícones utilizados como nós no grafo. Os componentes podem ser arrastados para ajustar a visualização conforme necessidade.

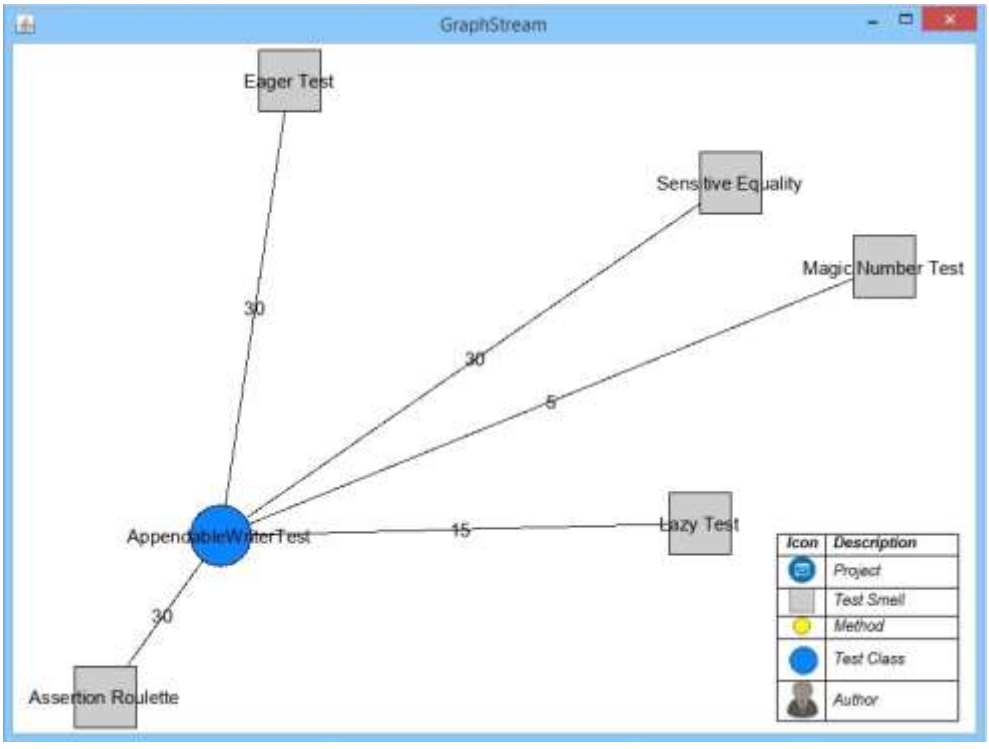
[1] https://github.com/arieslab/TSVizzEvolution
[2] https://plugins.jetbrains.com/plugin/14160-vitrum
[3] http://jnose.herokuapp.com/

Quadro 4.1 - Granularidade da Ferramenta

Granularidade	**Descrição**
Project	Exibe a relação entre todo o projeto e os *test smells* (1:N).
All Test Classes	Exibe a relação entre as classes de teste e os *test smells* (N:N).
A Specific Test Class	Exibe a relação entre as classes de teste e *test smells* (1:N). Para executar a ferramenta com essa granularidade, os usuários precisam selecionar uma classe de teste.
A Specific Test Smells	Exibe a relação entre as classes de teste e os *test smells* (N:1). Para executar a ferramenta com essa granularidade, os usuários devem selecionar um *test smell*.
Author	Exibe a relação entre os autores, os *test smells* (N:N) e as classes de teste (1: N). Para utilizar a ferramenta com essa granularidade, os usuários devem selecionar o autor (um específico ou todos autores) e um *test smell*.
Methods	Exibe a relação entre o *test smell*, a classe de teste e os métodos (1:N). Os usuários precisam selecionar um *test smell* e uma classe de teste.

Fonte: Do autor (2021).

Figura 4.1 - *Graph View*



Fonte: Do autor (2021).

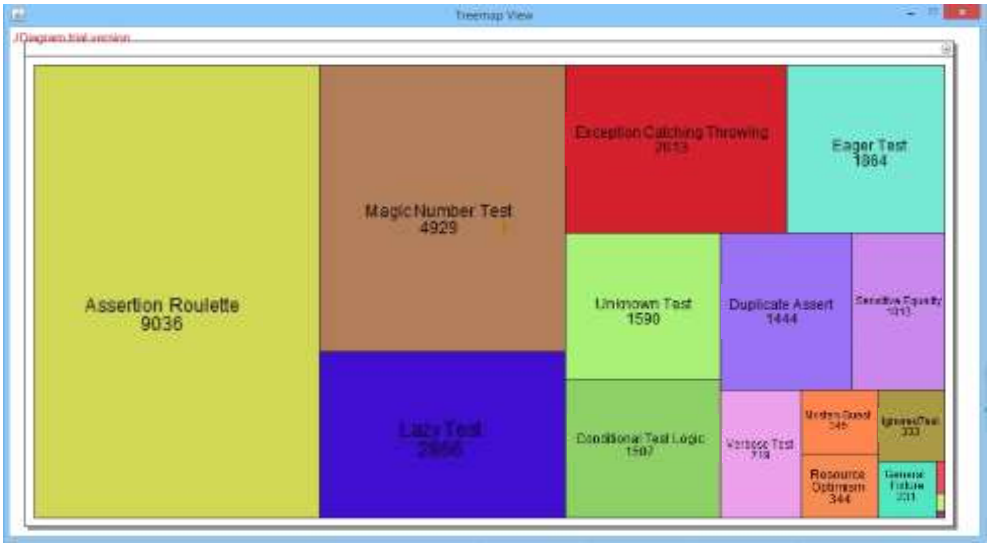
Figura 4.2 - Legenda *Graph View*

Icon	Description
	Project
	Test Smell
	Method
	Test Class
	Author

Fonte: Do autor (2021).

b) ***Treemap View***. Os *test smells* são representados por retângulos, onde cada retângulo possui uma cor, gerada aleatoriamente. Além disso, os tamanhos dos retângulos representam as suas quantidades de ocorrências, ou seja, quanto maior o retângulo mais quantidade de ocorrências de *test smells* (Figura 4.3).

Figura 4.3 - *Treemap View*



Fonte: Do autor (2021).

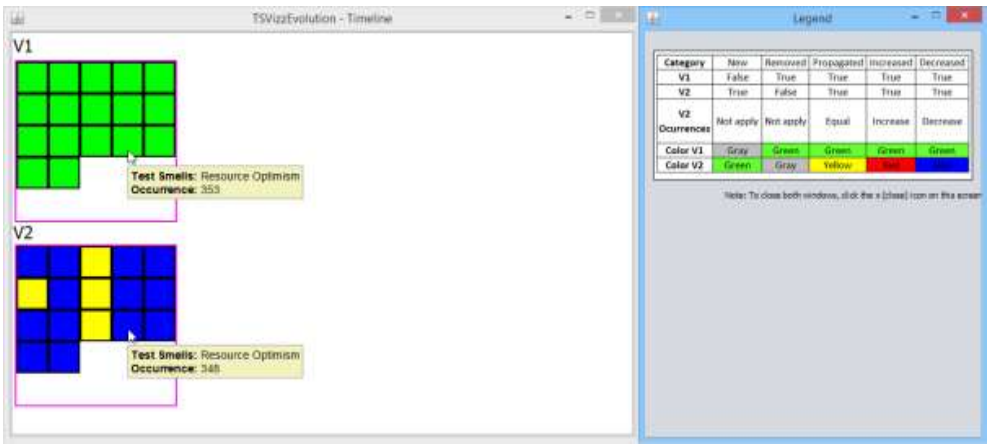
4.1.2 Análise de Evolução

Para a análise das ocorrências de *test smells* para duas versões é utilizada a técnica *Timeline View*, essa técnica é explicada a seguir.

a) ***Timeline View***. Os usuários podem escolher entre três granularidades: i) *Project*; ii) *All Test Classes;* ou *iii) Methods*. Os artefatos (projeto, classe de teste e *test smells*) do código de teste são representados por retângulos e as versões são apresentadas em paralelo horizontalmente, conforme ordem de inserção dos arquivos `.csv`. Os retângulos são aninhados da seguinte forma: i) **retângulo externo** representa o projeto ou a classe de teste

(conforme granularidade selecionada) possui borda na **cor rosa**; e ii) **retângulos menores** representam cada *test smell* detectado em uma classe de teste ou projeto, com as cores conforme comportamento durante a evolução, podendo ser verde, azul, cinza ou vermelho.

Figura 4.4 - *Timeline View*



Fonte: Do autor (2021).

As posições dos retângulos nas duas versões são mantidas para facilitar a rastreabilidade entre classes de teste e *test smells*. As cores dos *test smells* são categorizados conforme sua existência e quantidade de ocorrências (Quadro 4.2) e apresentadas ao lado da *timeline* (Figura 4.4). Os usuários podem ver os nomes das classes de teste, projeto e *test smells* e suas quantidade de ocorrências ao posicionar o *mouse* sobre cada retângulo. As categorias são (Versão um - V_1 e Versão dois - V2):

- **Novo**. Se V_1 não tiver o *test smell* específico (*False*) e V_2 (*True*), a categoria é "**New**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**cinza**" e "**verde**", respectivamente;
- **Removido**. Se V_1 tiver o *test smell* específico (*True*) e V_2 (*False*), a categoria é "**Removido**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**verde**" e "**cinza**", respectivamente;
- **Propagado**. Se V_1 e V_2 tiverem o *test smell* específico (*True - True*) e a quantidade for a mesma (*Igual*), a categoria é "**Propagado**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**verde**" e "**amarelo**", respectivamente;
- **Aumentado**. Se V_1 e V_2 tiverem um *test smell* específico (*True - True*), mas a quantidade de *test smells* aumenta em V_2 (**Aumenta**), a categoria é "**Aumentado**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem a cores "**verde**" e "**vermelho**", respectivamente;

– **Diminuído**. Se V_1 e V_2 tiverem um *test smell* específico (*True* - *True*), mas a quantidade de *test smells* diminui na V_2 (**Diminui**), a categoria é "**Diminuído**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem a cores "**verde**" e "**azul**".

Quadro 4.2 - Classificação dos *Test Smells*

Categoria	**V_1**	**V_2**	**V_2 Ocorrências**	**Cor V_1**	**Cor V_2**
Novo	*False*	*True*	Não se aplica	Cinza	Verde
Removido	*True*	*False*	Não se aplica	Verde	Cinza
Propagado	*True*	*True*	Igual	Verde	Amarelo
Aumentado	*True*	*True*	Aumentado	Verde	Vermelho
Diminuído	*True*	*True*	Diminuído	Verde	Azul

Fonte: Do autor (2021).

4.2 Treinamento 1 - `TSVizzEvolution - Commons IO`

`TSVizzEvolution`	- Selecionar 1 versão - Carregar o arquivo "*resultado_evolution1.csv*" da pasta *commons-io-2.6* - Escolher a técnica *Graph View* - Selecionar a granularidade *A Specific Test Class* - Selecionar a classe de teste *BrokenOutputStreamTest* e clicar em *Generate Graph View* **P1: Quais são os *test smells* que essa classe possui?** - Fechar a tela do grafo gerado e selecionar outra classe de teste que desejar e clicar em *Generate Graph View* **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Selecionar a granularidade *A Specific Test Smells* - Selecionar o *test smell Sensitive Equality* e clicar em *Generate Graph View* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** - Fechar a janela do grafo e selecionar a Granularidade *Author* - Selecionar o *Author: All*, que corresponde a todos autores - Selecionar o *test smell Sensitive Equality* **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação dos possíveis autores pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** - Fechar a janela do grafo - Selecionar o *Author Benson Margulies* **P7: Quais são os *test smells* que esse autor é responsável?** - Selecionar a *View Type: Treemap View* e clicar em *Generate Treemap View* **P8: Qual *test smell com* mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** - Fechar a janela da *Treemap* - Fechar a janela *TSVizzEvolution - One Version* - Selecionar 2 versões - Carregar o arquivo "*resultado_evolution1.csv*" da pasta *commons-io-2.1* em "*Select the first .csv*"

	- Carregar o arquivo "*resultado_evolution1.csv"* da pasta *commons-io-2.6* em "*Select the second .csv*" - Selecionar *Methods* em "*Select the level of granularity"* - Carregar o arquivo fornecido "*commons-io_result_byclasstest_testsmells.csv*" da pasta *commons-io-2.1* em "*Select the first .csv File*" - Carregar o arquivo "*commons-io_result_byclasstest_testsmells.csv"* da pasta *commons-io-2.6 "Select the second .csv File" e clicar em Generate Timeline View* - Passar o mouse sobre a primeira classe: *AppendableOutputStreamTest* **P9: Qual o comportamento do *test smell Assertion Roulette* da classe *AppendableOutputStreamTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais método? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-io/blob/master/src/test/java/org/apache/commons/io/output/AppendableOutputStreamTest.java* e procurar o método *testWriteInt* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.** - Clicar em fechar ao na janela de título *Legend*

4.3 Tarefa 1 - `TSVizzEvolution - Commons Email`

O formulário para responder as questões está disponível em : *https://forms.gle/ACZphwMn7SbrCmdP6*

`TSVizzEvolution`	- Selecionar 1 versão - Carregar o arquivo "*resultado_evolution1.csv*" da pasta *commons-email-1.9* - Escolher a técnica *Graph View* - Selecionar a granularidade *A Specific Test Class* - Selecionar a classe de teste *IDNEmailAddressConvertTest* e clicar em *Generate Graph View* **P1: Quais são os *test smells* que essa classe possui?** - Fechar a tela do grafo gerado e selecionar outra classe de teste que desejar e clicar em *Generate Graph View* **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Fechar a janela do grafo e selecionar a Granularidade *A Specific Test Smells* - Selecionar o *test smell Sensitive Equality* e clicar em *Generate Graph View* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** - Fechar a janela do grafo e selecione a Granularidade *Author* - Selecionar o *Author: All*, que corresponde a todos autores - Selecionar o *test smells Sensitive Equality* **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** - Fechar a janela do grafo - Selecionar o *Author Stefan Bodewig* **P7: Quais são os *test smells* que esse autor é responsável?** - Selecionar a *View Type*: *Treemap View* e clicar em *Generate Treemap View* **P8: Qual *test smell* com mais ocorrências? Essa informação pode ser útil para o projeto? Explique.**

	- Fechar a janela da *Treemap* - Fechar a janela *TSVizzEvolution - One Version* - Selecionar 2 versões - Carregar o arquivo "resultado_evolution1.csv" da pasta *commons-email-1.0* em "*Select the first .csv*" - Carregar o arquivo "*resultado_evolution1.csv"* da pasta *commons-email-1.9* em "*Select the second .csv*" - Selecionar *Methods* em "*Select the level of granularity"* - Carregar o arquivo fornecido "*commons-email_result_byclasstest_testsmells.csv*" da pasta *commons-email-1.0* em "*Select the first .csv File*" - Carregar o arquivo "*commons-email_result_byclasstest_testsmells.csv"* da pasta *commons-email-1.9 "Select the second .csv File"* e clicar em *Generate Timeline View* - Passar o mouse sobre a quinta classe: *DefaultAuthenticatorTest* **P9: Qual o comportamento do *test smell Assertion Roulette* da classe *DefaultAuthenticatorTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais método? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-email/blob/master/src/test/java/org/apache/commons/mail/DefaultAuthenticatorTest.java* e procurar o método *testDefaultAuthenticatorConstructor* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.**

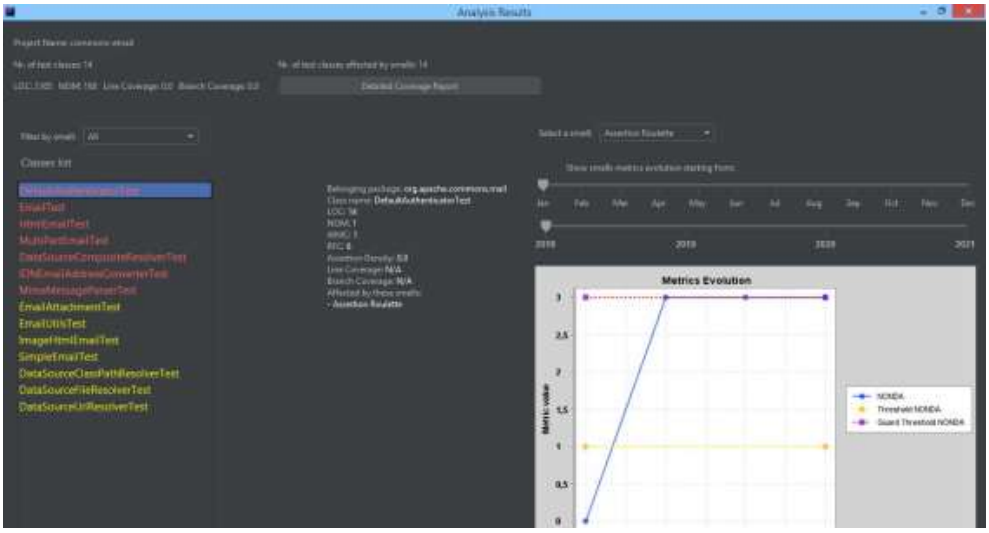
4.4 VITRuM

A ferramenta VITRuM (*VIsualization of Test-Related Metrics*) é um *plug-in* para o IntelliJ[1] para código Java que inclui o cálculo de medidas de código de teste e detecta 7 *test smells.* As medidas se dividem em dois tipos (i) estruturais referem ao tamanho, à coesão, ao acoplamento e à complexidade e (ii) dinâmicas exibem informações sobre a eficácia dos testes e cobertura de testes. Os *test smells* são exibidos através de uma lista com as classes de teste em cores diferentes (i) vermelho para classes de teste afetadas por *test smells*, com valores de medidas de código de teste acima dos limites definidos; (ii) amarelo para classes de teste afetadas por *test smells*, com valores de medidas de código de teste dentro dos limites definidos; e (iii) cinza para classes de teste sem *test smells*. Além disso, os usuários podem filtrar os resultados por *test smell* para exibir sua evolução em um gráfico durante o desenvolvimento do projeto (Figura 4.5). A seguir são detalhados as medidas e os *test smells* que a ferramenta abrange:

a) **Medidas estruturais:** LOC (*Line of Code*), AD (*Assertion Density*) e WMC (*Weigthed Methods Per Class*);
b) **Medidas dinâmicas:** *Line Coverage, Branch Coverage, Mutation Coverage e Flaky Tests.*
c) ***Test Smells:*** *Assertion Roulette, Eager Test, General Fixture, Indirect Testing, Sensitive Equality, Mystery Guest e Resource Optimism.*

[1] https://www.jetbrains.com/pt-br/idea/

Figura 4.5 - Painel Principal VITRuM



Fonte: Do autor (2021).

4.5. Treinamento 2 - VITRuM – Commons IO

VITRuM	- Acessar *File* - *Open* e carregar a pasta do projeto *Commons IO* - Acessar *Tools* - VITRuM - *Calculate Test Factors* - Deixar selecionado somente *Test Smells (default)* - Clicar em *Star Analysis* - Na janela aberta, procurar a classe *BrokenOutputStreamTest* **P1: Quais são os *test smells* que essa classe possui?** - Selecionar outra classe de teste que desejar **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Selecionar o *test smell Sensitive Equality* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** **P7: Quais são os *test smells* que esse autor é responsável?** **P8: Qual *test smell com* mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** **P9: Qual o comportamento do *test smell Assertion Roulette* da classe *AppendableOutputStreamTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais métodos? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-io/blob/master/src/test/java/org/apache/commons/io/output/AppendableOutputStreamTest.java* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.**

4.6. Tarefa 2 - VITRuM – Commons Text

VITRuM	- Acessar *File* - *Open* e carregar a pasta do projeto *Commons Text* - Acessar *Tools* - VITRuM - *Calculate Test Factors* - Deixar selecionado somente *Test Smells (default)* - Clicar em *Star Analysis* - Na janela aberta, procurar a classe *BiFunctionStringLookupTest*

	P1: Quais são os *test smells* que essa classe possui? - Selecionar outra classe de teste que desejar **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Selecionar o *test smell Sensitive Equality* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** **P7: Quais são os *test smells* que esse autor é responsável?** **P8: Qual *test smell com* mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** **P9: Qual o comportamento *do test smell Assertion Roulette* da classe *AggregateTranslatorTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais métodos? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-text/blob/master/src/test/java/org/apache/commons/text/translate/AggregateTranslatorTest.java* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.**

4.7. Questionário

Disponível em*:* *https://forms.gle/1mis7iVRHrbBSvws5*

REFERÊNCIAS

CALEGARE, A. J. **Introdução ao delineamento de experimentos**. Editora Blucher, 2009.

PALOMBA, F., BAVOTA, G., DI PENTA, M., OLIVETO, R., DE LUCIA, A., POSHYVANYK, D. **Detecting bad smells in source code using change history information**. In: International Conference on Automated Software Engineering. p. 268-278. 2013.

