UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

FÁBIO PEREIRA DA SILVA

Visualização de Informação de Depuração: Uma Avaliação Experimental

São Paulo

2017

FÁBIO PEREIRA DA SILVA

Visualização de Informação de Depuração: Uma Avaliação Experimental

Dissertação apresentada à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo para obtenção do título de Mestre em Ciências pelo Programa de Pós-graduação em Sistemas de Informação.

Área de concentração: Metodologia e Técnicas da Computação

Versão corrigida contendo as alterações solicitadas pela comissão julgadora em 15 de dezembro de 2017. A versão original encontra-se em acervo reservado na Biblioteca da EACH-USP e na Biblioteca Digital de Teses e Dissertações da USP (BDTD), de acordo com a Resolução CoPGr 6018, de 13 de outubro de 2011.

Orientador: Prof. Dr. Marcos Lordello Chaim

São Paulo

2017

Autorizo a reprodução e divulgação total ou parcial deste trabalho, por qualquer meio convencional ou eletrônico, para fins de estudo e pesquisa, desde que citada a fonte.

CATALOGAÇÃO-NA-PUBLICAÇÃO

(Universidade de São Paulo. Escola de Artes, Ciências e Humanidades. Biblioteca)
CRB – 8 4625

Silva, Fábio Pereira da
Visualização de informação de depuração : uma avaliação
experimental / Fábio Pereira da Silva ; orientador, Marcos Lordello
Chaim. – 2017.
164 f. : il

Dissertação (Mestrado em Ciências) - Programa de Pós-
Graduação em Sistemas de Informação, Escola de Artes,
Ciências e Humanidades, Universidade de São Paulo.
Versão corrigida

1. Desenvolvimento de software. 2. Teste e avaliação de
software. I. Chaim, Marcos Lordello, orient. II. Título.

CDD 22.ed.– 005.14

Dissertação de autoria de Fábio Pereira da Silva, sob o título “**Visualização de Informação de Depuração: Uma Avaliação Experimental**”, apresentada à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo, para obtenção do título de Mestre em Ciências pelo Programa de Pós-graduação em Sistemas de Informação, na área de concentração Metodologia e Técnicas da Computação, aprovada em 15 de dezembro de 2017 pela comissão julgadora constituída pelos doutores:

Prof. Dr. Marcos Lordello Chaim

Presidente

Instituição: Universidade de São Paulo

Prof. Dr. Marcelo Fantinato

Instituição: Universidade de São Paulo

Prof. Dr. Auri Marcelo Rizzo Vincenzi

Instituição: Universidade Federal de São Carlos

Prof. Dr. Delano Medeiros Beder

Instituição: Universidade Federal de São Carlos

A Deus e à minha família.

Agradecimentos

Em primeiro lugar, gostaria de agradecer a Deus que me deu força para que não desistisse em cada um dos momentos complicados ao longo destes quase três anos de curso.

À minha mãe Aparecida Zenaide Pereira da Silva, ao meu pai Edivaldo da Silva e ao meu irmão Felipe Pereira da Silva que me apoiaram em todos os momentos de dificuldades que passei durante o desenvolvimento deste projeto. Com o apoio da minha família superei momentos em que nada parecia dar certo com muita força de vontade e dedicação. Agradeço por todos os ensinamentos dados em todos os anos da minha vida, fundamentais para o alcance de alguns dos meus objetivos na minha trajetória acadêmica e profissional, sem nunca esquecer das dificuldades que vivi em cada um dos desafios superados. Não apenas nas vitórias; mas nas derrotas que me proporcionaram aprendizado.

Ao meu orientador Marcos Lordello Chaim, por todos os conselhos, ensinamentos no decorrer deste trabalho e superação dos momentos difíceis ao longo do curso, mesmo quando tudo indicava que não conseguiria alcançar este objetivo.

Aos professores da Faculdade de Tecnologia da Zona Leste por tudo que representaram ao longo da minha trajetória acadêmica e profissional em que tive a oportunidade de me graduar em Análise e Desenvolvimento de Sistemas. Além da minha formação, a Fatec Zona Leste teve importância vital para o ingresso no curso de pós-graduação da Universidade de São Paulo e na condução do estudo realizado neste trabalho. Agradeço a todas as pessoas que participaram do estudo e docentes desta instituição; mas em especial aos professores Antonio Rodrigues Carvalho, Andreia Zotovici, Celia Viderman Oliveira, Edson Saraiva, Leandro Colevati dos Santos, Luciano Francisco, Wellington Pinto de Oliveira e Wilson Vendramel.

A Universidade de São Paulo pela oportunidade única de estudar em uma das maiores universidades do mundo e a todos os professores que tiveram a sua parcela de contribuição para que desenvolvesse novos conhecimentos fundamentais para a elaboração do projeto de pesquisa.