PECORELLI, F.; LILLO, G.; PALOMBA, F.; LUCIA, A. **VITRuM-A Plug-In for the Visualization of Test-Related Metrics**. In: International Conference on Advanced Visual Interfaces. p.1-3, 2020.

PERUMA, A.; ALMALKI, K.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A.; PALOMBA, F. **On the distribution of test smells in open source Android applications: an exploratory study**. In: International Conference on Computer Science and Software Engineering. p. 193-202, 2019.

VAN DEURSEN, A.; MOONEN, L.; VAN DEN BERGH, A.; KOK, G. **Refactoring test code**. In: Conference on extreme programming and flexible processes in software engineering. p. 92-95, 2001

VIRGÍNIO, T.; SANTANA, R.; MARTINS, L. A.; SOARES, L. R.; COSTA, H.; MACHADO, I. **On the influence of Test Smells on Test Coverage**. In: Brazilian Symposium on Software Engineering. p. 467-471, 2019.

APÊNDICE C

Perguntas Experimento Controlado G1

*Obrigatório

TSVizzEvolution

. P1: Quais são os test smells que essa classe possui? *

Marque todas que se aplicam.

- [ ] Assertion Roulette
- [ ] Conditional Test Logic
- [ ] Eager Test
- [ ] Lazy Test
- [ ] Sensitive Equality

Outro: [ ] _____

P2: Qual classe foi escolhida? Qual test smell com mais ocorrências e a quantidade? *

_____

_____

_____

_____

_____

P3: Quais são as classes que possuem o test smell Sensitive Equality? *

Marque todas que se aplicam.

- [ ] AggregateTranslatorTest
- [ ] AlphabetConverterTest
- [ ] BiFunctionStringLookupTest
- [ ] DateStringLookupTest
- [ ] DnsStringLookupTest
- [ ] EmailTest
- [ ] ExtendedMessageFormatTest
- [ ] FileStringLookupTest
- [ ] FormattableUtilsTest
- [ ] FunctionStringLookupTest
- [ ] InterpolatorStringLookupTest
- [ ] JavaPlatformStringLookupTest
- [ ] LevenshteinDetailedDistanceTest
- [ ] LocalHostStringLookupTest
- [ ] LookupTranslatorTest
- [ ] PropertiesStringLookupTest
- [ ] ResourceBundleStringLookupTest
- [ ] ScriptStringLookupTest
- [ ] StrBuilderTest
- [ ] StringTokenizerTest
- [ ] StrSubstitutorTest
- [ ] TextStringBuilderTest
- [ ] UrlDecodeStringLookupTest
- [ ] UrlEncoderStringLookupTest
- [ ] UrlStringLookupTest
- [ ] XmlStringLookupTest

Outro: [ ] _____

P4: Qual autor provavelmente é responsável por mais test smells? Qual a quantidade de ocorrências? *

Marcar apenas uma oval.

- ◯ Stefan Bodewig
- ◯ Thomas Neidhart
- ◯ Outro: _____

Informe aqui a quantidade de ocorrências: *

_____

P5: Você acha que a informação dos possíveis autores pelo test smell pode ajudar nas decisões de projeto e tomada de decisão? Explique. *

_____

_____

_____

_____

_____

P6: Quem são os possíveis autores pelo test smell Sensitive Equality? *

Marque todas que se aplicam.

- ☐ Stefan Bodewig
- ☐ Thomas Neidhart
- ☐ Gary Gregory
- ☐ Benson Mergulies

Outro: ☐ _____

P7: Quais são os test smells que esse autor é responsável? *

Marcar apenas uma oval.

- ◯ Assertion Roulette
- ◯ Conditional Test Logic
- ◯ Duplicate Assert
- ◯ Eager Test
- ◯ Exception Catching Throwing
- ◯ General Fixture
- ◯ Ignored Test
- ◯ Lazy Test
- ◯ Magic Number Test
- ◯ Mystery Guest
- ◯ Redundant Assertion
- ◯ Resource Optimism
- ◯ Sensitive Equality
- ◯ Unknown Test
- ◯ Verbose Test
- ◯ Todas as alternativas

P8: Qual test smell com mais ocorrências? Essa informação pode ser útil para o projeto? Explique. *

P9: Qual o comportamento do test smell Assertion Roulette da classe DefaultAuthenticatorTest comparando V1 e V2 (evolução)? *

Marcar apenas uma oval.

- ◯ Aumentou
- ◯ Diminuiu
- ◯ Permaneceu
- ◯ Outros

P10: O test smell Assertion Roulette ocorre em quais métodos? E em quais linhas? *

Marque todas que se aplicam.

- ☐ testNullConstructor
- ☐ testNullVarargConstructor
- ☐ testWrite
- ☐ testNonNull
- ☐ testDefaultAuthenticatorConstructor

Outro: ☐ _____

Informe aqui as linhas em que ocorre : *

_____

P11: Você conseguiria corrigir os test smells com as informações fornecidas pela ferramenta? Explique. *

_____

_____

_____

_____

_____

VITRuM

P1: Quais são os test smells que essa classe possui? *

Marque todas que se aplicam.

- [ ] Assertion Roulette
- [ ] Conditional Test Logic
- [ ] Eager Test
- [ ] Lazy Test
- [ ] Sensitive Equality

Outro: [ ] _____

P2: Qual classe foi escolhida? Qual test smell com mais ocorrências e a quantidade? *

_____

_____

_____

_____

_____

P3: Quais são as classes que possuem o test smell Sensitive Equality? *

Marque todas que se aplicam.

- [ ] AggregateTranslatorTest
- [ ] AlphabetConverterTest
- [ ] BiFunctionStringLookupTest
- [ ] DnsStringLookupTest
- [ ] ExtendedMessageFormatTest
- [ ] FileStringLookupTest
- [ ] FormattableUtilsTest
- [ ] FunctionStringLookupTest
- [ ] InterpolatorStringLookupTest
- [ ] JavaPlatformStringLookupTest
- [ ] LevenshteinDetailedDistanceTest
- [ ] LocalHostStringLookupTest
- [ ] LookupTranslatorTest
- [ ] ScriptStringLookupTest
- [ ] StrBuilderTest
- [ ] StringEscapeUtilsTest
- [ ] StringMatcherFactoryTest
- [ ] StringSubstitutorTest
- [ ] StringTokenizerTest
- [ ] StrTokenizerTest
- [ ] StrSubstitutorTest
- [ ] TextStringBuilderTest
- [ ] UrlDecodeStringLookupTest
- [ ] UrlEncoderStringLookupTest
- [ ] UrlStringLookupTest
- [ ] XmlStringLookupTest

Outro: ☐ _____

P4: Qual autor provavelmente é responsável por mais test smells? Qual a quantidade de ocorrências? *

Marcar apenas uma oval.

- ◯ Stefan Bodewig
- ◯ Thomas Neidhart
- ◯ A ferramenta não fornece essa informação
- ◯ Outro: _____

P5: Você acha que a informação dos possíveis autores pelo test smell pode ajudar nas decisões de projeto e tomada de decisão? Explique. *

_____

_____

_____

_____

_____

P6: Quem são os possíveis autores pelo test smell Sensitive Equality? *

Marque todas que se aplicam.

- ☐ Stefan Bodewig
- ☐ Thomas Neidhart
- ☐ Gary Gregory
- ☐ Benson Mergulies
- ☐ A ferramenta não fornece essa informação

Outro: ☐ _____

P7: Quais são os test smells que esse autor é responsável? *

Marcar apenas uma oval.

- ◯ Assertion Roulette
- ◯ Conditional Test Logic
- ◯ Duplicate Assert
- ◯ Eager Test
- ◯ Exception Catching Throwing
- ◯ General Fixture
- ◯ Ignored Test
- ◯ Lazy Test
- ◯ Magic Number Test
- ◯ Mystery Guest
- ◯ Redundant Assertion
- ◯ Resource Optimism
- ◯ Sensitive Equality
- ◯ Unknown Test
- ◯ Verbose Test
- ◯ Todas as alternativas
- ◯ A ferramenta não fornece essa informação

P8: Qual test smell com mais ocorrências? Essa informação pode ser útil para o projeto? Explique. *

P9: Qual o comportamento do test smell Assertion Roulette da classe AggregateTranslatorTest comparando V1 e V2 (evolução)? *

Marcar apenas uma oval.

- ◯ Aumentou
- ◯ Diminuiu
- ◯ Permaneceu
- ◯ Outros
- ◯ A ferramenta não fornece essa informação

P10: O test smell Assertion Roulette ocorre em quais métodos? E em quais linhas? *

Marque todas que se aplicam.

- ☐ testNullConstructor
- ☐ testNullVarargConstructor
- ☐ testWrite
- ☐ testNonNull
- ☐ testDefault

Outro: ☐ _____

P11: Você conseguiria corrigir os test smells com as informações fornecidas pela ferramenta? Explique. *

_____

_____

_____

_____

_____

APÊNDICE D

ADRIANA PRISCILA SANTOS CRUZ

Atividades do Estudo Experimental com Ferramentas para Visualização de *Test Smells* - Grupo 2

LAVRAS-MG
2021

Sumário

1 EXPERIMENTO CONTROLADO

As informações são lidas, com a finalidade de manter o padrão, pois trata-se de um estudo experimental e o treinamento com todos os participantes deve ser o mais semelhante. São explicados os conceitos de *test smells* e as ferramentas para a visualização de *test smells* utilizadas nesse estudo. Posteriormente, são realizadas as tarefas propostas nesse experimento.

2 *DESIGN* DO ESTUDO

O objetivo desse estudo é visualizar a ocorrência e a evolução de *test smells*. Sendo conduzido na seguinte ordem:

a) ***Test Smells*:** São explicados os conceitos e exemplificados dois *test smells*;
b) **Ferramentas para a Visualização de *Test Smells*:** São explicadas sobre duas ferramentas para a visualização de *test smells;*
c) **`VITRuM`** é a primeira ferramenta detalhada no estudo;
d) **Treinamento 1.** `VITRuM - Commons IO;`
e) **Tarefa 1.** `VITRuM – Commons Text;`
f) **`TSVizzEvolution`** é a segunda ferramenta detalhada no estudo;
g) **Treinamento 2.** `TSVizzEvolution - Commons IO;`
h) **Tarefa 2.** `TSVizzEvolution - Commons Email;`
i) **Aplicar questionário.** Questionário para caracterização dos participantes e *feedback* do experimento.

3 *TEST SMELLS*

Os *test smells* são considerados decisões ou práticas ruins de *design* no código de teste, os quais diminuem a qualidade dos testes e tornam os testes mais difíceis de serem compreendidos e mantidos durante a evolução dos sistemas de software. Dois *test smells* são explicados a seguir:

a) ***Assertion Roulette* (AR)**. Ocorre quando asserções em um método de teste não têm explicação, não é documentado e afeta a legibilidade, a compreensibilidade e a manutenção. Se uma das afirmações falhar, é difícil identificar qual falhou. O método *assertThat*() (linhas 7, 10 e 11) é "invocado" 3 vezes no método de teste. Em cada declaração, é verificada uma condição diferente, mas não é fornecida uma mensagem de explicação para cada declaração. Portanto, se uma das declarações falhar, a identificação da causa da falha não é direta.

Código 3.1 - *Exemplo de Test Smell Assertion Roulette*

```
1   @MediumTest
2   public void testCloneNonBareRepoFromLocalTestServer() throws
    Exception {
3       Clone cloneOp = new Clone(false,
    integrationGitServerURIFor("small-repo.early.git"),
    helper().newFolder());
4
5       Repository repo = executeAndWaitFor(cloneOp);
6
7       assertThat(repo,
    hasGitObject("ba1f63e4430bff267d112b1e8afc1d6294db0ccc"));
8
9       File readmeFile = new File(repo.getWorkTree(), "README");
10      assertThat(readmeFile, exists());
11      assertThat(readmeFile, ofLength(12));
12  }
```

Fonte: Adaptado de Peruma *et al*. (2019).

b) ***Sensitive Equality* (SE)**. Ocorre quando o método *toString*() é usado em métodos de teste, "invocando-o" para verificar objetos, para comparar a sua saída com a sequência específica. Alterações na implementação do método *toString*() podem resultar em falha. Na linha 15 do exemplo, o valor retornado pelo método *toString*() é comparado a *string* "/54.204.10.41", se houver alterações na implementação dos objetos do método *toString*(), pode ocorrer falha.

Código 3.2 - *Exemplo de Test Smell Sensitive Equality*

```
1. @Test
2. public void test1() throws UnknownHostException {
3.
4.      String peersPacket = "F8 4E 11 F8 4B C5 36 81 " +
5.          "CC 0A 29 82 76 5F B8 40 D8 D6 0C 25 80 FA 79 5C " +
6.          "FC 03 13 EF DE BA 86 9D 21 94 E7 9E 7C B2 B5 22 " +
7.          "F7 82 FF A0 39 2C BB AB 8D 1B AC 30 12 08 B1 37 " +
8.          "E0 DE 49 98 33 4F 3B CF 73 FA 11 7E F2 13 F8 74 " +
9.          "17 08 9F EA F8 4C 21 B0";
10.
11.       byte[] payload = Hex.decode(peersPacket);
12.
13.       byte[] ip = decodeIP4Bytes(payload, 5);
14.
15.       assertEquals(InetAddress.getByAddress(ip).toString(),
    ("/54.204.10.41"));
16.   }
```

Fonte: Adaptado de Peruma *et al.* (2019).

4 FERRAMENTAS PARA VISUALIZAÇÃO DE *TEST SMELLS*

Neste experimento, as ferramentas `TsVizzEvolution`[39] e `VITRuM`[40] são submetidas a um estudo comparativo. Esse estudo visa analisar fatores correspondentes a eficácia, e usabilidade que podem auxiliar *testers*, gerente de testes e demais envolvidos nas atividades de teste de software a visualizar mais agilmente os *test smells*, para corrigi-los e melhorar a qualidade do código de teste.

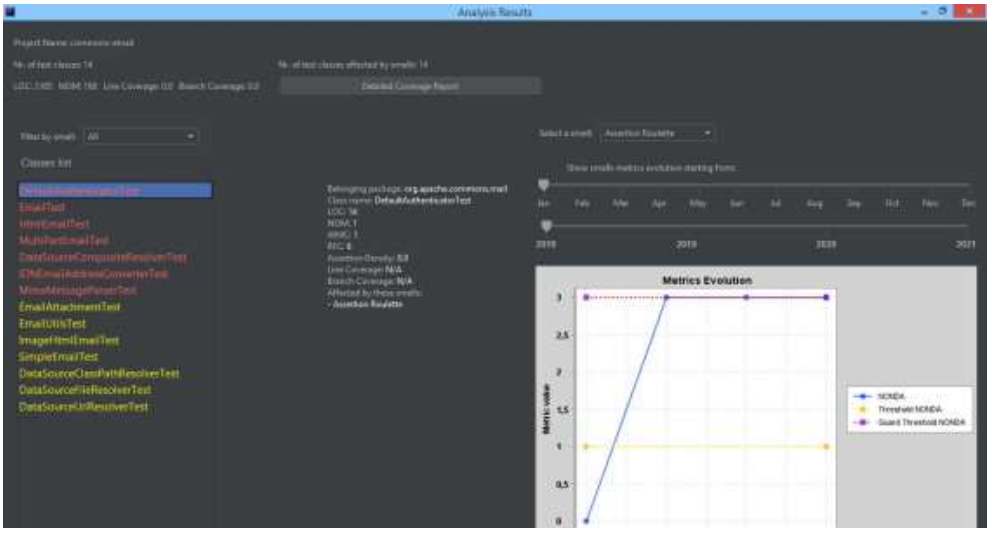
4.1 `VITRuM`

A ferramenta `VITRuM` (*VIsualization of Test-Related Metrics*) é um *plugin* para o IntelliJ[41] para código Java que inclui o cálculo de medidas de código de teste e detecta 7 *test smells.* As medidas se dividem em dois tipos (i) estruturais referem ao tamanho, à coesão, ao acoplamento e à complexidade e (ii) dinâmicas exibem informações sobre a eficácia dos testes e cobertura de testes. Os *test smells* são exibidos através de uma lista com as classes de teste em cores diferentes (i) vermelho para classes de teste afetadas por *test smells*, com valores de medidas de código de teste acima dos limites definidos; (ii) amarelo para classes de teste afetadas por *test smells*, com valores de medidas de código de teste dentro dos limites definidos; e (iii) cinza para classes de teste sem *test smells*. Além disso, os usuários podem filtrar os resultados por *test smell* para exibir sua evolução em um gráfico durante o desenvolvimento do projeto (Figura 4.5). A seguir são detalhados as medidas e os *test smells* que a ferramenta abrange:

a) **Medidas estruturais:** LOC (*Line of Code*), AD (*Assertion Density*) e WMC (*Weigthed Methods Per Class*);
b) **Medidas dinâmicas:** *Line Coverage, Branch Coverage*, *Mutation Coverage e Flaky Tests.*
c) ***Test Smells:*** *Assertion Roulette, Eager Test, General Fixture, Indirect Testing, Sensitive Equality, Mystery Guest e Resource Optimism.*

[39] https://github.com/arieslab/TSVizzEvolution
[40] https://plugins.jetbrains.com/plugin/14160-vitrum
[41] https://www.jetbrains.com/pt-br/idea/

Figura 4.1 - Painel Principal VITRuM



Fonte: Do autor (2021).

4.2 Treinamento 1 - VITRuM – Commons IO

VITRuM	- Acessar *File* - *Open* e carregar a pasta do projeto *Commons IO* - Acessar *Tools* - VITRuM - *Calculate Test Factors* - Deixar selecionado somente *Test Smells (default)* - Clicar em *Star Analysis* - Na janela aberta, procurar a classe *BrokenOutputStreamTest* **P1: Quais são os *test smells* que essa classe possui?** - Selecionar outra classe de teste que desejar **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Selecionar o *test smell Sensitive Equality* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** **P7: Quais são os *test smells* que esse autor é responsável?** **P8: Qual *test smell com* mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** **P9: Qual o comportamento do *test smell Assertion Roulette* da classe *AppendableOutputStreamTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais métodos? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-io/blob/master/src/test/java/org/apache/commons/io/output/AppendableOutputStreamTest.java* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.**

4.3 Tarefa 1 - VITRuM – Commons Text

O formulário para responder as questões está disponível em: *https://forms.gle/io91wm7HdhToGHuH7*

VITRuM	- Acessar *File* - *Open* e carregar a pasta do projeto *Commons Text* - Acessar *Tools* - VITRuM - *Calculate Test Factors* - Deixar selecionado somente *Test Smells (default)* - Clicar em *Star Analysis* - Na janela aberta, procurar a classe *BiFunctionStringLookupTest* **P1: Quais são os *test smells* que essa classe possui?** - Selecionar outra classe de teste que desejar **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Selecionar o *test smell Sensitive Equality* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** **P7: Quais são os *test smells* que esse autor é responsável?** **P8: Qual *test smell com* mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** **P9: Qual o comportamento *do test smell Assertion Roulette* da classe *AggregateTranslatorTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais métodos? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-text/blob/master/src/test/java/org/apache/commons/text/translate/AggregateTranslatorTest.java* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas? Explique.**

4.4 TSVizzEvolution

A ferramenta TSVizzEvolution utiliza três técnicas de visualização para exibir a ocorrência e evolução de 21 *test smells* (Palomba *et al.*, 2013; Peruma *et al.*, 2019; Deursen *et al.*, 2001). Esses *test smells* são detectados pela ferramenta JNose Test[42]: *Assertion Roulette, Conditional Test Logic, Constructor Initialization, Default Test, Dependent Test, Duplicate Assert, Eager Test, Empty Test, Exception Catching Throwing, General Fixture, Ignored Test, Lazy Test, Magic Number Test, Mystery Guest, Print Statement, Redundant Assertion, Resource Optimism, Sensitive Equality, Sleepy Test, Unknown Test, Verbose Test* e armazenados em um arquivo .csv. A ferramenta TSVizzEvolution estrutura os dados do arquivo .csv conforme visualização escolhida, a análise de *test smells* pode ser realizada para uma versão do código de teste (Análise Única) ou para duas versões (Análise de Evolução).