Aos amigos Eduardo Hubsch, Felipe Barbosa, Gabriela Lopes e Vitor Pinheiro por todo o apoio dado no decorrer do curso. Ao Higor Amario de Souza e a Mayra Satiko Hosokawa pela ajuda na condução dos experimentos.

A todos que me ajudaram, incentivando sempre a superar cada desafio.

“Cada sonho deixado para trás, representa um pedaço do futuro que deixa de existir.”
(Steve Jobs)

Resumo

SILVA, Fábio Pereira da. **Visualização de Informação de Depuração: Uma Avaliação Experimental**. 2017. 164 f. Dissertação (Mestrado em Ciências) – Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, São Paulo, 2017.

Depuração é a tarefa de localizar e corrigir defeitos em um programa. Apesar do esforço de pesquisa em depuração, especialmente nos últimos anos, ela ainda é realizada da mesma forma desde a década de 60, quando os primeiros depuradores simbólicos foram introduzidos. Localização de defeitos baseada em cobertura (LDC) é uma técnica de depuração promissora devido ao seu baixo custo de execução. LDC identifica os elementos mais suspeitos de um programa ao classificar linhas, métodos, classes e pacotes com maior valor de suspeição. Recentemente, ferramentas de visualização têm sido propostas para representar os valores de suspeição dos elementos de um programa. Entretanto, nenhuma delas foi introduzida em ambientes industriais e a utilização de depuradores simbólicos ainda é predominante. Nesta dissertação, foi avaliada a eficácia, a eficiência e a usabilidade de duas ferramentas de depuração, chamadas CodeForest e Jaguar, em ambientes reais. Jaguar apresenta os trechos mais suspeitos de um programa em uma lista ordenada por seus valores de suspeição. A CodeForest recebe informações de classes, métodos e blocos (conjunto de instruções executadas em sequência) suspeitos para construir uma floresta de cactus tridimensional representando o programa inspecionado. Na CodeForest, as classes são representadas como cactus, os métodos como galhos e os blocos como espinhos de um galho. Em ambas as ferramentas, os elementos do programa recebem cores que variam de acordo com o seu valor de suspeição. A questão básica respondida ao término deste trabalho é se as informações da depuração quando exibidas em uma metáfora visual melhoram a eficácia, a eficiência e a usabilidade na localização de defeitos. A eficácia e a eficiência foram avaliadas, respectivamente, pela capacidade da ferramenta direcionar o desenvolvedor ao método ou linha do defeito e o tempo necessário para localizá-los. A usabilidade das ferramentas foi avaliada por meio de um questionário baseado no modelo TAM (*Technology Acceptance Model*). Os resultados obtidos demonstram que a Jaguar foi mais eficaz, eficiente e com maior grau de usabilidade do que a CodeForest; entretanto, o tamanho do efeito estatístico é insignificante para a eficácia e eficiência e baixo para a usabilidade.

Palavras-chaves: Depuração, Visualização das Informações da Depuração, Localização de defeitos, Experiência de usuário.

Abstract

SILVA, Fábio Pereira da. **Visualization of Debugging Information: An Empirical Assessment**. 2017. 164 p. Dissertation (Master of Science) – School of Arts, Sciences and Humanities, University of São Paulo, São Paulo, 2017.

Debugging is the task of locating and fixing defects in a program. Despite the research effort in debugging, especially in recent years, this task is still carried out in the same way since the 60s when the first symbolic debuggers were introduced. Spectrum-Based Fault Localization (SFL) is a promising debugging technique due to its relative low execution cost. SFL pinpoints the most suspicious program elements by ranking lines, methods, classes and packages with greater suspicious values. Recently, visualization techniques have been proposed to represent the suspicious values of program elements. However, none of them have been introduced at industrial settings and the use of symbolic debuggers is still prevalent. This dissertation assessed the effectiveness, efficiency and usability of two debugging tools, called CodeForest and Jaguar, in real environments. Jaguar presents the most suspicious elements of a program in a list sorted by suspicious values. CodeForest receives lists of suspicious classes, methods and blocks (set of statements executed in sequence) to build a three-dimensional cacti forest representing the program inspected. In CodeForest, classes are represented as cacti, methods as branches and blocks as thorns of a branch. In both tools, the program elements receive colors that vary according to the suspicious values. The basic question answered at the end of this research is whether debugging information when displayed as a visual metaphor improves the effectiveness, efficiency and usability during fault localization. The effectiveness and efficiency were assessed, respectively, by the tool's ability to direct the developer to the faulty method or line and the time spent to locate them. The tools' usability was evaluated using the Technology Acceptance Model (TAM). The results show that Jaguar is more effective, efficient and presented greater usability than CodeForest; however, the statistical effect size is insignificant for effectiveness and efficiency and low for usability.

Keywords: Debugging, Visualization of Debugging Information, Fault Localization, User Experience.