4.4.1 Análise Única

Para a análise das ocorrências de *test smells* para uma versão do código de teste estão disponíveis duas técnicas: *Graph View* e *Treemap View*, essas técnicas são explicadas a seguir:

[42] http://jnose.herokuapp.com/

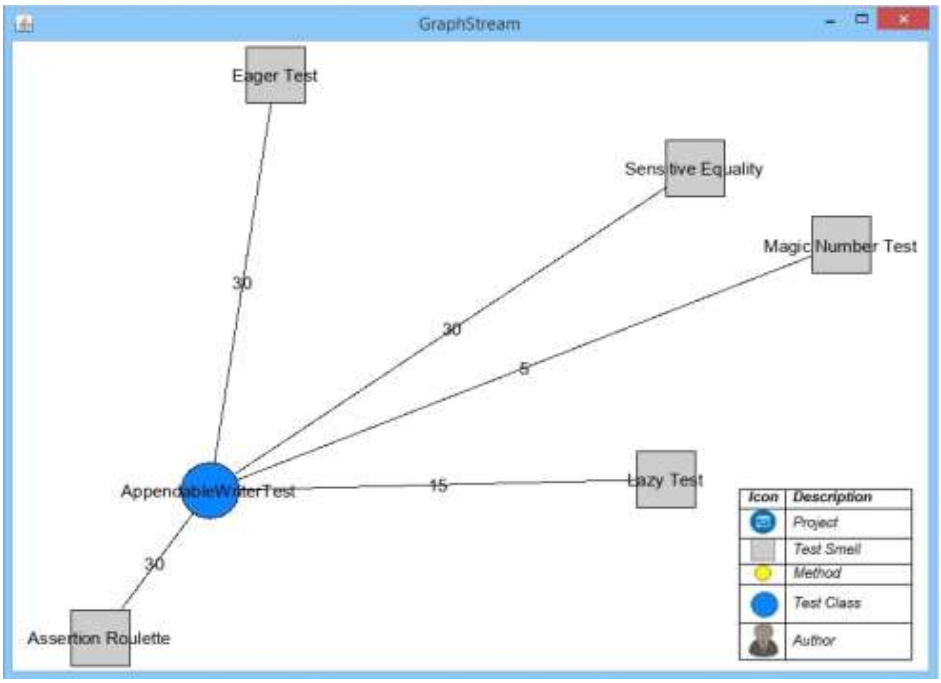
a) ***Graph View***. Os dados retornados pela `JNose` são estruturados em grafos. Os componentes do grafo variam de acordo com a granularidade escolhida, tem-se seis granularidades disponíveis (Quadro 4.1). Na Figura 4.1 verifica-se um grafo e na Figura 4.2 o detalhamento da legenda, que exibe o significado dos ícones utilizados como nós no grafo. Os componentes podem ser arrastados para ajustar a visualização conforme necessidade.

Quadro 4.1 - Granularidade da Ferramenta

Granularidade	**Descrição**
Project	Exibe a relação entre todo o projeto e os *test smells* (1:N).
All Test Classes	Exibe a relação entre as classes de teste e os *test smells* (N:N).
A Specific Test Class	Exibe a relação entre as classes de teste e *test smells* (1:N). Para executar a ferramenta com essa granularidade, os usuários precisam selecionar uma classe de teste.
A Specific Test Smells	Exibe a relação entre as classes de teste e os *test smells* (N:1). Para executar a ferramenta com essa granularidade, os usuários devem selecionar um *test smell*.
Author	Exibe a relação entre os autores, os *test smells* (N:N) e as classes de teste (1: N). Para utilizar a ferramenta com essa granularidade, os usuários devem selecionar o autor (um específico ou todos autores) e um *test smell*.
Methods	Exibe a relação entre o *test smell*, a classe de teste e os métodos (1:N). Os usuários precisam selecionar um *test smell* e uma classe de teste.

Fonte: Do autor (2021).

Figura 4.2 - *Graph View*



Fonte: Do autor (2021).

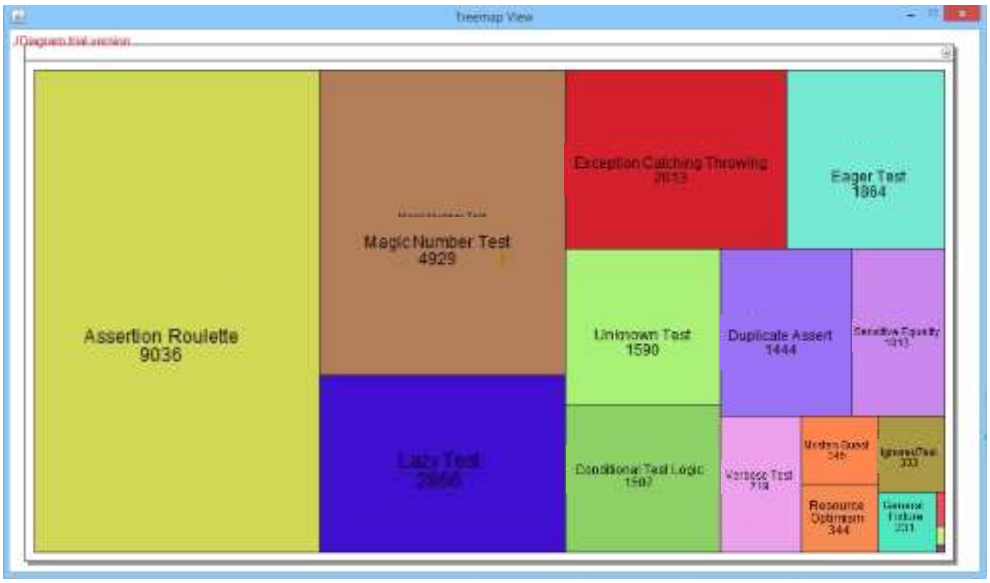
Figura 4.3 - Legenda *Graph View*

Icon	Description
	Project
	Test Smell
	Method
	Test Class
	Author

Fonte: Do autor (2021).

b) ***Treemap View***. Os *test smells* são representados por retângulos, onde cada retângulo possui uma cor, gerada aleatoriamente. Além disso, os tamanhos dos retângulos representam as suas quantidades de ocorrências, ou seja, quanto maior o retângulo mais quantidade de ocorrências de *test smells* (Figura 4.3).

Figura 4.4 - *Treemap View*



Fonte: Do autor (2021).

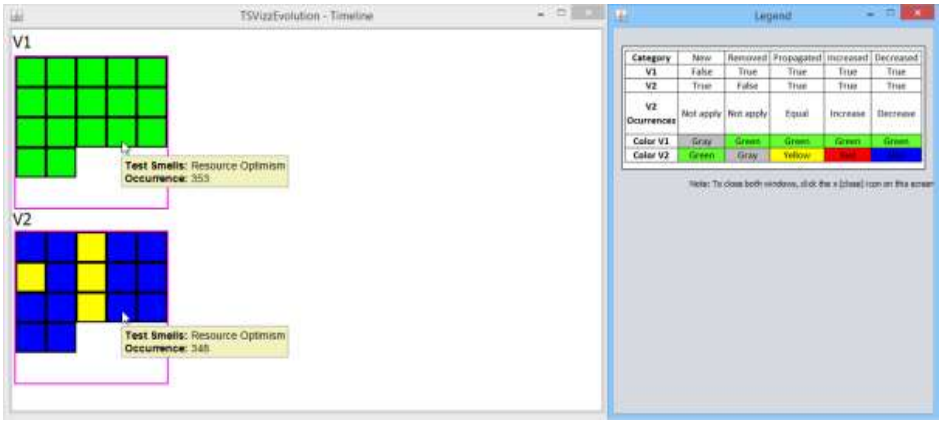
4.4.2 Análise de Evolução

Para a análise das ocorrências de *test smells* para duas versões é utilizada a técnica *Timeline View*, essa técnica é explicada a seguir.

a) ***Timeline View***. Os usuários podem escolher entre três granularidades: i) *Project*; ii) *All Test Classes;* ou *iii) Methods*. Os artefatos (projeto, classe de teste e *test smells*) do código de teste são representados por retângulos e as versões são apresentadas em paralelo horizontalmente, conforme ordem de inserção dos arquivos `.csv`. Os retângulos são aninhados da seguinte forma: i) **retângulo externo** representa o projeto ou a classe de teste

(conforme granularidade selecionada) possui borda na **cor rosa**; e ii) **retângulos menores** representam cada *test smell* detectado em uma classe de teste ou projeto, com as cores conforme comportamento durante a evolução, podendo ser verde, azul, cinza ou vermelho.

Figura 4.5 - *Timeline View*



Fonte: Do autor (2021).

As posições dos retângulos nas duas versões são mantidas para facilitar a rastreabilidade entre classes de teste e *test smells*. As cores dos *test smells* são categorizados conforme sua existência e quantidade de ocorrências (Quadro 4.2) e apresentadas ao lado da *timeline* (Figura 4.4). Os usuários podem ver os nomes das classes de teste, projeto e *test smells* e suas quantidade de ocorrências ao posicionar o *mouse* sobre cada retângulo. As categorias são (Versão um - V_1 e Versão dois - V2):

- **Novo**. Se V_1 não tiver o *test smell* específico (*False*) e V_2 (*True*), a categoria é "**New**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**cinza**" e "**verde**", respectivamente;
- **Removido**. Se V_1 tiver o *test smell* específico (*True*) e V_2 (*False*), a categoria é "**Removido**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**verde**" e "**cinza**", respectivamente;
- **Propagado**. Se V_1 e V_2 tiverem o *test smell* específico (*True* - *True*) e a quantidade for a mesma (*Igual*), a categoria é "**Propagado**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**verde**" e "**amarelo**", respectivamente;
- **Aumentado**. Se V_1 e V_2 tiverem um *test smell* específico (*True* - *True*), mas a quantidade de *test smells* aumenta em V_2 (**Aumenta**), a categoria é "**Aumentado**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem a cores "**verde**" e "**vermelho**", respectivamente;

– **Diminuído**. Se V_1 e V_2 tiverem um *test smell* específico (*True* - *True*), mas a quantidade de *test smells* diminui na V_2 (**Diminui**), a categoria é "**Diminuído**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem a cores "**verde**" e "**azul**".

Quadro 4.2 - Classificação dos *Test Smells*

Categoria	**V_1**	**V_2**	**V_2 Ocorrências**	**Cor V_1**	**Cor V_2**
Novo	*False*	*True*	Não se aplica	Cinza	Verde
Removido	*True*	*False*	Não se aplica	Verde	Cinza
Propagado	*True*	*True*	Igual	Verde	Amarelo
Aumentado	*True*	*True*	Aumentado	Verde	Vermelho
Diminuído	*True*	*True*	Diminuído	Verde	Azul

Fonte: Do autor (2021).

4.5 Treinamento 2 - `TSVizzEvolution - Commons IO`

`TSVizzEvolution`	- Selecionar 1 versão - Carregar o arquivo "*resultado_evolution1.csv*" da pasta *commons-io-2.6* - Escolher a técnica *Graph View* - Selecionar a granularidade *A Specific Test Class* - Selecionar a classe de teste *BrokenOutputStreamTest* e clicar em *Generate Graph View* **P1: Quais são os *test smells* que essa classe possui?** - Fechar a tela do grafo gerado e selecionar outra classe de teste que desejar e clicar em *Generate Graph View* **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Selecionar a granularidade *A Specific Test Smells* - Selecionar o *test smell Sensitive Equality* e clicar em *Generate Graph View* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** - Fechar a janela do grafo e selecionar a Granularidade *Author* - Selecionar o *Author: All*, que corresponde a todos autores - Selecionar o *test smell Sensitive Equality* **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** - Fechar a janela do grafo - Selecionar o *Author* Benson Margulies **P7: Quais são os *test smells* que esse autor é responsável?** - Selecionar a *View Type: Treemap View* e clicar em *Generate Treemap View* **P8: Qual *test smell com* mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** - Fechar a janela da *Treemap* - Fechar a janela *TSVizzEvolution - One Version* - Selecionar 2 versões - Carregar o arquivo "*resultado_evolution1.csv*" da pasta *commons-io-2.1* em "*Select the first .csv*"

	- Carregar o arquivo "*resultado_evolution1.csv"* da pasta *commons-io-2.6* em "*Select the second .csv*" - Selecionar *Methods* em "*Select the level of granularity"* - Carregar o arquivo fornecido "*commons-io_result_byclasstest_testsmells.csv*" da pasta *commons-io-2.1* em "*Select the first .csv File*" - Carregar o arquivo "*commons-io_result_byclasstest_testsmells.csv"* da pasta *commons-io-2.6 "Select the second .csv File" e clicar em Generate Timeline View* - Passar o mouse sobre a primeira classe: *AppendableOutputStreamTest* **P9: Qual o comportamento do *test smell Assertion Roulette* da classe *AppendableOutputStreamTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais método? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-io/blob/master/src/test/java/org/apache/commons/io/output/AppendableOutputStreamTest.java* e procurar o método *testWriteInt* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.** - Clicar em fechar ao na janela de título *Legend*

4.6 Tarefa 2 - `TSVizzEvolution - Commons Email`

`TSVizzEvolution`	- Selecionar 1 versão - Carregar o arquivo "*resultado_evolution1.csv*" da pasta *commons-email-1.9* - Escolher a técnica *Graph View* - Selecionar a granularidade *A Specific Test Class* - Selecionar a classe de teste *IDNEmailAddressConvertTest* e clicar em *Generate Graph View* **P1: Quais são os test smells que essa classe possui?** - Fechar a tela do grafo gerado e selecionar outra classe de teste que desejar e clicar em *Generate Graph View* **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Fechar a janela do grafo e selecionar a Granularidade *A Specific Test Smells* - Selecionar o *test smell Sensitive Equality* e clicar em *Generate Graph View* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** - Fechar a janela do grafo e selecione a Granularidade *Author* - Selecionar o *Author: All*, que corresponde a todos autores - Selecionar o *test smells Sensitive Equality* **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** - Fechar a janela do grafo - Selecionar o *Author Stefan Bodewig* **P7: Quais são os *test smells* que esse autor é responsável?** - Selecionar a *View Type*: *Treemap View* e clicar em *Generate Treemap View* **P8: Qual *test smell* com mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** - Fechar a janela da *Treemap* - Fechar a janela *TSVizzEvolution - One Version* - Selecionar 2 versões - Carregar o arquivo "resultado_evolution1.csv" da pasta *commons-email-1.0* em "*Select the first .csv*" - Carregar o arquivo "*resultado_evolution1.csv"* da pasta *commons-email-1.9* em "*Select the second .csv*"

	- Selecionar *Methods* em "*Select the level of granularity"* - Carregar o arquivo fornecido "*commons-email_result_byclasstest_testsmells.csv*" da pasta *commons-email-1.0* em "*Select the first .csv File*" - Carregar o arquivo "*commons-email_result_byclasstest_testsmells.csv"* da pasta *commons-email-1.9 "Select the second .csv File"* e clicar em *Generate Timeline View* - Passar o mouse sobre a quinta classe: *DefaultAuthenticatorTest* **P9: Qual o comportamento do *test smell Assertion Roulette da classe DefaultAuthenticatorTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais método? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-email/blob/master/src/test/java/org/apache/commons/mail/DefaultAuthenticatorTest.java* e procurar o método *testDefaultAuthenticatorConstructor* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.**

4.7 Questionário

Disponível em*:* *https://forms.gle/1mis7iVRHrbBSvws5*

REFERÊNCIAS

CALEGARE, A. J. **Introdução ao delineamento de experimentos**. Editora Blucher, 2009.

PALOMBA, F., BAVOTA, G., DI PENTA, M., OLIVETO, R., DE LUCIA, A., POSHYVANYK, D. **Detecting bad smells in source code using change history information**. In: International Conference on Automated Software Engineering. p. 268-278. 2013.

PECORELLI, F.; LILLO, G.; PALOMBA, F.; LUCIA, A. **VITRuM-A Plug-In for the Visualization of Test-Related Metrics**. In: International Conference on Advanced Visual Interfaces. p.1-3, 2020.

PERUMA, A.; ALMALKI, K.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A.; PALOMBA, F. **On the distribution of test smells in open source Android applications: an exploratory study**. In: International Conference on Computer Science and Software Engineering. p. 193-202, 2019.

VAN DEURSEN, A.; MOONEN, L.; VAN DEN BERGH, A.; KOK, G. **Refactoring test code**. In: Conference on extreme programming and flexible processes in software engineering. p. 92-95, 2001

VIRGÍNIO, T.; SANTANA, R.; MARTINS, L. A.; SOARES, L. R.; COSTA, H.; MACHADO, I. **On the influence of Test Smells on Test Coverage**. In: Brazilian Symposium on Software Engineering. p. 467-471, 2019.

APÊNDICE E

Perguntas Experimento Controlado G2

*Obrigatório

VITRuM

P1: Quais são os test smells que essa classe possui? *

Marque todas que se aplicam.

- [ ] Assertion Roulette
- [ ] Conditional Test Logic
- [ ] Eager Test
- [ ] Lazy Test
- [ ] Sensitive Equality

Outro: [ ] _____

P2: Qual classe foi escolhida? Qual test smell com mais ocorrências e a quantidade? *

_____

_____

_____

_____

_____

P3: Quais são as classes que possuem o test smell Sensitive Equality? *

Marque todas que se aplicam.

- [ ] AggregateTranslatorTest
- [ ] AlphabetConverterTest
- [ ] BiFunctionStringLookupTest
- [ ] DateStringLookupTest
- [ ] DnsStringLookupTest
- [ ] ExtendedMessageFormatTest
- [ ] FileStringLookupTest
- [ ] FormattableUtilsTest
- [ ] FunctionStringLookupTest
- [ ] InterpolatorStringLookupTest
- [ ] JavaPlatformStringLookupTest
- [ ] LevenshteinDetailedDistanceTest
- [ ] LocalHostStringLookupTest
- [ ] LookupTranslatorTest
- [ ] PropertiesStringLookupTest
- [ ] ResourceBundleStringLookupTest
- [ ] ScriptStringLookupTest
- [ ] StrBuilderTest
- [ ] StringEscapeUtilsTest
- [ ] StringMatcherFactoryTest
- [ ] StringSubstitutorTest
- [ ] StringTokenizerTest
- [ ] StrTokenizerTest
- [ ] StrSubstitutorTest
- [ ] TextStringBuilderTest
- [ ] UrlDecodeStringLookupTest
- [ ] UrlEncoderStringLookupTest
- [ ] UrlStringLookupTest
- [ ] XmlStringLookupTest

Outro: ☐ _____

P4: Qual autor provavelmente é responsável por mais test smells? Qual a quantidade de ocorrências? *

Marcar apenas uma oval.

- ◯ Stefan Bodewig
- ◯ Thomas Neidhart
- ◯ A ferramenta não fornece essa informação
- ◯ Outro: _____

Informe aqui a quantidade de ocorrências:

_____

P5: Você acha que a informação dos possíveis autores pelo test smell pode ajudar nas decisões de projeto e tomada de decisão? Explique. *

_____

_____

_____

_____

_____

P6: Quem são os possíveis autores pelo test smell Sensitive Equality? *

Marque todas que se aplicam.

- ☐ Stefan Bodewig
- ☐ Thomas Neidhart
- ☐ Gary Gregory
- ☐ Benson Mergulies
- ☐ A ferramenta não fornece essa informação

Outro: ☐ _____

P7: Quais são os test smells que esse autor é responsável? *

Marcar apenas uma oval.

- ◯ Assertion Roulette
- ◯ Conditional Test Logic
- ◯ Duplicate Assert
- ◯ Eager Test
- ◯ Exception Catching Throwing
- ◯ General Fixture
- ◯ Ignored Test
- ◯ Lazy Test
- ◯ Magic Number Test
- ◯ Mystery Guest
- ◯ Redundant Assertion
- ◯ Resource Optimism
- ◯ Sensitive Equality
- ◯ Unknown Test
- ◯ Verbose Test
- ◯ Todas as alternativas
- ◯ A ferramenta não fornece essa informação

P8: Qual test smell com mais ocorrências? Essa informação pode ser útil para o projeto? Explique. *

P9: Qual o comportamento do test smell Assertion Roulette da classe EmailTest comparando V1 e V2 (evolução)? *

Marcar apenas uma oval.

- ◯ Aumentou
- ◯ Diminuiu
- ◯ Permaneceu
- ◯ Outros
- ◯ A ferramenta não fornece essa informação

P10: O test smell Assertion Roulette ocorre em quais métodos? E em quais linhas? *

Marque todas que se aplicam.

- ☐ testNullConstructor
- ☐ testNullVarargConstructor
- ☐ testWrite
- ☐ testNonNull
- ☐ testDefault
- ☐ A ferramenta não fornece essa opção

Outro: ☐ _____

P11: Você conseguiria corrigir os test smells com as informações fornecidas pela ferramenta? Explique. *

_____

_____

_____

_____

_____

TSVizzEvolution

P1: Quais são os test smells que essa classe possui? *

Marque todas que se aplicam.

- [ ] Assertion Roulette
- [ ] Conditional Test Logic
- [ ] Eager Test
- [ ] Lazy Test
- [ ] Sensitive Equality

Outro: [ ] _____

P2: Qual classe foi escolhida? Qual test smell com mais ocorrências e a quantidade? *

_____

_____

_____

_____

_____

P3: Quais são as classes que possuem o test smell Sensitive Equality? *

Marque todas que se aplicam.

- [ ] AggregateTranslatorTest
- [ ] AlphabetConverterTest
- [ ] BiFunctionStringLookupTest
- [ ] DateStringLookupTest
- [ ] DnsStringLookupTest
- [ ] EmailTest
- [ ] ExtendedMessageFormatTest
- [ ] FileStringLookupTest
- [ ] FormattableUtilsTest
- [ ] FunctionStringLookupTest
- [ ] InterpolatorStringLookupTest
- [ ] JavaPlatformStringLookupTest
- [ ] LevenshteinDetailedDistanceTest
- [ ] LocalHostStringLookupTest
- [ ] LookupTranslatorTest
- [ ] PropertiesStringLookupTest
- [ ] ResourceBundleStringLookupTest
- [ ] ScriptStringLookupTest
- [ ] StrBuilderTest
- [ ] StringTokenizerTest
- [ ] StrSubstitutorTest
- [ ] TextStringBuilderTest
- [ ] UrlDecodeStringLookupTest
- [ ] UrlEncoderStringLookupTest
- [ ] UrlStringLookupTest
- [ ] XmlStringLookupTest

Outro: ☐ _____

P4: Qual autor provavelmente é responsável por mais test smells? Qual a quantidade de ocorrências? *

Marcar apenas uma oval.

- ◯ Stefan Bodewig
- ◯ Thomas Neidhart
- ◯ Outro: _____

*

_____

P5: Você acha que a informação dos possíveis autores pelo test smell pode ajudar nas decisões de projeto e tomada de decisão? Explique. *

_____

_____

_____

_____

_____

P6: Quem são os possíveis autores pelo test smell Sensitive Equality? *

Marque todas que se aplicam.

- ☐ Stefan Bodewig
- ☐ Thomas Neidhart
- ☐ Gary Gregory
- ☐ Benson Mergulies

Outro: ☐ _____

P7: Quais são os test smells que esse autor é responsável? *

Marcar apenas uma oval.

- ◯ Assertion Roulette
- ◯ Conditional Test Logic
- ◯ Duplicate Assert
- ◯ Eager Test
- ◯ Exception Catching Throwing
- ◯ General Fixture
- ◯ Ignored Test
- ◯ Lazy Test
- ◯ Magic Number Test
- ◯ Mystery Guest
- ◯ Redundant Assertion
- ◯ Resource Optimism
- ◯ Sensitive Equality
- ◯ Unknown Test
- ◯ Verbose Test
- ◯ Todas as alternativas

P8: Qual test smell com mais ocorrências? Essa informação pode ser útil para o projeto? Explique. *

P9: Qual o comportamento do test smell Assertion Roulette da classe DefaultAuthenticatorTest comparando V1 e V2 (evolução)? *

Marcar apenas uma oval.

- ◯ Aumentou
- ◯ Diminuiu
- ◯ Permaneceu
- ◯ Outros

P10: O test smell Assertion Roulette ocorre em quais métodos? E em quais linhas? *

Marque todas que se aplicam.

- ☐ testNullConstructor
- ☐ testNullVarargConstructor
- ☐ testWrite
- ☐ testNonNull
- ☐ testDefaultAuthenticatorConstructor

Outro: ☐ _____

Informe aqui em quais linhas ocorre: *

P11: Você conseguiria corrigir os test smells com as informações fornecidas pela ferrramenta? Explique. *

APÊNDICE F

ADRIANA PRISCILA SANTOS CRUZ

Atividades do Estudo Experimental com Ferramentas para Visualização de *Test Smells* - Grupo 3

LAVRAS-MG
2021

Sumário

1 EXPERIMENTO CONTROLADO

As informações são lidas, com a finalidade de manter o padrão, pois trata-se de um estudo experimental e o treinamento com todos os participantes deve ser o mais semelhante. São explicados os conceitos de *test smells* e as ferramentas para a visualização de *test smells* utilizadas nesse estudo. Posteriormente, são realizadas as tarefas propostas nesse experimento.

2 *DESIGN* DO ESTUDO

O objetivo desse estudo é visualizar a ocorrência e a evolução de *test smells*. Sendo conduzido na seguinte ordem:

a) ***Test Smells*:** São explicados os conceitos e exemplificados dois *test smells*;
b) **Ferramentas para a Visualização de *Test Smells*:** São explicadas sobre duas ferramentas para a visualização de *test smells;*
c) **`TSVizzEvolution`** é a primeira ferramenta detalhada no estudo;
d) **Treinamento 1.** `TSVizzEvolution - Commons IO;`
e) **Tarefa 1.** `TSVizzEvolution – Commons Text;`
f) **`VITRuM`** é a segunda ferramenta detalhada no estudo;
g) **Treinamento 2.** `VITRuM - Commons IO;`
h) **Tarefa 2.** `VITRuM - Commons Email;`
i) **Aplicar questionário.** Questionário para caracterização dos participantes e *feedback* do experimento.

3 *TEST SMELLS*

Os *test smells* são considerados decisões ou práticas ruins de design no código de teste, os quais diminuem a qualidade dos testes e tornam os testes mais difíceis de serem compreendidos e mantidos durante a evolução dos sistemas de software. Dois *test smells* são explicados a seguir:

a) ***Assertion Roulette* (AR)**. Ocorre quando asserções em um método de teste não têm explicação, não é documentado e afeta a legibilidade, a compreensibilidade e a manutenção. Se uma das afirmações falhar, é difícil identificar qual falhou. O método *assertThat*() (linhas 7, 10 e 11) é "invocado" 3 vezes no método de teste. Em cada declaração, é verificada uma condição diferente, mas não é fornecida uma mensagem de explicação para cada declaração. Portanto, se uma das declarações falhar, a identificação da causa da falha não é direta.

Código 3.1 - *Exemplo de Test Smell Assertion Roulette*

```
1   @MediumTest
2   public void testCloneNonBareRepoFromLocalTestServer() throws
    Exception {
3       Clone cloneOp = new Clone(false,
    integrationGitServerURIFor("small-repo.early.git"),
    helper().newFolder());
4
5       Repository repo = executeAndWaitFor(cloneOp);
6
7       assertThat(repo,
    hasGitObject("ba1f63e4430bff267d112b1e8afc1d6294db0ccc"));
8
9       File readmeFile = new File(repo.getWorkTree(), "README");
10      assertThat(readmeFile, exists());
11      assertThat(readmeFile, ofLength(12));
12  }
```

Fonte: Adaptado de Peruma *et al.* (2019).

b) ***Sensitive Equality* (SE)**. Ocorre quando o método *toString*() é usado em métodos de teste, "invocando-o" para verificar objetos, para comparar a sua saída com a sequência específica. Alterações na implementação do método *toString*() podem resultar em falha. Na linha 15 do exemplo, o valor retornado pelo método *toString*() é comparado a *string* "/54.204.10.41", se houver alterações na implementação dos objetos do método *toString*(), pode ocorrer falha.

Código 3.2 - *Exemplo de Test Smell Sensitive Equality*

```
1.  @Test
2.  public void test1() throws UnknownHostException {
3.
4.      String peersPacket = "F8 4E 11 F8 4B C5 36 81 " +
5.          "CC 0A 29 82 76 5F B8 40 D8 D6 0C 25 80 FA 79 5C " +
6.          "FC 03 13 EF DE BA 86 9D 21 94 E7 9E 7C B2 B5 22 " +
7.          "F7 82 FF A0 39 2C BB AB 8D 1B AC 30 12 08 B1 37 " +
8.          "E0 DE 49 98 33 4F 3B CF 73 FA 11 7E F2 13 F8 74 " +
9.          "17 08 9F EA F8 4C 21 B0";
10.
11.       byte[] payload = Hex.decode(peersPacket);
12.
13.       byte[] ip = decodeIP4Bytes(payload, 5);
14.
15.       assertEquals(InetAddress.getByAddress(ip).toString(),
    ("/54.204.10.41"));
16.   }
```

Fonte: Adaptado de Peruma *et al.* (2019).

4 FERRAMENTAS PARA VISUALIZAÇÃO DE *TEST SMELLS*

Neste experimento, as ferramentas `TsVizzEvolution`[43] e `VITRuM`[44] são submetidas a um estudo comparativo. Esse estudo visa analisar fatores correspondentes a eficácia, e usabilidade que podem auxiliar *testers*, gerente de testes e demais envolvidos nas atividades de teste de software a visualizar mais agilmente os *test smells*, para corrigi-los e melhorar a qualidade do código de teste.

4.1 `TSVizzEvolution`

A ferramenta `TSVizzEvolution` utiliza três técnicas de visualização para exibir a ocorrência e evolução de 21 *test smells (*Palomba *et al.*, 2013; Peruma *et al.*, 2019; Deursen *et al.*, 2001). Esses *test smells* são detectados pela ferramenta `JNose Test`[45]: *Assertion Roulette, Conditional Test Logic, Constructor Initialization, Default Test, Dependent Test, Duplicate Assert, Eager Test, Empty Test, Exception Catching Throwing, General Fixture, Ignored Test, Lazy Test, Magic Number Test, Mystery Guest, Print Statement, Redundant Assertion, Resource Optimism, Sensitive Equality, Sleepy Test, Unknown Test, Verbose Test* e armazenados em um arquivo `.csv`. A ferramenta `TSVizzEvolution` estrutura os dados do arquivo `.csv` conforme visualização escolhida, a análise de *test smells* pode ser realizada para uma versão do código de teste (Análise Única) ou para duas versões (Análise de Evolução).

4.1.1 Análise Única

Para a análise das ocorrências de *test smells* para uma versão do código de teste estão disponíveis duas técnicas: *Graph View* e *Treemap View*, essas técnicas são explicadas a seguir:

a) ***Graph View***. Os dados retornados pela `JNose` são estruturados em grafos. Os componentes do grafo variam de acordo com a granularidade escolhida, tem-se seis granularidades disponíveis (Quadro 4.1). Na Figura 4.1 verifica-se um grafo e na Figura 4.2 o detalhamento da legenda, que exibe o significado dos ícones utilizados como nós no grafo. Os componentes podem ser arrastados para ajustar a visualização conforme necessidade.

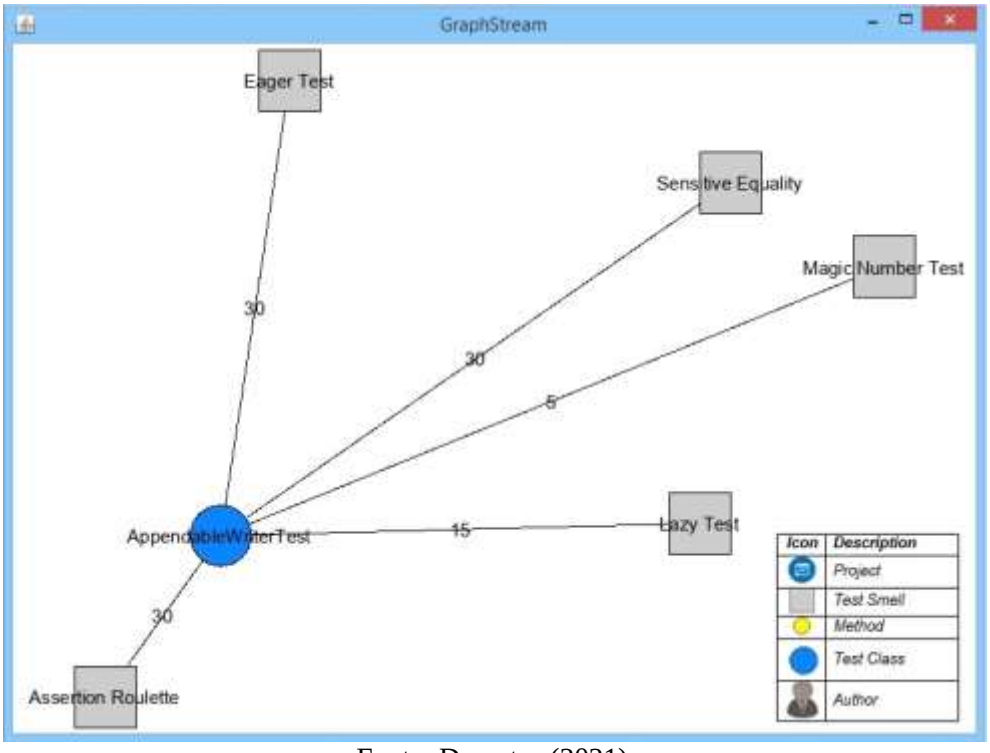
[43] https://github.com/arieslab/TSVizzEvolution
[44] https://plugins.jetbrains.com/plugin/14160-vitrum
[45] http://jnose.herokuapp.com/

Quadro 4.1 - Granularidade da Ferramenta

Granularidade	Descrição
Project	Exibe a relação entre todo o projeto e os *test smells* (1:N).
All Test Classes	Exibe a relação entre as classes de teste e os *test smells* (N:N).
A Specific Test Class	Exibe a relação entre as classes de teste e *test smells* (1:N). Para executar a ferramenta com essa granularidade, os usuários precisam selecionar uma classe de teste.
A Specific Test Smells	Exibe a relação entre as classes de teste e os *test smells* (N:1). Para executar a ferramenta com essa granularidade, os usuários devem selecionar um *test smell*.
Author	Exibe a relação entre os autores, os *test smells* (N:N) e as classes de teste (1: N). Para utilizar a ferramenta com essa granularidade, os usuários devem selecionar o autor (um específico ou todos autores) e um *test smell*.
Methods	Exibe a relação entre o *test smell*, a classe de teste e os métodos (1:N). Os usuários precisam selecionar um *test smell* e uma classe de teste.

Fonte: Do autor (2021).

Figura 4.1 - *Graph View*



Fonte: Do autor (2021).

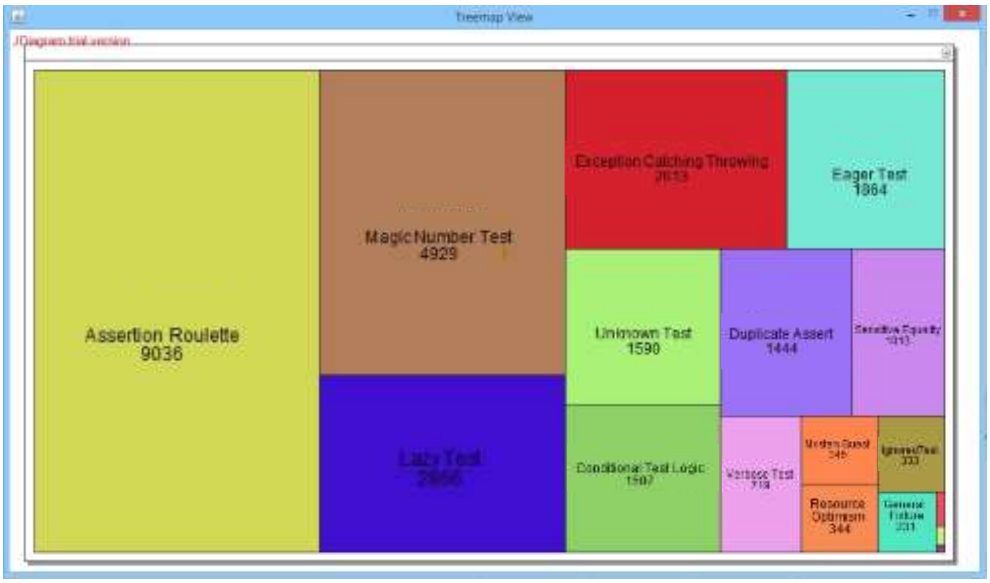
Figura 4.2 - Legenda *Graph View*

Icon	Description
	Project
	Test Smell
	Method
	Test Class
	Author

Fonte: Do autor (2021).

b) ***Treemap View***. Os *test smells* são representados por retângulos, onde cada retângulo possui uma cor, gerada aleatoriamente. Além disso, os tamanhos dos retângulos representam as suas quantidades de ocorrências, ou seja, quanto maior o retângulo mais quantidade de ocorrências de *test smells* (Figura 4.3).

Figura 4.3 - *Treemap View*



Fonte: Do autor (2021).

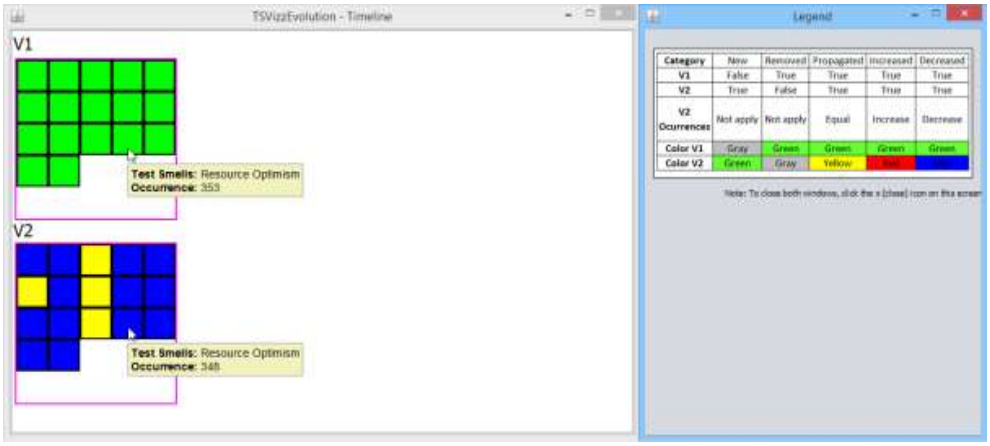
4.1.2 Análise de Evolução

Para a análise das ocorrências de *test smells* para duas versões é utilizada a técnica *Timeline View*, essa técnica é explicada a seguir.

a) ***Timeline View***. Os usuários podem escolher entre três granularidades: i) *Project*; ii) *All Test Classes;* ou *iii) Methods*. Os artefatos (projeto, classe de teste e *test smells*) do código de teste são representados por retângulos e as versões são apresentadas em paralelo horizontalmente, conforme ordem de inserção dos arquivos `.csv`. Os retângulos são aninhados da seguinte forma: i) **retângulo externo** representa o projeto ou a classe de teste

(conforme granularidade selecionada) possui borda na **cor rosa**; e ii) **retângulos menores** representam cada *test smell* detectado em uma classe de teste ou projeto, com as cores conforme comportamento durante a evolução, podendo ser verde, azul, cinza ou vermelho.

Figura 4.4 - *Timeline View*



Fonte: Do autor (2021).

As posições dos retângulos nas duas versões são mantidas para facilitar a rastreabilidade entre classes de teste e *test smells*. As cores dos *test smells* são categorizados conforme sua existência e quantidade de ocorrências (Quadro 4.2) e apresentadas ao lado da *timeline* (Figura 4.4). Os usuários podem ver os nomes das classes de teste, projeto e *test smells* e suas quantidade de ocorrências ao posicionar o *mouse* sobre cada retângulo. As categorias são (Versão um - V_1 e Versão dois - V2):

- **Novo**. Se V_1 não tiver o *test smell* específico (*False*) e V_2 (*True*), a categoria é "**New**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**cinza**" e "**verde**", respectivamente;
- **Removido**. Se V_1 tiver o *test smell* específico (*True*) e V_2 (*False*), a categoria é "**Removido**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**verde**" e "**cinza**", respectivamente;
- **Propagado**. Se V_1 e V_2 tiverem o *test smell* específico (*True* - *True*) e a quantidade for a mesma (*Igual*), a categoria é "**Propagado**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**verde**" e "**amarelo**", respectivamente;
- **Aumentado**. Se V_1 e V_2 tiverem um *test smell* específico (*True* - *True*), mas a quantidade de *test smells* aumenta em V_2 (**Aumenta**), a categoria é "**Aumentado**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem a cores "**verde**" e "**vermelho**", respectivamente;

– **Diminuído**. Se V_1 e V_2 tiverem um *test smell* específico (*True* - *True*), mas a quantidade de *test smells* diminui na V_2 (**Diminui**), a categoria é "**Diminuído**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem a cores "**verde**" e "**azul**".

Quadro 4.2 - Classificação dos *Test Smells*

Categoria	**V_1**	**V_2**	**V_2 Ocorrências**	**Cor V_1**	**Cor V_2**
Novo	*False*	*True*	Não se aplica	Cinza	Verde
Removido	*True*	*False*	Não se aplica	Verde	Cinza
Propagado	*True*	*True*	Igual	Verde	Amarelo
Aumentado	*True*	*True*	Aumentado	Verde	Vermelho
Diminuído	*True*	*True*	Diminuído	Verde	Azul

Fonte: Do autor (2021).