Lista de figuras

Figura 1 – Método Max	23
Figura 2 – Grafo de fluxo de controle do método Max	27
Figura 3 – Grafo de fluxo de controle anotado	28
Figura 4 – Fórmula da heurística Tarantula	36
Figura 5 – Fórmula da heurística Ochiai	37
Figura 6 – Estágios de visualização	39
Figura 7 – Technology Acceptance Model	40
Figura 8 – Representação em anel e particionamento vertical da GZoltar	50
Figura 9 – Representação hierárquica da GZoltar	50
Figura 10 – Representação da ferramenta Tarantula	51
Figura 11 – Representação da ferramenta Code Bubbles	52
Figura 12 – Análise gráfica da ferramenta Bug Maps	53
Figura 13 – Análise estatística realizada pela ferramenta In *Bug	54
Figura 14 – Representação gráfica através de diagramas de sequência	56
Figura 15 – Análise dos pontos do diagrama focados pelo usuário durante a avaliação	56
Figura 16 – Representação tridimensional do código inspecionado	58
Figura 17 – Avaliação da usabilidade das ferramentas	59
Figura 18 – Arquitetura da ferramenta Jaguar	64
Figura 19 – Informações para inspeção apresentadas pela Jaguar	65
Figura 20 – Posicionamento dos cactus na CodeForest	66
Figura 21 – Representação visual da CodeForest	68
Figura 22 – Interação com o código do programa	69
Figura 23 – Tempo médio para interação com o método e linha do defeito	95
Figura 24 – Quantidade média de interações com o método e linha do defeito	95
Figura 25 – Correlação entre as variáveis para a Jaguar	105
Figura 26 – Correlação entre as variáveis para a CodeForest	106
Figura 27 – Facilidade de uso da ferramenta Jaguar	107
Figura 28 – Facilidade de uso da ferramenta CodeForest	108
Figura 29 – Experiência profissional na área de desenvolvimento	110
Figura 30 – Distribuição dos participantes	111

Figura 31 – Participantes que encontraram a classe, método ou linha do defeito . . 114

Lista de tabelas

Tabela 1 – Principais modelos de depuração	31
Tabela 2 – Cobertura do método Max	35
Tabela 3 – Conjunto de testes para o método Max	36
Tabela 4 – Critérios de inclusão/exclusão	43
Tabela 5 – Artigos incluídos na revisão da literatura	45
Tabela 6 – Distribuição dos participantes	76
Tabela 7 – Divisão dos grupos de questões avaliadas	78
Tabela 8 – Questões avaliadas	79
Tabela 9 – Questões do TAM	83
Tabela 10 – Teste de hipóteses	85
Tabela 11 – Escala de avaliação do tamanho do efeito	86
Tabela 12 – Uso das ferramentas durante a atividade	89
Tabela 13 – Interrupções no uso das ferramentas durante a atividade	89
Tabela 14 – Interações com a ferramenta	90
Tabela 15 – Linhas inspecionadas	90
Tabela 16 – Interações com depuradores e o JUnit	91
Tabela 17 – Inclusão de breakpoints	91
Tabela 18 – Inclusão de breakpoints no método ou linha do defeito	92
Tabela 19 – Valores de suspeição	92
Tabela 20 – Interações com o trecho do defeito	93
Tabela 21 – Avaliação da igualdade para a eficácia	97
Tabela 22 – Avaliação das diferenças para a eficácia	97
Tabela 23 – Avaliação da igualdade para a eficiência	98
Tabela 24 – Avaliação das diferenças para a eficiência	99
Tabela 25 – Avaliação da igualdade para a usabilidade	99
Tabela 26 – Avaliação das diferenças para a usabilidade	100
Tabela 27 – Representatividade para o TAM	101
Tabela 28 – Representatividade para os resultados	102
Tabela 29 – Análise de Cronbach	103
Tabela 30 – Análise fatorial para a Jaguar	103

Tabela 31 – Análise fatorial para a CodeForest	104
Tabela 32 – Análise da correlação	104
Tabela 33 – Distribuição das respostas	108
Tabela 34 – Localização do defeito - JSoup	112
Tabela 35 – Localização do defeito - XStream	113
Tabela 36 – Avaliação das funcionalidades da Jaguar	114
Tabela 37 – Avaliação das funcionalidades da CodeForest	116
Tabela 38 – Síntese dos resultados dos logs	117
Tabela 39 – Síntese dos resultados por grupo do TAM	119