4.2 Treinamento 1 - `TSVizzEvolution - Commons IO`

`TSVizzEvolution`	- Selecionar 1 versão - Carregar o arquivo "*resultado_evolution1.csv*" da pasta *commons-io-2.6* - Escolher a técnica *Graph View* - Selecionar a granularidade *A Specific Test Class* - Selecionar a classe de teste *BrokenOutputStreamTest* e clicar em *Generate Graph View* **P1: Quais são os *test smells* que essa classe possui?** - Fechar a tela do grafo gerado e selecionar outra classe de teste que desejar e clicar em *Generate Graph View* **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Selecionar a granularidade *A Specific Test Smells* - Selecionar o *test smell Sensitive Equality* e clicar em *Generate Graph View* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** - Fechar a janela do grafo e selecionar a Granularidade *Author* - Selecionar o *Author: All*, que corresponde a todos autores - Selecionar o *test smell Sensitive Equality* **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** - Fechar a janela do grafo - Selecionar o *Author* Benson Margulies **P7: Quais são os *test smells* que esse autor é responsável?** - Selecionar a *View Type: Treemap View* e clicar em *Generate Treemap View* **P8: Qual *test smell com* mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** - Fechar a janela da *Treemap* - Fechar a janela *TSVizzEvolution - One Version* - Selecionar 2 versões - Carregar o arquivo "*resultado_evolution1.csv*" da pasta *commons-io-2.1* em "*Select the first .csv*"

	- Carregar o arquivo "*resultado_evolution1.csv"* da pasta *commons-io-2.6* em "*Select the second .csv*" - Selecionar *Methods* em "*Select the level of granularity"* - Carregar o arquivo fornecido "*commons-io_result_byclasstest_testsmells.csv*" da pasta *commons-io-2.1* em "*Select the first .csv File*" - Carregar o arquivo "*commons-io_result_byclasstest_testsmells.csv"* da pasta *commons-io-2.6 "Select the second .csv File" e clicar em Generate Timeline View* - Passar o mouse sobre a primeira classe: *AppendableOutputStreamTest* **P9: Qual o comportamento do *test smell Assertion Roulette* da classe *AppendableOutputStreamTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais método? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-io/blob/master/src/test/java/org/apache/commons/io/output/AppendableOutputStreamTest.java* e procurar o método *testWriteInt* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.** - Clicar em fechar ao na janela de título *Legend*

4.3 Tarefa 1 - `TSVizzEvolution - Commons Text`

O formulário para responder as questões está disponível em : *https://forms.gle/8RADB4c3kQARLCjk8*

`TSVizzEvolution`	- Selecionar 1 versão - Carregar o arquivo "*resultado_evolution1.csv*" da pasta *commons-text-1.9* - Escolher a técnica *Graph View* - Selecionar a granularidade *A Specific Test Class* - Selecionar a classe de teste *BiFunctionStringLookupTest* e clicar em *Generate Graph View* **P1: Quais são os *test smells* que essa classe possui?** - Fechar a tela do grafo gerado e selecionar outra classe de teste que desejar e clicar em *Generate Graph View* **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Fechar a janela do grafo e selecionar a granularidade *A Specific Test Smells* - Selecionar o *test smell Sensitive Equality* e clicar em *Generate Graph View* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** - Fechar a janela do grafo e selecionar a Granularidade *Author* - Selecionar o *Author: All*, que corresponde a todos autores - Selecionar o *test smell Sensitive Equality* **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** - Fechar a janela do grafo - Selecionar o *Author Gary Gregory* **P7: Quais são os *test smells* que esse autor é responsável?** - Selecionar a *View Type*: *Treemap View* e clicar em *Generate Treemap View* **P8: Qual *test smell com* mais ocorrências? Essa informação pode ser útil para o projeto? Explique.**

	- Fechar a janela da *Treemap* - Fechar a janela *TSVizzEvolution - One Version* - Selecionar 2 versões - Carregar o arquivo *"resultado_evolution1.csv"* da pasta *commons-text-1.0* em "*Select the first .csv*" - Carregar o arquivo "*resultado_evolution1.csv"* da pasta *commons-text-1.9* em "*Select the second .csv*" - Selecionar *Methods* em "*Select the level of granularity"* - Carregar o arquivo fornecido "*commons-text_result_byclasstest_testsmells.csv*" da pasta *commons-text-1.0* em "*Select the first .csv File*" - Carregar o arquivo "*commons-text_result_byclasstest_testsmells.csv"* da pasta *commons-text-1.9 "Select the second .csv File"* e clicar *em Generate Timeline View* - Passar o mouse sobre a primeira classe: *AggregateTranslatorTest* **P9: Qual o comportamento *do test smell Sensitive Equality* da classe *AggregateTranslatorTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais métodos? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-text/blob/master/src/test/java/org/apache/commons/text/translate/AggregateTranslatorTest.java* e procurar o método *testNonNull* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.** - Clicar em fechar ao lado do título *Legend*

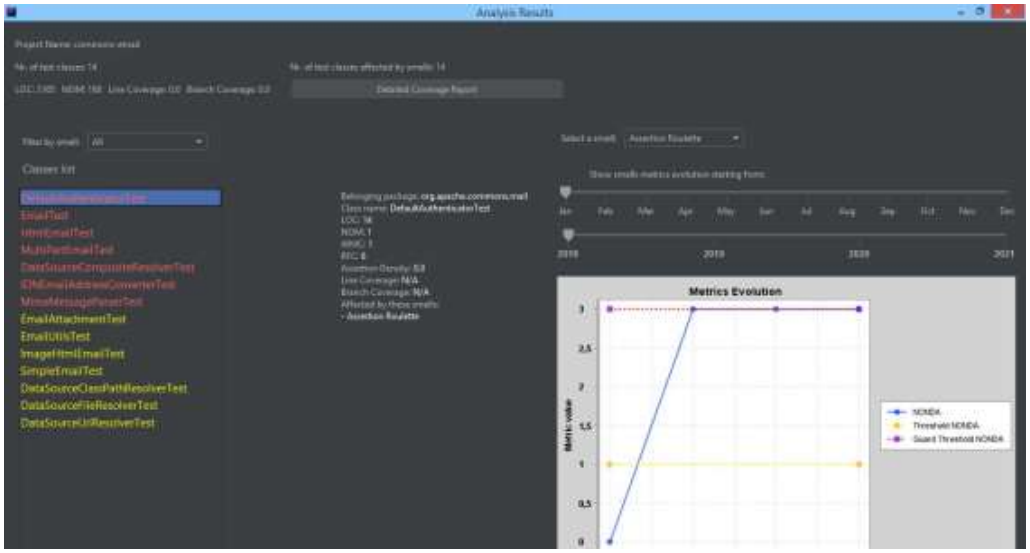
4.4 VITRuM

A ferramenta VITRuM (*VIsualization of Test-Related Metrics*) é um *plug-in* para o IntelliJ[46] para código Java que inclui o cálculo de medidas de código de teste e detecta 7 *test smells.* As medidas se dividem em dois tipos (i) estruturais referem ao tamanho, à coesão, ao acoplamento e à complexidade e (ii) dinâmicas exibem informações sobre a eficácia dos testes e cobertura de testes. Os *test smells* são exibidos através de uma lista com as classes de teste em cores diferentes (i) vermelho para classes de teste afetadas por *test smells*, com valores de medidas de código de teste acima dos limites definidos; (ii) amarelo para classes de teste afetadas por *test smells*, com valores de medidas de código de teste dentro dos limites definidos; e (iii) cinza para classes de teste sem *test smells*. Além disso, os usuários podem filtrar os resultados por *test smell* para exibir sua evolução em um gráfico durante o desenvolvimento do projeto (Figura 4.5). A seguir são detalhados as medidas e os *test smells* que a ferramenta abrange:

a) **Medidas estruturais:** LOC (*Line of Code*), AD (*Assertion Density*) e WMC (*Weigthed Methods Per Class*);
b) **Medidas dinâmicas:** *Line Coverage, Branch Coverage, Mutation Coverage e Flaky Tests.*
c) ***Test Smells:*** *Assertion Roulette, Eager Test, General Fixture, Indirect Testing, Sensitive Equality, Mystery Guest e Resource Optimism.*

[46] https://www.jetbrains.com/pt-br/idea/

Figura 4.5 - Painel Principal VITRuM



Fonte: Do autor (2021).

4.5 Treinamento 2 - VITRuM – Commons IO

VITRuM	- Acessar *File* - *Open* e carregar a pasta do projeto *Commons IO* - Acessar *Tools* - VITRuM - *Calculate Test Factors* - Deixar selecionado somente *Test Smells (default)* - Clicar em *Star Analysis* - Na janela aberta, procurar a classe *BrokenOutputStreamTest* **P1: Quais são os *test smells* que essa classe possui?** - Selecionar outra classe de teste que desejar **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Selecionar o *test smell Sensitive Equality* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** **P7: Quais são os *test smells* que esse autor é responsável?** **P8: Qual *test smell com* mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** **P9: Qual o comportamento do *test smell Assertion Roulette* da classe *AppendableOutputStreamTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais métodos? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-io/blob/master/src/test/java/org/apache/commons/io/output/AppendableOutputStreamTest.java* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.**

4.6 Tarefa 2 - VITRuM – Commons Email

VITRuM	- Acessar *File* - *Open* e carregar a pasta do projeto *Commons Email* - Acessar *Tools* - VITRuM - *Calculate Test Factors* - Deixar selecionado somente *Test Smells (default)* - Clicar em *Star Analysis* - Na janela aberta, procurar a classe *IDNEmailAddressConverterTest*

	P1: Quais são os *test smells* que essa classe possui? - Selecionar outra classe de teste que desejar **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Selecionar o *test smell Sensitive Equality* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** **P7: Quais são os *test smells* que esse autor é responsável?** **P8: Qual *test smell* com mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** **P9: Qual o comportamento do *test smell Assertion Roulette* da classe *EmailTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais métodos? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-email/blob/master/src/test/java/org/apache/commons/mail/DefaultAuthenticatorTest.java* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.**

4.7 Questionário

Disponível em*:* *https://forms.gle/1mis7iVRHrbBSvws5*

REFERÊNCIAS

CALEGARE, A. J. **Introdução ao delineamento de experimentos**. Editora Blucher, 2009.

PALOMBA, F., BAVOTA, G., DI PENTA, M., OLIVETO, R., DE LUCIA, A., POSHYVANYK, D. **Detecting bad smells in source code using change history information**. In: International Conference on Automated Software Engineering. p. 268-278. 2013.

PECORELLI, F.; LILLO, G.; PALOMBA, F.; LUCIA, A. **VITRuM-A Plug-In for the Visualization of Test-Related Metrics**. In: International Conference on Advanced Visual Interfaces. p.1-3, 2020.

PERUMA, A.; ALMALKI, K.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A.; PALOMBA, F. **On the distribution of test smells in open source Android applications: an exploratory study**. In: International Conference on Computer Science and Software Engineering. p. 193-202, 2019.

VAN DEURSEN, A.; MOONEN, L.; VAN DEN BERGH, A.; KOK, G. **Refactoring test code**. In: Conference on extreme programming and flexible processes in software engineering. p. 92-95, 2001

VIRGÍNIO, T.; SANTANA, R.; MARTINS, L. A.; SOARES, L. R.; COSTA, H.; MACHADO, I. **On the influence of Test Smells on Test Coverage**. In: Brazilian Symposium on Software Engineering. p. 467-471, 2019.

APÊNDICE G

Perguntas Experimento Controlado G3

*Obrigatório

TSVizzEvolution

P1: Quais são os test smells que essa classe possui? *

Marque todas que se aplicam.

- [ ] Assertion Roulette
- [ ] Conditional Test Logic
- [ ] Eager Test
- [ ] Lazy Test
- [ ] Sensitive Equality

Outro: [ ] _____

P2: Qual classe foi escolhida? Qual test smell com mais ocorrências e a quantidade? *

_____

_____

_____

_____

_____

P3: Quais são as classes que possuem o test smell Sensitive Equality? *

Marque todas que se aplicam.

- [ ] AggregateTranslatorTest
- [ ] AlphabetConverterTest
- [ ] BiFunctionStringLookupTest
- [ ] DateStringLookupTest
- [ ] DnsStringLookupTest
- [ ] EmailTest
- [ ] ExtendedMessageFormatTest
- [ ] FileStringLookupTest
- [ ] FormattableUtilsTest
- [ ] FunctionStringLookupTest
- [ ] InterpolatorStringLookupTest
- [ ] JavaPlatformStringLookupTest
- [ ] LevenshteinDetailedDistanceTest
- [ ] LocalHostStringLookupTest
- [ ] LookupTranslatorTest
- [ ] PropertiesStringLookupTest
- [ ] ResourceBundleStringLookupTest
- [ ] ScriptStringLookupTest
- [ ] StrBuilderTest
- [ ] StringTokenizerTest
- [ ] StrTokenizerTest
- [ ] StrSubstitutorTest
- [ ] TextStringBuilderTest
- [ ] UrlDecodeStringLookupTest
- [ ] UrlEncoderStringLookupTest
- [ ] UrlStringLookupTest
- [ ] XmlStringLookupTest

Outro: ☐ _____

P4: Qual autor provavelmente é responsável por mais test smells? Qual a quantidade de ocorrências? *

Marcar apenas uma oval.

- ◯ Stefan Bodewig
- ◯ Thomas Neidhart
- ◯ Rob Thompkins
- ◯ Gary Gregory
- ◯ Outro: _____

Informe aqui a quantidade de ocorrências: *

_____

P5: Você acha que a informação dos possíveis autores pelo test smell pode ajudar nas decisões de projeto e tomada de decisão? Explique. *

_____

_____

_____

_____

_____

P6: Quem são os possíveis autores pelo test smell Sensitive Equality? *

Marque todas que se aplicam.

- ☐ Stefan Bodewig
- ☐ Thomas Neidhart
- ☐ Gary Gregory
- ☐ Benson Mergulies
- ☐ Rob Thompkins

Outro: ☐ _____

P7: Quais são os test smells que esse autor é responsável? *

Marcar apenas uma oval.

- ◯ Nenhuma das alternativas
- ◯ Todas as alternativas
- ◯ Assertion Roulette
- ◯ Conditional Test Logic
- ◯ Duplicate Assert
- ◯ Eager Test
- ◯ Exception Catching Throwing
- ◯ General Fixture
- ◯ Ignored Test
- ◯ Lazy Test
- ◯ Magic Number Test
- ◯ Mystery Guest
- ◯ Print Statemente
- ◯ Redundant Assertion
- ◯ Resource Optimism
- ◯ Sensitive Equality
- ◯ Unknown Test
- ◯ Verbose Test

P8: Qual test smell com mais ocorrências? Essa informação pode ser útil para o projeto? Explique. *

P9: Qual o comportamento do test smell Assertion Roulette da classe AggregateTranslatorTest comparando V1 e V2 (evolução)? *

Marcar apenas uma oval.

- ◯ Aumentou
- ◯ Diminuiu
- ◯ Permaneceu
- ◯ Outros

P10: O test smell Assertion Roulette ocorre em quais métodos? E em quais linhas? *

Marque todas que se aplicam.

- ☐ testNullConstructor
- ☐ testNullVarargConstructor
- ☐ testWrite
- ☐ testNonNull
- ☐ testDefault

Outro: ☐ _____

Informe aqui em quais linhas ocorre: *

_____

P11: Você conseguiria corrigir os test smells com as informações fornecidas pela ferramenta? Explique. *

_____

_____

_____

_____

_____

VITRuM

P1: Quais são os test smells que essa classe possui? *

Marque todas que se aplicam.

- [ ] Assertion Roulette
- [ ] Conditional Test Logic
- [ ] Eager Test
- [ ] Lazy Test
- [ ] Sensitive Equality

Outro: [ ] _____

P2: Qual classe foi escolhida? Qual test smell com mais ocorrências e a quantidade? *

_____

_____

_____

_____

_____

P3: Quais são as classes que possuem o test smell Sensitive Equality? *

Marque todas que se aplicam.

- [ ] AggregateTranslatorTest
- [ ] AlphabetConverterTest
- [ ] BiFunctionStringLookupTest
- [ ] DateStringLookupTest
- [ ] EmailTest
- [ ] DnsStringLookupTest
- [ ] ExtendedMessageFormatTest
- [ ] FileStringLookupTest
- [ ] FormattableUtilsTest
- [ ] FunctionStringLookupTest
- [ ] InterpolatorStringLookupTest
- [ ] JavaPlatformStringLookupTest
- [ ] LevenshteinDetailedDistanceTest
- [ ] LocalHostStringLookupTest
- [ ] LookupTranslatorTest
- [ ] PropertiesStringLookupTest
- [ ] ResourceBundleStringLookupTest
- [ ] ScriptStringLookupTest
- [ ] StrBuilderTest
- [ ] StringEscapeUtilsTest
- [ ] StringMatcherFactoryTest
- [ ] StringSubstitutorTest
- [ ] StringTokenizerTest
- [ ] StrSubstitutorTest
- [ ] TextStringBuilderTest
- [ ] UrlDecodeStringLookupTest
- [ ] UrlEncoderStringLookupTest
- [ ] UrlStringLookupTest
- [ ] XmlStringLookupTest

Outro: ☐ _____

P4: Qual autor provavelmente é responsável por mais test smells? Qual a quantidade de ocorrências? *

Marcar apenas uma oval.

- ◯ Stefan Bodewig
- ◯ Thomas Neidhart
- ◯ A ferramenta não fornece essa informação
- ◯ Outro: _____

P5: Você acha que a informação dos possíveis autores pelo test smell pode ajudar nas decisões de projeto e tomada de decisão? Explique. *

_____

_____

_____

_____

_____

P6: Quem são os possíveis autores pelo test smell Sensitive Equality? *

Marque todas que se aplicam.

- ☐ Stefan Bodewig
- ☐ Thomas Neidhart
- ☐ Gary Gregory
- ☐ Benson Mergulies
- ☐ A ferramenta não fornece essa informação

Outro: ☐ _____

P7: Quais são os test smells que esse autor é responsável? *

Marcar apenas uma oval.

- Assertion Roulette
- Conditional Test Logic
- Duplicate Assert
- Eager Test
- Exception Catching Throwing
- General Fixture
- Ignored Test
- Lazy Test
- Magic Number Test
- Mystery Guest
- Redundant Assertion
- Resource Optimism
- Sensitive Equality
- Unknown Test
- Verbose Test
- Todas as alternativas
- A ferramenta não fornece essa informação

P8: Qual test smell com mais ocorrências? Essa informação pode ser útil para o projeto? Explique. *

P9: Qual o comportamento do test smell Assertion Roulette da classe DefaultAuthenticatorTest comparando V1 e V2 (evolução)? *

Marcar apenas uma oval.

- ◯ Aumentou
- ◯ Diminuiu
- ◯ Permaneceu
- ◯ Outros
- ◯ A ferramenta não fornece essa informação

P10: O test smell Assertion Roulette ocorre em quais métodos? E em quais linhas? *

Marque todas que se aplicam.

- ☐ testNullConstructor
- ☐ testNullVarargConstructor
- ☐ testWrite
- ☐ testNonNull
- ☐ testDefault
- ☐ A ferramenta não fornece essa informação

Outro: ☐ _____

P11: Você conseguiria corrigir os test smells com as informações fornecidas pela ferramenta? Explique. *

_____

_____

_____

_____

_____

APÊNDICE H

ADRIANA PRISCILA SANTOS CRUZ

Atividades do Estudo Experimental com Ferramentas para Visualização de *Test Smells* - Grupo 4

LAVRAS-MG
2021

Sumário

1 EXPERIMENTO CONTROLADO

As informações são lidas, com a finalidade de manter o padrão, pois trata-se de um estudo experimental e o treinamento com todos os participantes deve ser o mais semelhante. São explicados os conceitos de *test smells* e as ferramentas para a visualização de *test smells* utilizadas nesse estudo. Posteriormente, são realizadas as tarefas propostas nesse experimento.

2 *DESIGN* DO ESTUDO

O objetivo desse estudo é visualizar a ocorrência e a evolução de *test smells*. Sendo conduzido na seguinte ordem:

a) ***Test Smells*:** São explicados os conceitos e exemplificados dois *test smells*;
b) **Ferramentas para a Visualização de *Test Smells*:** São explicadas sobre duas ferramentas para a visualização de *test smells;*
c) **`VITRuM`** é a primeira ferramenta detalhada no estudo;
d) **Treinamento 1.** `VITRuM - Commons IO;`
e) **Tarefa 1.** `VITRuM – Commons Email;`
f) **`TSVizzEvolution`** é a segunda ferramenta detalhada no estudo;
g) **Treinamento 2.** `TSVizzEvolution - Commons IO;`
h) **Tarefa 2.** `TSVizzEvolution – Commons Text;`
i) **Aplicar questionário.** Questionário para caracterização dos participantes e *feedback* do experimento.

3 *TEST SMELLS*

Os *test smells* são considerados decisões ou práticas ruins de design no código de teste, os quais diminuem a qualidade dos testes e tornam os testes mais difíceis de serem compreendidos e mantidos durante a evolução dos sistemas de software. Dois *test smells* são explicados a seguir:

a) ***Assertion Roulette* (AR)**. Ocorre quando asserções em um método de teste não têm explicação, não é documentado e afeta a legibilidade, a compreensibilidade e a manutenção. Se uma das afirmações falhar, é difícil identificar qual falhou. O método *assertThat*() (linhas 7, 10 e 11) é "invocado" 3 vezes no método de teste. Em cada declaração, é verificada uma condição diferente, mas não é fornecida uma mensagem de explicação para cada declaração. Portanto, se uma das declarações falhar, a identificação da causa da falha não é direta.

Código 3.1 - *Exemplo de Test Smell Assertion Roulette*

```
1   @MediumTest
2   public void testCloneNonBareRepoFromLocalTestServer() throws
    Exception {
3       Clone cloneOp = new Clone(false,
    integrationGitServerURIFor("small-repo.early.git"),
    helper().newFolder());
4
5       Repository repo = executeAndWaitFor(cloneOp);
6
7       assertThat(repo,
    hasGitObject("ba1f63e4430bff267d112b1e8afc1d6294db0ccc"));
8
9       File readmeFile = new File(repo.getWorkTree(), "README");
10      assertThat(readmeFile, exists());
11      assertThat(readmeFile, ofLength(12));
12  }
```

Fonte: Adaptado de Peruma *et al.* (2019).

b) ***Sensitive Equality* (SE)**. Ocorre quando o método *toString*() é usado em métodos de teste, "invocando-o" para verificar objetos, para comparar a sua saída com a sequência específica. Alterações na implementação do método *toString*() podem resultar em falha. Na linha 15 do exemplo, o valor retornado pelo método *toString*() é comparado a *string* "/54.204.10.41", se houver alterações na implementação dos objetos do método *toString*(), pode ocorrer falha.

Código 3.2 - *Exemplo de Test Smell Sensitive Equality*

```
1.  @Test
2.  public void test1() throws UnknownHostException {
3.
4.      String peersPacket = "F8 4E 11 F8 4B C5 36 81 " +
5.          "CC 0A 29 82 76 5F B8 40 D8 D6 0C 25 80 FA 79 5C " +
6.          "FC 03 13 EF DE BA 86 9D 21 94 E7 9E 7C B2 B5 22 " +
7.          "F7 82 FF A0 39 2C BB AB 8D 1B AC 30 12 08 B1 37 " +
8.          "E0 DE 49 98 33 4F 3B CF 73 FA 11 7E F2 13 F8 74 " +
9.          "17 08 9F EA F8 4C 21 B0";
10.
11.       byte[] payload = Hex.decode(peersPacket);
12.
13.       byte[] ip = decodeIP4Bytes(payload, 5);
14.
15.       assertEquals(InetAddress.getByAddress(ip).toString(),
    ("/54.204.10.41"));
16.   }
```

Fonte: Adaptado de Peruma *et al.* (2019).

4 FERRAMENTAS PARA VISUALIZAÇÃO DE *TEST SMELLS*

Neste experimento, as ferramentas `TsVizzEvolution`[1] e `VITRuM`[2] são submetidas a um estudo comparativo. Esse estudo visa analisar fatores correspondentes a eficácia, e usabilidade que podem auxiliar *testers*, gerente de testes e demais envolvidos nas atividades de teste de software a visualizar mais agilmente os *test smells*, para corrigi-los e melhorar a qualidade do código de teste.

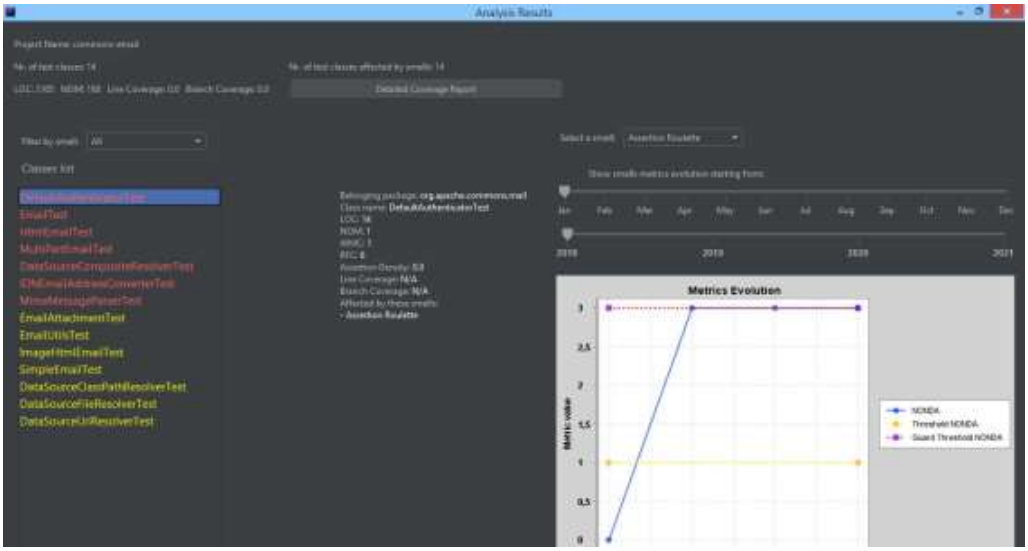
4.1. `VITRuM`

A ferramenta `VITRuM` (*VIsualization of Test-Related Metrics*) é um *plugin* para o IntelliJ[3] para código Java que inclui o cálculo de medidas de código de teste e detecta 7 *test smells.* As medidas se dividem em dois tipos (i) estruturais referem ao tamanho, à coesão, ao acoplamento e à complexidade e (ii) dinâmicas exibem informações sobre a eficácia dos testes e cobertura de testes. Os *test smells* são exibidos através de uma lista com as classes de teste em cores diferentes (i) vermelho para classes de teste afetadas por *test smells*, com valores de medidas de código de teste acima dos limites definidos; (ii) amarelo para classes de teste afetadas por *test smells*, com valores de medidas de código de teste dentro dos limites definidos; e (iii) cinza para classes de teste sem *test smells*. Além disso, os usuários podem filtrar os resultados por *test smell* para exibir sua evolução em um gráfico durante o desenvolvimento do projeto (Figura 4.5). A seguir são detalhados as medidas e os *test smells* que a ferramenta abrange:

a) **Medidas estruturais:** LOC (*Line of Code*), AD (*Assertion Density*) e WMC (*Weigthed Methods Per Class*);
b) **Medidas dinâmicas:** *Line Coverage, Branch Coverage, Mutation Coverage e Flaky Tests.*
c) ***Test Smells:*** *Assertion Roulette, Eager Test, General Fixture, Indirect Testing, Sensitive Equality, Mystery Guest e Resource Optimism.*

[1] https://github.com/arieslab/TSVizzEvolution
[2] https://plugins.jetbrains.com/plugin/14160-vitrum
[3] https://www.jetbrains.com/pt-br/idea/

Figura 4.1 - Painel Principal VITRuM



Fonte: Do autor (2021).

4.2. Treinamento 1 - VITRuM – Commons IO

VITRuM	- Acessar *File* - *Open* e carregar a pasta do projeto *Commons IO* - Acessar *Tools* - VITRuM - *Calculate Test Factors* - Deixar selecionado somente *Test Smells (default)* - Clicar em *Star Analysis* - Na janela aberta, procurar a classe *BrokenOutputStreamTest* **P1: Quais são os *test smells* que essa classe possui?** - Selecionar outra classe de teste que desejar **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Selecionar o *test smell Sensitive Equality* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** **P7: Quais são os *test smells* que esse autor é responsável?** **P8: Qual *test smell com* mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** **P9: Qual o comportamento do *test smell Assertion Roulette* da classe *AppendableOutputStreamTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais métodos? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-io/blob/master/src/test/java/org/apache/commons/io/output/AppendableOutputStreamTest.java* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.**

4.3. Tarefa 1 - VITRuM – Commons Email

O formulário para responder as questões está disponível em : *https://forms.gle/HS4eBD82o6Z9RYL29*

VITRuM	- Acessar *File* - *Open* e carregar a pasta do projeto *Commons Email* - Acessar *Tools* - VITRuM - *Calculate Test Factors* - Deixar selecionado somente *Test Smells (default)* - Clicar em *Star Analysis* - Na janela aberta, procurar a classe *IDNEmailAddressConverterTest* **P1: Quais são os *test smells* que essa classe possui?** - Selecionar outra classe de teste que desejar **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Selecionar o *test smell Sensitive Equality* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** **P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências?** **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** **P7: Quais são os *test smells* que esse autor é responsável?** **P8: Qual *test smell* com mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** **P9: Qual o comportamento do *test smell Assertion Roulette* da classe *EmailTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais métodos? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-email/blob/master/src/test/java/org/apache/commons/mail/DefaultAuthenticatorTest.java* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.**

4.4. TSVizzEvolution

A ferramenta TSVizzEvolution utiliza três técnicas de visualização para exibir a ocorrência e evolução de 21 *test smells* (Palomba *et al.*, 2013; Peruma *et al.*, 2019; Deursen *et al.*, 2001). Esses *test smells* são detectados pela ferramenta JNose Test[1]: *Assertion Roulette, Conditional Test Logic, Constructor Initialization, Default Test, Dependent Test, Duplicate Assert, Eager Test, Empty Test, Exception Catching Throwing, General Fixture, Ignored Test, Lazy Test, Magic Number Test, Mystery Guest, Print Statement, Redundant Assertion, Resource Optimism, Sensitive Equality, Sleepy Test, Unknown Test, Verbose Test* e armazenados em um arquivo .csv. A ferramenta TSVizzEvolution estrutura os dados do arquivo .csv conforme visualização escolhida, a análise de *test smells* pode ser realizada para uma versão do código de teste (Análise Única) ou para duas versões (Análise de Evolução).

[1] http://jnose.herokuapp.com/

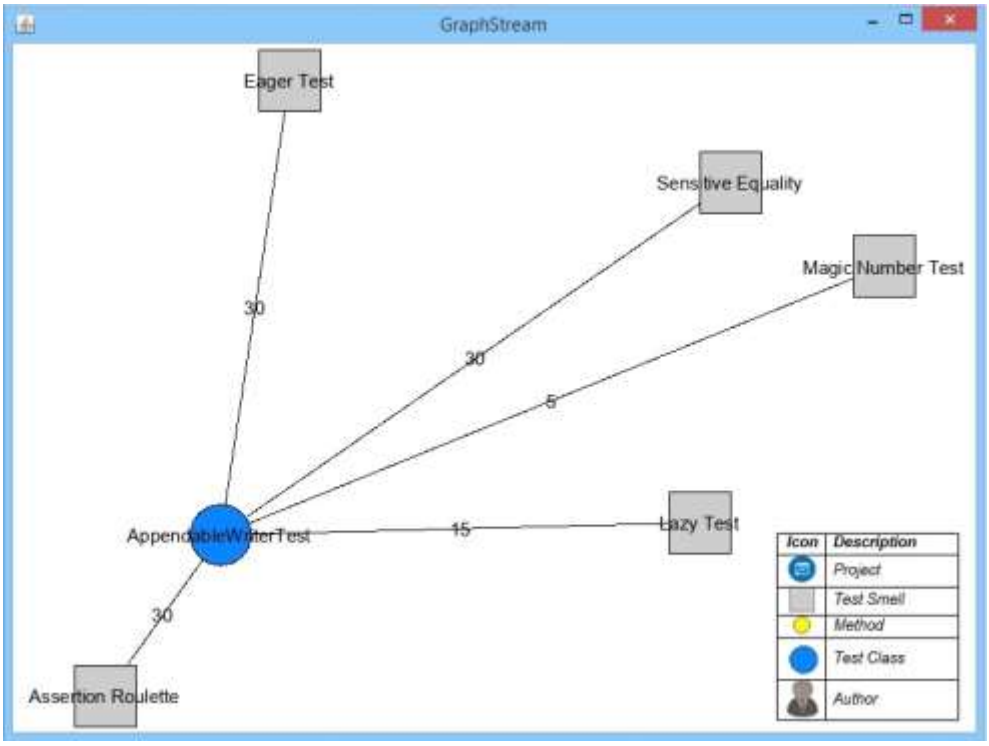
4.4.1 Análise Única

Para a análise das ocorrências de *test smells* para uma versão do código de teste estão disponíveis duas técnicas: *Graph View* e *Treemap View*, essas técnicas são explicadas a seguir:

a) ***Graph View***. Os dados retornados pela `JNose` são estruturados em grafos. Os componentes do grafo variam de acordo com a granularidade escolhida, tem-se seis granularidades disponíveis (Quadro 4.1). Na Figura 4.1 verifica-se um grafo e na Figura 4.2 o detalhamento da legenda, que exibe o significado dos ícones utilizados como nós no grafo. Os componentes podem ser arrastados para ajustar a visualização conforme necessidade.

b) ***Treemap View***. Os *test smells* são representados por retângulos, onde cada retângulo possui uma cor, gerada aleatoriamente. Além disso, os tamanhos dos retângulos representam as suas quantidades de ocorrências, ou seja, quanto maior o retângulo mais quantidade de ocorrências de *test smells* (Figura 4.3).

Figura 4.2 - *Graph View*



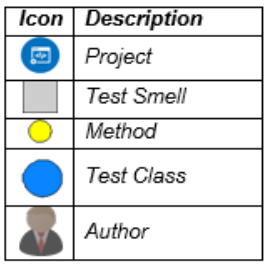
Fonte: Do autor (2021).

Quadro 4.1 - Granularidade da Ferramenta

Granularidade	**Descrição**
Project	Exibe a relação entre todo o projeto e os *test smells* (1:N).
All Test Classes	Exibe a relação entre as classes de teste e os *test smells* (N:N).
A Specific Test Class	Exibe a relação entre as classes de teste e *test smells* (1:N). Para executar a ferramenta com essa granularidade, os usuários precisam selecionar uma classe de teste.
A Specific Test Smells	Exibe a relação entre as classes de teste e os *test smells* (N:1). Para executar a ferramenta com essa granularidade, os usuários devem selecionar um *test smell*.
Author	Exibe a relação entre os autores, os *test smells* (N:N) e as classes de teste (1: N). Para utilizar a ferramenta com essa granularidade, os usuários devem selecionar o autor (um específico ou todos autores) e um *test smell*.
Methods	Exibe a relação entre o *test smell*, a classe de teste e os métodos (1:N). Os usuários precisam selecionar um *test smell* e uma classe de teste.

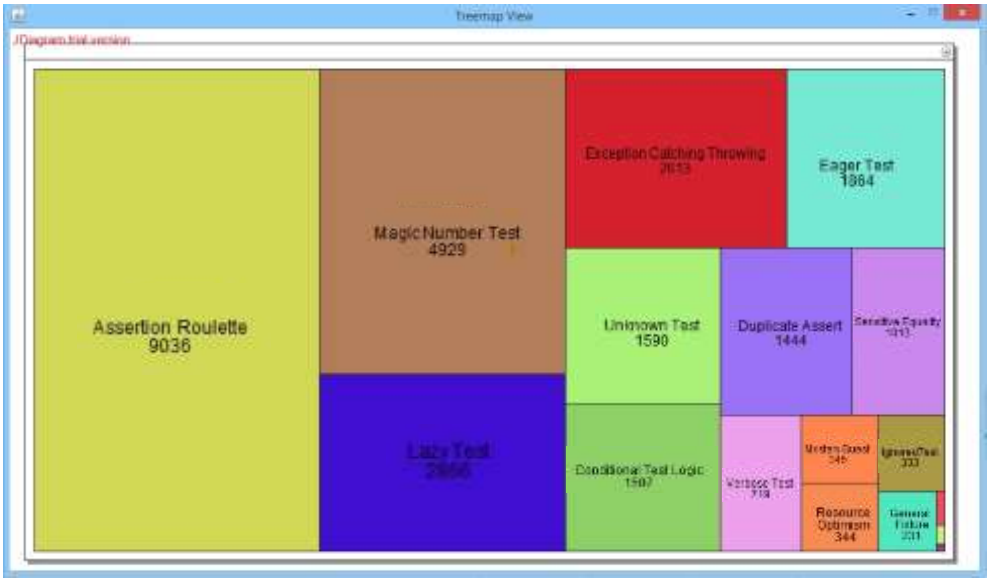
Fonte: Do autor (2021).

Figura 4.3 - Legenda *Graph View*



Fonte: Do autor (2021).

Figura 4.4 - *Treemap View*



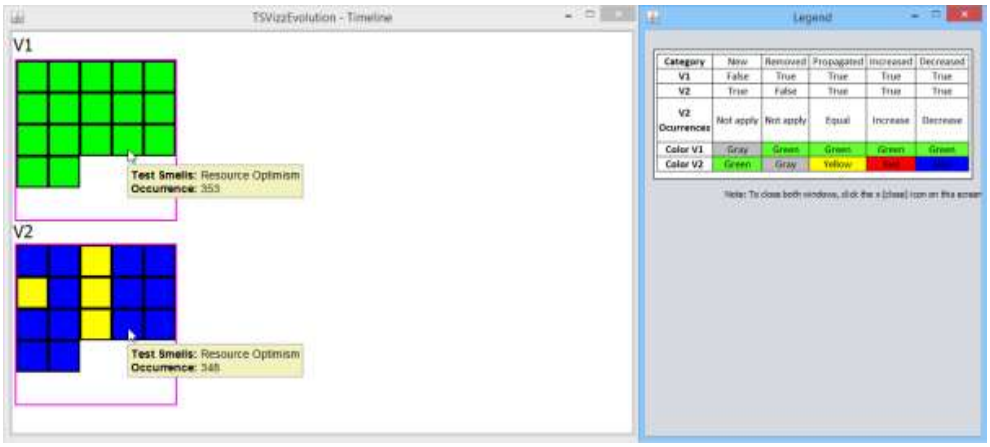
Fonte: Do autor (2021).

4.4.2 Análise de Evolução

Para a análise das ocorrências de *test smells* para duas versões é utilizada a técnica *Timeline View,* essa técnica é explicada a seguir.

a) ***Timeline View***. Os usuários podem escolher entre três granularidades: i) *Project*; ii) *All Test Classes;* ou *iii) Methods*. Os artefatos (projeto, classe de teste e *test smells*) do código de teste são representados por retângulos e as versões são apresentadas em paralelo horizontalmente, conforme ordem de inserção dos arquivos `.csv`. Os retângulos são aninhados da seguinte forma: i) **retângulo externo** representa o projeto ou a classe de teste (conforme granularidade selecionada) possui borda na **cor rosa**; e ii) **retângulos menores** representam cada *test smell* detectado em uma classe de teste ou projeto, com as cores conforme comportamento durante a evolução, podendo ser verde, azul, cinza ou vermelho.

Figura 4.5 - *Timeline View*



Fonte: Do autor (2021).

As posições dos retângulos nas duas versões são mantidas para facilitar a rastreabilidade entre classes de teste e *test smells*. As cores dos *test smells* são categorizados conforme sua existência e quantidade de ocorrências (Quadro 4.2) e apresentadas ao lado da *timeline* (Figura 4.4). Os usuários podem ver os nomes das classes de teste, projeto e *test smells* e suas quantidade de ocorrências ao posicionar o *mouse* sobre cada retângulo. As categorias são (Versão um - V_1 e Versão dois - V2):

- **Novo**. Se V_1 não tiver o *test smell* específico (*False*) e V_2 (*True*), a categoria é "**New**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**cinza**" e "**verde**", respectivamente;

- **Removido**. Se V_1 tiver o *test smell* específico (*True*) e V_2 (*False*), a categoria é "**Removido**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**verde**" e "**cinza**", respectivamente;
- **Propagado**. Se V_1 e V_2 tiverem o *test smell* específico (*True* - *True*) e a quantidade for a mesma (*Igual*), a categoria é "**Propagado**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem as cores "**verde**" e "**amarelo**", respectivamente;
- **Aumentado**. Se V_1 e V_2 tiverem um *test smell* específico (*True* - *True*), mas a quantidade de *test smells* aumenta em V_2 (**Aumenta**), a categoria é "**Aumentado**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem a cores "**verde**" e "**vermelho**", respectivamente;
- **Diminuído**. Se V_1 e V_2 tiverem um *test smell* específico (*True* - *True*), mas a quantidade de *test smells* diminui na V_2 (**Diminui**), a categoria é "**Diminuído**". Assim, o retângulo atribuído a esse *test smell* em V_1 e V_2 tem a cores "**verde**" e "**azul**".

Quadro 4.2 - Classificação dos *Test Smells*

Categoria	**V_1**	**V_2**	**V_2 Ocorrências**	**Cor V_1**	**Cor V_2**
Novo	*False*	*True*	Não se aplica	Cinza	Verde
Removido	*True*	*False*	Não se aplica	Verde	Cinza
Propagado	*True*	*True*	Igual	Verde	Amarelo
Aumentado	*True*	*True*	Aumentado	Verde	Vermelho
Diminuído	*True*	*True*	Diminuído	Verde	Azul

Fonte: Do autor (2021).

4.5. Treinamento 2 - `TSVizzEvolution - Commons IO`

`TSVizzEvolution`	- Selecionar 1 versão - Carregar o arquivo "*resultado_evolution1.csv*" da pasta *commons-io-2.6* - Escolher a técnica *Graph View* - Selecionar a granularidade *A Specific Test Class* - Selecionar a classe de teste *BrokenOutputStreamTest* e clicar em *Generate Graph View* **P1: Quais são os *test smells* que essa classe possui?** - Fechar a tela do grafo gerado e selecionar outra classe de teste que desejar e clicar em *Generate Graph View* **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Selecionar a granularidade *A Specific Test Smells* - Selecionar o *test smell Sensitive Equality* e clicar em *Generate Graph View* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** - Fechar a janela do grafo e selecionar a Granularidade *Author* - Selecionar o *Author: All*, que corresponde a todos autores - Selecionar o *test smell Sensitive Equality*

	P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências? **P5: Você acha que a informação do responsável pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** - Fechar a janela do grafo - Selecionar o *Author* Benson Margulies **P7: Quais são os *test smells* que esse autor é responsável?** - Selecionar a *View Type: Treemap View* e clicar em *Generate Treemap View* **P8: Qual *test smell com* mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** - Fechar a janela da *Treemap* - Fechar a janela *TSVizzEvolution - One Version* - Selecionar 2 versões - Carregar o arquivo "*resultado_evolution1.csv*" da pasta *commons-io-2.1* em "*Select the first .csv*" - Carregar o arquivo "*resultado_evolution1.csv"* da pasta *commons-io-2.6* em "*Select the second .csv*" - Selecionar *Methods* em "*Select the level of granularity"* - Carregar o arquivo fornecido "*commons-io_result_byclasstest_testsmells.csv*" da pasta *commons-io-2.1* em "*Select the first .csv File*" - Carregar o arquivo "*commons-io_result_byclasstest_testsmells.csv"* da pasta *commons-io-2.6 "Select the second .csv File" e clicar em Generate Timeline View* - Passar o mouse sobre a primeira classe: *AppendableOutputStreamTest* **P9: Qual o comportamento do *test smell Assertion Roulette* da classe *AppendableOutputStreamTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais método? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-io/blob/master/src/test/java/org/apache/commons/io/output/AppendableOutputStreamTest.java* e procurar o método *testWriteInt* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.** - Clicar em fechar ao na janela de título *Legend*

4.6. Tarefa 2 - `TSVizzEvolution - Commons Text`

`TSVizzEvolution`	- Selecionar 1 versão - Carregar o arquivo "*resultado_evolution1.csv*" da pasta *commons-text-1.9* - Escolher a técnica *Graph View* - Selecionar a granularidade *A Specific Test Class* - Selecionar a classe de teste *BiFunctionStringLookupTest* e clicar em *Generate Graph View* **P1: Quais são os *test smells* que essa classe possui?** - Fechar a tela do grafo gerado e selecionar outra classe de teste que desejar e clicar em *Generate Graph View* **P2: Qual classe foi escolhida? Qual *test smell* com mais ocorrências e a quantidade?** - Fechar a janela do grafo e selecionar a granularidade *A Specific Test Smells* - Selecionar o *test smell Sensitive Equality* e clicar em *Generate Graph View* **P3: Quais são as classes que possuem o *test smell Sensitive Equality*?** - Fechar a janela do grafo e selecionar a Granularidade *Author* - Selecionar o *Author: All*, que corresponde a todos autores - Selecionar o *test smell Sensitive Equality*

	P4: Qual autor provavelmente é responsável por mais *test smells?* Qual a quantidade de ocorrências? **P5: Você acha que a informação dos possíveis autores pelo *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique.** **P6: Quem são os possíveis autores pelo *test smell Sensitive Equality*?** - Fechar a janela do grafo - Selecionar o *Author Gary Gregory* **P7: Quais são os *test smells* que esse autor é responsável?** - Selecionar a *View Type*: *Treemap View* e clicar em *Generate Treemap View* **P8: Qual *test smell com* mais ocorrências? Essa informação pode ser útil para o projeto? Explique.** - Fechar a janela da *Treemap* - Fechar a janela *TSVizzEvolution - One Version* - Selecionar 2 versões - Carregar o arquivo *"resultado_evolution1.csv"* da pasta *commons-text-1.0* em "*Select the first .csv*" - Carregar o arquivo "*resultado_evolution1.csv"* da pasta *commons-text-1.9* em "*Select the second .csv*" - Selecionar *Methods* em "*Select the level of granularity"* - Carregar o arquivo fornecido "*commons-text_result_byclasstest_testsmells.csv*" da pasta *commons-text-1.0* em "*Select the first .csv File*" - Carregar o arquivo "*commons-text_result_byclasstest_testsmells.csv"* da pasta *commons-text-1.9 "Select the second .csv File"* e clicar *em Generate Timeline View* - Passar o mouse sobre a primeira classe: *AggregateTranslatorTest* **P9: Qual o comportamento *do test smell Sensitive Equality* da classe *AggregateTranslatorTest* comparando V1 e V2 (evolução)?** **P10: O *test smell Assertion Roulette* ocorre em quais métodos? E em quais linhas?** - Abrir a classe no *GitHub* : *https://github.com/apache/commons-text/blob/master/src/test/java/org/apache/commons/text/translate/AggregateTranslatorTest.java* e procurar o método *testNonNull* **P11: Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique.** - Clicar em fechar ao lado do título *Legend*

4.7. Questionário

Disponível em*: https://forms.gle/1mis7iVRHrbBSvws5*

REFERÊNCIAS

CALEGARE, A. J. **Introdução ao delineamento de experimentos**. Editora Blucher, 2009.

PALOMBA, F., BAVOTA, G., DI PENTA, M., OLIVETO, R., DE LUCIA, A., POSHYVANYK, D. **Detecting bad smells in source code using change history information**. In: International Conference on Automated Software Engineering. p. 268-278. 2013.

PECORELLI, F.; LILLO, G.; PALOMBA, F.; LUCIA, A. **VITRuM-A Plug-In for the Visualization of Test-Related Metrics**. In: International Conference on Advanced Visual Interfaces. p.1-3, 2020.

PERUMA, A.; ALMALKI, K.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A.; PALOMBA, F. **On the distribution of test smells in open source Android applications: an exploratory study**. In: International Conference on Computer Science and Software Engineering. p. 193-202, 2019.

PERUMA, A. **Tsdetect: An open source test smells detection tool**. In: Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 1650-1654, 2020.

VAN DEURSEN, A.; MOONEN, L.; VAN DEN BERGH, A.; KOK, G. **Refactoring test code**. In: Conference on extreme programming and flexible processes in software engineering. p. 92-95, 2001

VIRGÍNIO, T.; SANTANA, R.; MARTINS, L. A.; SOARES, L. R.; COSTA, H.; MACHADO, I. **On the influence of Test Smells on Test Coverage**. In: Brazilian Symposium on Software Engineering. p. 467-471, 2019.

APÊNDICE I

Perguntas Experimento Controlado G4

*Obrigatório

VITRuM

P1: Quais são os test smells que essa classe possui? *

Marque todas que se aplicam.

- [ ] Assertion Roulette
- [ ] Conditional Test Logic
- [ ] Eager Test
- [ ] Lazy Test
- [ ] Sensitive Equality

Outro: [ ] _____

P2: Qual classe foi escolhida? Qual test smell com mais ocorrências e a quantidade? *

_____

_____

_____

_____

_____

P3: Quais são as classes que possuem o test smell Sensitive Equality? *

Marque todas que se aplicam.

- [ ] AggregateTranslatorTest
- [ ] AlphabetConverterTest
- [ ] BiFunctionStringLookupTest
- [ ] DateStringLookupTest
- [ ] DnsStringLookupTest
- [ ] EmailTest
- [ ] ExtendedMessageFormatTest
- [ ] FileStringLookupTest
- [ ] FormattableUtilsTest
- [ ] FunctionStringLookupTest
- [ ] InterpolatorStringLookupTest
- [ ] JavaPlatformStringLookupTest
- [ ] LevenshteinDetailedDistanceTest
- [ ] LocalHostStringLookupTest
- [ ] LookupTranslatorTest
- [ ] PropertiesStringLookupTest
- [ ] ResourceBundleStringLookupTest
- [ ] ScriptStringLookupTest
- [ ] StrBuilderTest
- [ ] StringEscapeUtilsTest
- [ ] StringMatcherFactoryTest
- [ ] StringSubstitutorTest
- [ ] StringTokenizerTest
- [ ] StrSubstitutorTest
- [ ] TextStringBuilderTest
- [ ] UrlDecodeStringLookupTest
- [ ] UrlEncoderStringLookupTest
- [ ] UrlStringLookupTest
- [ ] XmlStringLookupTest

Outro: ☐ _____

P4: Qual autor provavelmente é responsável por mais test smells? Qual a quantidade de ocorrências? *

Marcar apenas uma oval.

- ◯ Stefan Bodewig
- ◯ Thomas Neidhart
- ◯ A ferramenta não fornece essa informação
- ◯ Outro: _____

P5: Você acha que a informação dos possíveis autores pelo test smell pode ajudar nas decisões de projeto e tomada de decisão? Explique. *

_____

_____

_____

_____

_____

P6: Quem são os possíveis autores pelo test smell Sensitive Equality? *

Marque todas que se aplicam.

- ☐ Stefan Bodewig
- ☐ Thomas Neidhart
- ☐ Gary Gregory
- ☐ Benson Mergulies
- ☐ A ferramenta não fornece essa informação
- Outro: ☐ _____

P7: Quais são os test smells que esse autor é responsável? *

Marcar apenas uma oval.

- ◯ Assertion Roulette
- ◯ Conditional Test Logic
- ◯ Duplicate Assert
- ◯ Eager Test
- ◯ Exception Catching Throwing
- ◯ General Fixture
- ◯ Ignored Test
- ◯ Lazy Test
- ◯ Magic Number Test
- ◯ Mystery Guest
- ◯ Redundant Assertion
- ◯ Resource Optimism
- ◯ Sensitive Equality
- ◯ Unknown Test
- ◯ Verbose Test
- ◯ Todas as alternativas
- ◯ A ferramenta não fornece essa informação

P8: Qual test smell com mais ocorrências? Essa informação pode ser útil para o projeto? Explique. *

P9: Qual o comportamento do test smell Assertion Roulette da classe AggregateTranslatorTest comparando V1 e V2 (evolução)? *

Marcar apenas uma oval.

- ◯ Aumentou
- ◯ Diminuiu
- ◯ Permaneceu
- ◯ Outros
- ◯ A ferramenta não fornece essa informação

P10: O test smell Assertion Roulette ocorre em quais métodos? E em quais linhas? *

Marque todas que se aplicam.

- ☐ testNullConstructor
- ☐ testNullVarargConstructor
- ☐ testWrite
- ☐ testNonNull
- ☐ testDefault
- ☐ A ferramenta não fornece essa opção

Outro: ☐ _____

P11: Você conseguiria corrigir os test smells com as informações fornecidas pela ferramenta? Explique. *

_____

_____

_____

_____

_____

TSVizzEvolution

P1: Quais são os test smells que essa classe possui? *

Marque todas que se aplicam.

- [ ] Assertion Roulette
- [ ] Conditional Test Logic
- [ ] Eager Test
- [ ] Lazy Test
- [ ] Sensitive Equality

Outro: [ ] _____

P2: Qual classe foi escolhida? Qual test smell com mais ocorrências e a quantidade? *

_____

_____

_____

_____

_____

P3: Quais são as classes que possuem o test smell Sensitive Equality? *

Marque todas que se aplicam.

- [ ] AggregateTranslatorTest
- [ ] AlphabetConverterTest
- [ ] BiFunctionStringLookupTest
- [ ] DateStringLookupTest
- [ ] DnsStringLookupTest
- [ ] EmailTest
- [ ] ExtendedMessageFormatTest
- [ ] FileStringLookupTest
- [ ] FormattableUtilsTest
- [ ] FunctionStringLookupTest
- [ ] InterpolatorStringLookupTest
- [ ] JavaPlatformStringLookupTest
- [ ] LevenshteinDetailedDistanceTest
- [ ] LocalHostStringLookupTest
- [ ] LookupTranslatorTest
- [ ] PropertiesStringLookupTest
- [ ] ResourceBundleStringLookupTest
- [ ] ScriptStringLookupTest
- [ ] StrBuilderTest
- [ ] StringTokenizerTest
- [ ] StrSubstitutorTest
- [ ] TextStringBuilderTest
- [ ] UrlDecodeStringLookupTest
- [ ] UrlEncoderStringLookupTest
- [ ] UrlStringLookupTest
- [ ] XmlStringLookupTest

Outro: [ ] _____

P4: Qual autor provavelmente é responsável por mais test smells? Qual a quantidade de ocorrências? *

Marcar apenas uma oval.

- ◯ Stefan Bodewig
- ◯ Thomas Neidhart
- ◯ Rob Thompkins
- ◯ Gary Gregory
- ◯ Outro: _____

Informe aqui a quantidade de ocorrências: *

_____

P5: Você acha que a informação do responsável pelo test smell pode ajudar nas decisões de projeto e tomada de decisão? Explique. *

_____

_____

_____

_____

_____

P6: Quem são os possíveis autores pelo test smell Sensitive Equality? *

Marque todas que se aplicam.

- ☐ Stefan Bodewig
- ☐ Thomas Neidhart
- ☐ Gary Gregory
- ☐ Benson Mergulies
- ☐ Rob Thompkins

Outro: ☐ _____

P7: Quais são os test smells que esse autor é responsável? *

Marcar apenas uma oval.

- ◯ Nenhuma das alternativas
- ◯ Todas as alternativas
- ◯ Assertion Roulette
- ◯ Conditional Test Logic
- ◯ Duplicate Assert
- ◯ Eager Test
- ◯ Exception Catching Throwing
- ◯ General Fixture
- ◯ Ignored Test
- ◯ Lazy Test
- ◯ Magic Number Test
- ◯ Print Statement
- ◯ Redundant Assertion
- ◯ Resource Optimism
- ◯ Sensitive Equality
- ◯ Unknown Test
- ◯ Verbose Test

P8: Qual test smell com mais ocorrências? Essa informação pode ser útil para o projeto? Explique. *

P9: Qual o comportamento do test smell Assertion Roulette da classe AggregateTranslatorTest comparando V1 e V2 (evolução)? *

Marcar apenas uma oval.

- ◯ Aumentou
- ◯ Diminuiu
- ◯ Permaneceu
- ◯ Outros

P10: O test smell Assertion Roulette ocorre em quais métodos? E em quais linhas? *

Marque todas que se aplicam.

- ☐ testNullConstructor
- ☐ testNullVarargConstructor
- ☐ testWrite
- ☐ testNonNull
- ☐ testDefault

Outro: ☐ _____

Informe aqui em quais linhas ocorre: *

_____

P11: Você conseguiria corrigir os test smells com as informações fornecidas pela ferramenta? Explique. *

_____

_____

_____

_____

_____

APÊNDICE J

Questionário

Caro(a) participante,

Este questionário é parte de um experimento sobre testes automatizados, especificamente Test Smells. Sendo parte da pesquisa acadêmica do mestrado em Ciência da Computação e linha de pesquisa de Engenharia de Software da Universidade Federal de Lavras (UFLA). O objetivo da pesquisa é analisar diferentes ferramentas para visualização de test smells. A sua participação neste estudo é fundamental.

*Obrigatório

Termo de consentimento livre e esclarecido

1. Você está sendo convidado para participar da pesquisa "UMA ABORDAGEM VISUAL PARA EVOLUÇÃO DE TEST SMELLS EM SOFTWARE JAVA".
2. Você foi selecionado para ser voluntário e sua participação não é obrigatória.
3. A qualquer momento você pode desistir de participar e retirar seu consentimento.
4. Sua recusa não trará nenhum prejuízo em sua relação com o pesquisador ou com a instituição.
5. Sua participação nesta pesquisa consistirá em responder um questionário com questões objetivas e de texto curto.
6. A sua participação nesta pesquisa pode envolver algum desconforto, ainda que pequeno, relacionado ao tempo despendido para o preenchimento do questionário. Faremos o possível para minimizar possíveis desconfortos e não ocupar seu tempo desnecessariamente.
7. As informações obtidas serão confidenciais e asseguramos o sigilo sobre sua participação. Os dados publicados não permitirão a sua identificação.
8. As informações obtidas serão utilizadas única e exclusivamente para os fins específicos deste estudo.
9. Os benefícios relacionados à sua participação estão em contribuir com a pesquisa científica. Será permitido o acesso aos resultados desta pesquisa por meio de publicações científicas realizadas a partir desse estudo.
10. Este questionário utiliza o pacote de aplicativo Google Docs, portanto a coleta e o uso de informações do Google estão sujeitos à Política de privacidade do Google (https://www.google.co.uk/policies/privacy/).
11. Ao clicar no botão "Concordo", você concorda com as informações aqui descritas, porém a qualquer momento você pode interromper a pesquisa sem ônus algum.
12. Abaixo seguem os dados de contato dos responsáveis por esta pesquisa, com os quais você pode tirar suas dúvidas sobre sua participação.

Adriana Priscila Santos Cruz (Aluna de Mestrado) <adriana.cruz@estudante.ufla.br>
Professor Dr. Heitor Augustus Xavier Costa (Supervisor) <heitor@ufla.br>

Perdões, Minas Gerais, 01 de junho de 2021

1. *

Marcar apenas uma oval.

- ◯ Concordo
- ◯ Discordo

Pular para a seção 7 (Agradecemos a sua colaboração, mas infelizmente você não poderá participar porque não aceitou os termos.)

Caracterização dos participantes

2. Qual seu estado? *

Marcar apenas uma oval.

- ◯ AC
- ◯ AL
- ◯ AP
- ◯ AM
- ◯ BA
- ◯ CE
- ◯ DF
- ◯ ES
- ◯ GO
- ◯ MA
- ◯ MT
- ◯ MS
- ◯ MG
- ◯ PA
- ◯ PB
- ◯ PR
- ◯ PE
- ◯ PI
- ◯ RJ
- ◯ RN
- ◯ RS
- ◯ RO
- ◯ RR
- ◯ SC
- ◯ SP
- ◯ SE
- ◯ TO

3. Qual a sua escolaridade? *

Marcar apenas uma oval.

- ◯ Estudante de Graduação
- ◯ Graduação Completa
- ◯ Estudante de Mestrado
- ◯ Mestrado Completo
- ◯ Estudante de Doutorado
- ◯ Doutorado Completo
- ◯ Outro: _____

4. Qual sua área de atuação? *

Marcar apenas uma oval.

- ◯ Desenvolvimento de Software
- ◯ Teste de Software
- ◯ Pesquisador em Engenharia de Software
- ◯ Outro: _____

5. Onde você atua? *

Marcar apenas uma oval.

- ◯ Academia *Pular para a pergunta 7*
- ◯ Indústria *Pular para a pergunta 6*
- ◯ Ambos

Pular para a pergunta 7

6. Quantos anos de experiência profissional você tem na área de testes de software?

Marcar apenas uma oval.

- ◯ < 1 ano
- ◯ >= 1 ano e < 5 anos
- ◯ >= 5 anos e < 10 anos
- ◯ >= 10 anos
- ◯ Não possuo

Pular para a pergunta 7

7. Quantos anos de experiência acadêmica você tem na área de testes de software?

Marcar apenas uma oval.

- ◯ < 1 ano
- ◯ >= 1 ano e < 5 anos
- ◯ >= 5 anos e < 10 anos
- ◯ >= 10 anos
- ◯ Não possuo

8. Em relação ao seu conhecimento sobre Test Smells. Responda. *

Marcar apenas uma oval.

- ◯ Sei o que são Test Smells
- ◯ Trabalho com tópicos relacionados a Test Smells
- ◯ Sou pesquisador de tópicos relacionados a Test Smells
- ◯ Sei o que é Test Smells, mas nunca trabalhei com Test Smells
- ◯ Eu nunca ouvi falar

9. Em relação ao seu conhecimento sobre Visualização de Software. Responda. *

Marcar apenas uma oval.

- ◯ Sei o que é Visualização de Software
- ◯ Sou pesquisador apenas de tópicos relacionados a Test Smells
- ◯ Sei o que é Visualização de Software, mas nunca trabalhei com visualização
- ◯ Sou pesquisador de tópicos relacionados a Visualização de Software
- ◯ Eu nunca ouvi falar

Feedback do Experimento

10. O que você achou do experimento? *

_____

_____

_____

_____

_____

11. Qual ferramenta você usaria? Justifique. *

Marcar apenas uma oval.

- ◯ TSVizzEvolution
- ◯ Vitrum
- ◯ Ambas
- ◯ Nenhuma

12. *

_____

13. Quais vantagens da ferramenta TSVizzEvolution? *

14. Quais desvantagens da ferramenta TSVizzEvolution? *

15. Quais vantagens da ferramenta VITRuM? *

16. Quais desvantagens da ferramenta VITRuM? *

17. Qual ferramenta você achou que representa visualmente melhor as ocorrências de test smells? *

Marcar apenas uma oval.

- ◯ TSVizzEvolution
- ◯ VITRuM

18. O que torna a ferramenta escolhida na questão anterior melhor na visualização das ocorrências de test smells? *

19. Qual ferramenta você achou que exibe melhor a evolução de test smells? *

Marcar apenas uma oval.

- ◯ TSVizzEvolution
- ◯ VITRuM

20. O que torna a ferramenta escolhida na questão anterior melhor na visualização da evolução? *

APÊNDICE K

Id	**Você acha que a informação do possível autor do *test smell* pode ajudar nas decisões de projeto e tomada de decisão? Explique**
Pr1	Eu considero importante conhecer o responsável pelos test smells pois pode direcionar recursos para melhorar as boa práticas em um time de desenvolvimento, como por exemplo treinamentos de boas práticas. Até mesmo na tomada de decisão de promoção de carreira.
Pr2	Sim. Sabendo qual autor foi responsável pelo teste torna a correção mais fácil e rápida.
Pr3	Sim, saber o responsável é uma boa maneira de filtrar e ajudar em uma tomada de decisão mais rápida, fazendo com que todo o processo também seja concluído de forma mais rápida e assertiva.
Pr4	Acredito que sim. A identificação do responsável pode ajudar a justificar o test smell e decidir se ele deve ser retirado ou pode ser mantido
Pr5	Pode sim, como mencionado em respostas anteriores definição dos papéis é de suma importância na gestão do projeto, desse modo ter o responsável mapeado ajuda no sentido de buscar soluções, melhorias e correções sobre eventuais pontos durante a etapa de desenvolvimento, bem como possibilita a equipe identificar com o responsável pontos de discussão para melhorar a comunicação e a implementação das correções de forma a maximizar a qualidade
Pr6	Sim. Acredito que assim é fácil consultar o autor e entender a situação.
Pr7	Sim, pois dá para saber quem tem mais propriedade para entender como o test smell foi gerado.
Pr8	Sim, pois fica melhor orientado as responsabilidades do projeto.
Pr9	Sim, pois permite que os gestores do projeto consigam pensar em formar de organizar e treinar a equipe, para tratar as ocorrências dos TestSmells e aumentar a manutenção do código
Pr10	Pode ajudar sim, sinal que o autor pode melhorar seu código e/ou ter um treinamento.
Pr11	sim, pelo fato de orientar no numero de ocorrências obtidas durante os testes, dando um melhor direcionamento.
Pr12	Se todos os membros da equipe estivessem inserindo determinado tipo de test smell, não seria relevante, já que poderia ser explicado por todos desconhecerem a prática. Caso houvesse apenas um ou poucos autores inserindo determinado test smells seria relevante para que fosse direcionado tal feedback com o objetivo de diminuir a prática.
Pr13	Pode. Facilita o processo de orientação aos desenvolvedores.
Pr14	Sim. Para buscar capacitar e orientar o testador para evitar a geração de novos test smells.
Pr15	Sim. Com essa informação é possível identificar os contribuidores que precisam de algum treinamento para a produção de código de teste com mais qualidade e identificar aqueles que podem estar fornecendo algum tipo de tutoria.
Pr16	Sim, pois ajuda identificar qual tester implementou o test smell e facilita na distribuição de tarefas no time

Fonte: Do autor (2021).

APÊNDICE L

Id	**Qual *test smell* com mais ocorrências? Essa informação pode ser útil para o projeto? Explique.**
Pr1	Assertion Roulette. É importante pois pode-se criar estratégias de incentivo de determinadas práticas durante o desenvolvimento de teste. No caso desse projeto, deve-se conscientizar sobre a importância do parâmetro de explicação nas estruturas dos assertions, de forma a evitar o test smell Assertion Roulette.
Pr2	Assertion Roulette (3997). Informação útil para organizar as correções de acordo com os Tests de maior quantidade de ocorrências.
Pr3	Assertion Roulette, sim, ajuda no direcionamento de decisões.
Pr4	Assertion Roulette - Pode sim, pois identifica qual o test smell que está prejudicando mais a qualidade do código e ajuda tbm ao programador na hora do desenvolvimento do projeto. Para o gestor é importante esse dado para verificar a qualidade da produção dos membros e oferecer possíveis treinamentos ou soluções no próprio time para minimizar a ocorrência de test smell
Pr5	Assertion Roulette. Eu não acho que esse test smell é um grande problema. Se os "assertions" estiver usando nomes de variáveis e métodos com nomes que representam o que fazem, o comentário sobre o teste seria desnecessário. Então a grande ocorrência desse test smell mostra como é comum e nesse caso não seria útil
Pr6	Assertion Roulette. Sim, quanto maior o número mais influência ele terá no código. Reduzi-lo seria uma forma de contribuir mais amplamente na qualidade do código.
Pr7	Assertion Roulette. Sim, pois é possível entender quais os tests smells mais comuns no projeto e então ser adotada uma estratégia para essa diminuição.
Pr8	Assertion Roulette. 66315
Pr9	Assertion Roullete. A informação é útil por trazer uma visão geral sobre o projeto e permitir que os gestores tenham um conhecimento de quais os principais problemas sobre os testes da aplicação, permitindo que tratativas sejam efetuadas e possíveis treinamentos executados.
Pr10	Assertion Roulette. Sim, com essa informação é possível identificar e fazer possíveis correções se necessário.
Pr11	Assertion Roulette. Sim. Essa informação pode ajudar a identificar os test smells que mais ocorrem no projeto o que pode servir para promover uma conscientização aos desenvolvedores sobre os problemas que estão sendo inseridos. Além disso, é possível identificar os pontos que precisam de melhoria.
Pr12	Nesse caso não é útil, visto que o Assertion Roulette é naturalmente o tipo de test smell com maior ocorrência em projetos [ref].
Pr13	Assertion Roulette. No caso do Assertion Roulette eu não considero uma informação útil pois a forma de deteção da ferramenta é muito rígida.
Pr14	Assertion Roulette com 3997. Toda informação pode ser útil para o o projeto depende do contexto. Talvez o(s) test mais crítico pode não ser o de mais ocorrência, pode ser os test que cobrem o core do sistema.

(continua)

Id	**Qual *test smell* com mais ocorrências? Essa informação pode ser útil para o projeto? Explique.**
Pr15	Assertion Roulette, essa informação pode ser muito útil pois, com essa informação podemos verificar onde está a maior quantidade de ocorrência facilmente, facilitando e agilizando o processo de verificações de onde esta sendo utilizado um maior esforço no projeto.
Pr16	Assertion Roulette. Sim, gera um maior controle dos testes

Fonte: Do autor (2021).

APÊNDICE M

Id	Justificativas da Escolha de Usar a Ferramenta `TSVizzEvolution`
Pr1	-
Pr2	A ferramenta contempla vários dados importantes que a Vitrum não oferece
Pr3	Essa ferramenta fornece informações mais específicas sobre os test smells
Pr4	-
Pr5	Possui informações mais precisas e assertivas.
Pr6	Pois traz mais dados de análise e visualização
Pr7	Eu acredito que a ferramenta Vitrum é limitada pois fornece apenas quais test smells afetam uma determinada classe. Enquanto isso, a ferramenta TSVizzEvolution apresenta funcionalidades que podem ser utilizadas durante a tomada de decisão na gestão de um projeto.
Pr8	Considerando apenas o tópico Test Smells, eu usaria a TSVizz porque apresenta diversas informações como a autoria dos test smells e a sua localização de maneira precisa. Além disso, a TSVizz suporta a visualização de 21 tipos de test smells, enquanto que a Vitrum suporta apenas 7. Outro ponto, a TSVizz é uma ferramenta standalone e a Vitrum é um plugin do IntelliJ. Considerando que não preciso que os dados sobre test smells sejam atualizados com frequência (e.g., a cada modificação do código), a escolha pela ferramenta standalone parece mais acertada, pois é necessário menos processamento e independe de ambiente de programação.
Pr9	Ferramenta mais completa com várias opções de filtros.
Pr10	Traz informações mais completas com uma usabilidade simples
Pr11	Nessa ferramenta há maiores possibilidades de saber o que deve ser corrigido pela apresentação de várias informações úteis durante os testes.
Pr12	Achei mais completa.
Pr13	Utilizaria a TSVizzEvolution devido a maior possibilidade de funcionalidades e domínio do processo, apresenta uma gama de possibilidades interessante para o analista de qualidade
Pr14	-
Pr15	TSVizzEvolution apresentou-se mais completa
Pr16	-

Fonte: Do autor (2021).

APÊNDICE N

Id	Justificativas da Escolha de Usar Ambas as Ferramentas
Pr1	O TSVizzEvolution é bastante útil e completo, e o Vitrum facilitaria por ser um Plugin do IntelliJ, mas é bastante incompleto.
Pr2	-
Pr3	-
Pr4	O Vitrum é de fácil uso ao estar alterando o código, podendo ser usado para rápidas checagens sobre a permanência do test smell. Porém o TSVizzEvolution fornece muito mais informação para planejar e realizar as correções.
Pr5	-
Pr6	-
Pr7	-
Pr8	-
Pr9	-
Pr10	-
Pr11	-
Pr12	-
Pr13	-
Pr14	Vitrum tem como vantagem ser integrado com IDE e TSVizzEvolution tem como vantagens poder visualizar várias informações a respeito da evolução de test smells em projetos.
Pr15	-
Pr16	-

Fonte: do Autor (2021).

APÊNDICE O

Id	**O que torna a ferramenta escolhida na questão anterior melhor na visualização das ocorrências de *test smells*?**
Pr1	Apresenta mais informações
Pr2	É mais fácil para visualizar graficamente as informações por meio de grafos.
Pr3	Tanto as abordagens gráficas quando na riqueza de detalhes e informações, permite com que o usuário tenha uma análise mais abrangente que a outra.
Pr4	Clareza e detalhe nas informações fornecidas: - Quantidade de cada test smell - Localização de cada test smell - Responsável por cada test smell - Comparação de duas versões do projeto
Pr5	Mostra os resultados de forma gráfica e textual com todas as informações necessárias.
Pr6	Diferentes tipos de visualização dos problemas, como grafo, gráfico de cores.
Pr7	A granularidade em que os dados são apresentados. Em particular, gostei bastante da treemap para visualização dos test smells no projeto como um todo.
Pr8	A identificação da linha
Pr9	A quantidade, tipos, filtros, versões e flexibilidade de informações coletadas e exibidas para análise.
Pr10	Mais informações a respeito da evolução do projeto considerando test smells comparado com a outra ferramenta.
Pr11	-
Pr12	As informações contidas para visualização.
Pr13	Na hora de identificar no local onde está o erro ela foi mais precisa.
Pr14	A Vitrum não mostra o número de ocorrências e também não mostra os responsáveis pelo desenvolvimento das atividades, desse modo a TSVizzEvolution trabalha melhor.
Pr15	Possui gráficos e números mais assertivos.
Pr16	-

Fonte: Do autor (2021).

APÊNDICE P

Id	O que torna a ferramenta escolhida na questão anterior melhor na visualização da evolução?
Pr1	A ferramenta VITRuM nem chega a fornecer dados sobre a evolução dos smells
Pr2	O gráfico onde mostra a evolução, indicando se ouve uma melhor ou não.
Pr3	A representação escolhida pela ferramenta.
Pr4	Possui informações que a outra não informa.
Pr5	Possui rotinas que se conseguem realizar uma análise mais profunda.
Pr6	A outra ferramenta não possui. A TS permite comparar 2 versões.
Pr7	A possibilidade de comparar duas versões de um projeto em tempo real, sem a necessidade de rodar versões separadas individualmente.
Pr8	Apresenta uma timeline que marca de diferentes cores a inserção, remoção e permanência de test smells entre duas versões.
Pr9	-
Pr10	Dá pra comparar duas versões do sistema
Pr11	Nela existe as opções para escolher que tipo de exibição irá mostrar as informações do resultado.
Pr12	A parte de detalhes de dados passados para o usuário.
Pr13	O número de ocorrências de test smell, a comparação de versões do sistema, desse modo apresenta um arcabouço melhor de possibilidades
Pr14	Mais informações a respeito da evolução do projeto considerando test smells comparado com a outra ferramenta.
Pr15	Tem mais opções de visualização e mais informações sobre os test smells
Pr16	Mostra a diferença entre os testes smells nas diferentes versões

Fonte: Do autor (2021).

APÊNDICE Q

Id	**Você conseguiria corrigir os *test smells* com as informações fornecidas pela ferramenta? Explique**
Pr1	Sim. Seria acrescentado um parâmetro de explicação descrevendo o objetivo do assert.
Pr2	Sim, pois possui todas as informações para chegar até o Teste realizado e a linha específica do resultado.
Pr3	Sim, a ferramenta auxilia agilizando o processo já mostrando qual o método e em qual linha esta a ocorrência a ser verificada.
Pr4	Conseguiria.
Pr5	Eu conseguiria. Colocaria um comentário informando que o assert verifica se o usuário foi autenticado
Pr6	Sim
Pr7	Sim, pois entendendo a natureza do Assertion Roulette e sabendo as linhas em que ocorre dá para saber o caminho para uma correção.
Pr8	Sim, uma vez que a ferramenta disponibiliza as linhas de código e os tipos de test smell que está ocorrendo, o que facilita a identificação de pontos com baixa qualidade e possibilita a melhoria para diminuição das ocorrências
Pr9	Sim, pois a ferramenta fornece o endereço exato de onde o problema foi encontrado, e mostra exatamente a linha do método e em qual classe.
Pr10	Sim, pois ela exibe claramente quais linha o test smell está ocorrendo.
Pr11	sim, pois teria todos os direcionamentos possíveis para identificar o problema.
Pr12	Sim, pois ficaria mais fácil a identificação da linha.
Pr13	Sim. Colocaria a descrição no assert.
Pr14	Sim, seria possível. Seria bem mais rápido identificar onde (linhas) está o problema para corrigir.
Pr15	Sim. A ferramenta apontou para a linha exata em que o problema ocorre, bastaria adicionar uma explicação sobre essa asserção para resolver o problema.
Pr16	Sim, pois informando exatamente a linha com erro vi que estava faltando a conversão do da variável result1 pra string

Fonte: Do autor (2021).

APÊNDICE R

Id	Vantagens `TSVizzEvolution`	Desvantagens `TSVizzEvolution`
Pr1	Ela mostra informações específicas, como autores e número de ocorrência.	As desvantagens são que os grafos podem as vezes mostrar muitas informações e não haver clareza.
Pr2	As principais são mostrar os Autores, e a linha onde ocorre o test smells, e apresenta uma interface gráfica que auxilia numa melhor visualização dos dados	Um pouco mais complexa de ser usada
Pr3	Visualizar várias informações a respeito da evolução de test smells em projetos.	Alguns aspectos de usabilidade poderiam melhorar a utilização da ferramenta.
Pr4	O retorno para o usuário é bem maior.	Não tive muita experiência com ela para encontrar desvantagens, na maior parte enxerguei mais vantagens.
Pr5	Possui uma análise muito mais completa e detalhista das ocorrências, além de mais opções de visualização	No momento talvez um pouco da usabilidade, mas a ferramenta em si já é bastante funcional
Pr6	1- Ferramenta standalone 2- Quantidade de test smells 3- Treemap da quantidade de test smells (gostei bastante desse) 4- Grafos para autoria e apresentação de smells 5- Evolução	1- Foi um pouco difícil identificar todas classes em que ocorrem um determinado tipo de test smell (acho que a atividade 3). Quanto mais classes, mais conexões tem o grafo. Eu diria que uma desvantagem é que a visualização por grafos pode ficar poluída. 2- Depende de uma segunda ferramenta para a geração dos arquivos de entrada, o que pode gerar alguma dificuldade já que preciso conhecer a outra ferramenta.
Pr7	Informações quantitativa de ocorrência de erros, informação quanto a linha do erro	Não sei responder
Pr8	Informar os locais de ocorrências de test smell, as granularidades possíveis, desse modo o filtro por autores, métodos, classes possibilita detectar a conjuntura do processo de forma mais completa auxiliando na tomada de decisão; controle de versão e a quantidade de test smell também são fatores importantes da ferramenta, bem como as visualizações de versões e ocorrências	Não coloco como desvantagens, mas melhorias, sendo uma delas para tornar visualmente na parte de design mais atrativa para o usuário. Um exemplo seria uma única tela com módulos e menus o qual o usuário poderia manipular toda a ferramenta sem a necessidade de abrir e fechar gráficos

(continua)

Id	Vantagens `TSVizzEvolution`	Desvantagens `TSVizzEvolution`
Pr9	A ferramenta TSVizzEvolution não é acoplada em uma IDE,permitindo que seja utilizada de forma independente. Além disso, apresenta a quantidade de cada test smell, informando a localização destes no código de teste, assim como a informação do provável responsável pela introdução de um test smell.	Ao usar a ferramenta, os tipos test smells que um desenvolvedor introduz são mostrados em um ComboBox (Utilizei essa informação para responder a P7), não achei muito intuitivo. Não sei se a TSVizzEvolution possui outra funcionalidade que traz essa informação, se sim, desconsidere esse comentário.
Pr10	Identificar a existência de test smell e a linha onde ele aparece	Não sei
Pr11	.	.
Pr12	Consegue-se visualizar maiores informações sobre os testes, como autor, linhas do código, gráficos.	Interface inicial mais simples.
Pr13	Informações diversas sobre os resultados dos testes.	Ser uma ferramenta externa ao software de desenvolvimento.
Pr14	Possui mais dados de análise detalhados e de formas visuais.	Poderia melhorar nas combinações dos filtros. Poderia existir mais funcionalidades tbm, por exemplo: ter um histórico/gráfico da evolução de várias versões (desde o início do projeto), ou seja, uma integração (se possível) com o git.
Pr15	Quantidade de informações dispostas sobre os tests smells encontrados e formas de visualização deles.	Necessário outro software para gerar os dados e é necessário sempre carregar novamente o arquivo caso tenha mudança no código.
Pr16	Apresenta mais informações	Nenhuma

Fonte: Do autor (2021).

APÊNDICE S

Id	Vantagens VITRuM	Desvantagens VITRuM
Pr1	Ela não usa desenhos para a representação e sim listas com os itens.	As desvantagens é a falta de informação como autores e quantidades.
Pr2	Fácil manuseio	Falta de dados importantes que poderiam auxiliar ainda mais.
Pr3	Ser integrada com IDE (pra quem usa essa IDE).	Ser dependente de IDE (pra quem não usa essa IDE). Exibe poucas informações a respeito da evolução dos Test smells.
Pr4	Apoio ao usuário.	Faltou ser mais detalhada com os dados de retorno para o usuário.
Pr5	É um plugin de uma das maiores IDEs para Java	É bastante vaga nas analises, e também possui alguns problemas de usabilidade
Pr6	1- A apresentação de classes que possuem smells em forma de lista facilita a execução da atividade 3 do experimento. 2- Não depende da geração de arquivos externos para sua utilização, basta clonar o projeto 3- O link entre test smells e métricas estruturais pode ajudar a priorizar quais classes refatorar primeiro. 4- Apresenta outras métricas de teste, como flaky tests (mas fora do escopo dessa pesquisa).	Poucas funcionalidades
Pr7	Visualização de classes e ocorrências de erros	.
Pr8	Usabilidade e interface	1- Apresenta poucos de test smells 2- Não contabiliza os test smells na classe 3- Não apresenta os test smells em granularidade mais fina 3- Apresenta apenas um gráfico de evolução dos test smells no projeto. O gráfico não respondeu aos filtros que apliquei, talvez ainda não esteja tão estável.
Pr9	Não vejo vantagens da VITRuM em relação a ferramenta TSVizzEvolution.	Não apresenta um detalhamento sobre a quantidade de cada test smell, a localização e os responsáveis.
Pr10	Identificar a existência de test smell	Não identifica a linha do test smell
Pr11	.	-
Pr12	Tem-se uma melhor interface para acesso.	Possui poucas informações sobre os test smell
Pr13	Ser incluída no software de desenvolvimento.	Não apresenta informações importantes para tomada de decisões (e correções) após os testes.

(continua)

Id	Vantagens VITRuM	Desvantagens VITRuM
Pr14	Não encontrei.	Identificação de menos tipos de tests smells, poucos detalhes e poucas informações visuais. Por ser uma ferramenta de análise e relatórios, ela é bem fraca no seu propósito. Não exibe uma evolução de antes e depois etc.
Pr15	Fácil acesso por estar ligado diretamente à uma IDE e fácil nova análise ao serem feitas mudanças.	Quase nenhuma informação sobre os tests smells encontrados.
Pr16	Nenhuma	Nenhuma

Fonte: Do autor (2021).

APÊNDICE T

Id	O que você achou do experimento?
Pr1	Muito interessante.
Pr2	O experimento foi interessante, bem explicado, e em todo momento foi possível tirar dúvidas relacionadas ao experimento.
Pr3	Interessante.
Pr4	Agregador, me deu um novo conhecimento na área.
Pr5	Bastante abrangente
Pr6	O experimento foi bem desenhado; foram apresentados conceitos iniciais sobre test smells e a pesquisadora realizou tarefas similares com as ferramentas para mostrar seu uso.
Pr7	ótimo
Pr8	Gostei do experimento, foi bem conduzido pela pesquisadora e acrescentou no sentido de novos conhecimentos e experiências sobre a área de teste
Pr9	O experimento foi muito bem planejado. A pesquisadora conduziu o experimento de forma clara e detalhada, tornando-o uma atividade simples para o participante.
Pr10	Interessante
Pr11	Gostei, claro e objetivo.
Pr12	Experimento legal para comparar o uso de ferramentas que facilitam na dinâmica dos testes.
Pr13	Interessante por mostrar informações relevantes para visualizar os resultados e evolução de um teste.
Pr14	Bem bacana.
Pr15	O experimento foi tranquilo para participar. Como as tarefas possuem um roteiro e acompanhamento, não senti muita necessidade do treinamento.
Pr16	Bom

Fonte: Do autor (2021).