Sumário

1	Introdução	17
1.1	<i>Justificativa</i>	19
1.2	<i>Objetivos</i>	20
1.3	<i>Organização</i>	20
2	Conceitos Básicos	22
2.1	<i>Teste de software</i>	22
2.1.1	Defeito, erro e falha	22
2.1.2	Caso de teste, conjunto de teste e oráculo	24
2.1.3	Teste funcional	24
2.1.4	Teste estrutural	26
2.1.5	Teste automatizado	29
2.2	<i>Depuração</i>	30
2.2.1	Processo de depuração	31
2.2.2	Técnicas de depuração	32
2.3	<i>Visualização de software</i>	37
2.3.1	Tipos de visualização	38
2.3.2	Estágios de visualização	38
2.3.3	Visualização na atividade de depuração	39
2.4	<i>Technology Acceptance Model (TAM)</i>	40
2.5	<i>Considerações finais</i>	41
3	Revisão da Literatura	42
3.1	<i>Materiais e métodos</i>	42
3.2	<i>Critérios de seleção de trabalhos</i>	42
3.3	<i>Condução da revisão da literatura</i>	44
3.4	<i>Ferramentas de visualização bidimensionais</i>	48
3.4.1	GZoltar	48
3.4.2	Tarantula	51
3.4.3	Code Bubbles	52
3.4.4	Bug Maps	53

3.4.5	In*bug	54
3.5	<i>Ferramentas de visualização tridimensionais</i>	55
3.5.1	Visualização em múltiplos planos 3D	55
3.5.2	Mapeamento tridimensional dos elementos do programa	57
3.6	<i>Resultados</i>	58
3.6.1	Avaliações de usabilidade	58
3.6.2	Análise do período dos artigos selecionados	59
3.6.3	Outras considerações importantes	60
3.7	<i>Discussão</i>	60
3.7.1	Ferramentas visuais propostas	60
3.7.2	Avaliação em ambientes reais	60
3.7.3	Uso de técnicas de Interação Humano Computador	61
3.7.4	Uso de ferramentas de depuração na prática	62
3.8	<i>Considerações finais</i>	62
4	Ferramentas Avaliadas	63
4.1	<i>Jaguar</i>	63
4.2	<i>CodeForest</i>	65
4.3	<i>Considerações finais</i>	70
5	Planejamento do Experimento	71
5.1	<i>Questões de pesquisa</i>	71
5.2	<i>Ambiente do Experimento</i>	72
5.2.1	Participantes	72
5.2.2	Preparo do ambiente	73
5.2.3	Programas e defeitos selecionados	74
5.3	<i>Divisão dos experimentos</i>	75
5.4	<i>Procedimentos</i>	75
5.5	<i>Análise de dados</i>	77
5.5.1	Análise dos arquivos <i>log</i>	78
5.5.2	Questões do TAM	82
5.5.3	Testes estatísticos	84
5.6	<i>Considerações finais</i>	86

6	Resultados do Experimento	88
6.1	<i>Resultados dos arquivos de log</i>	88
6.2	<i>Testes estatísticos</i>	96
6.2.1	Teste de hipóteses para avaliação da eficácia	96
6.2.2	Teste de hipóteses para a eficiência	98
6.2.3	Teste de hipóteses para avaliação da usabilidade	99
6.2.4	Tamanho do efeito	101
6.3	<i>Resultados do TAM</i>	103
6.3.1	Coefficiente de Cronbach	103
6.3.2	Análise fatorial	103
6.3.3	Análise da correlação	104
6.3.4	Representação dos dados	106
6.3.5	Distribuição das respostas	108
6.4	<i>Resultados do questionário</i>	109
6.4.1	Experiência na área de desenvolvimento	109
6.4.2	Distribuição dos participantes	111
6.4.3	Localização de defeitos	112
6.4.4	Opinião sobre a Jaguar	114
6.4.5	Opinião sobre a CodeForest	115
6.5	<i>Discussão</i>	116
6.5.1	Síntese dos resultados do experimento	117
6.5.2	Síntese dos resultados do TAM	118
6.5.3	Síntese dos resultados do questionário	120
6.5.4	Questões de pesquisa	120
6.6	<i>Ameaças à validade</i>	122
6.7	<i>Considerações finais</i>	123
7	Conclusões	124
7.1	<i>Contribuições</i>	125
7.2	<i>Trabalhos Futuros</i>	125

Referências¹	127
--	------------

¹ De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

Apêndice A – Termo de Consentimento Livre Esclarecido (TCLE)	133
Apêndice B – Apresentação	134
Apêndice C – Material de treinamento	144
C.1 <i>Material de treinamento - Jaguar Linha</i>	144
C.2 <i>Material de treinamento - Jaguar método</i>	146
C.3 <i>Material de treinamento - CodeForest</i>	148
Apêndice D – Questionário	153
Anexo A – Submissão do projeto no Comitê de Ética e Pesquisa	163
Anexo B – Comprovante de aprovação no Comitê de Ética e Pesquisa	164

1 Introdução

A Engenharia de Software dedica-se a todos os aspectos da construção do software (SOMMERVILLE, 2007, p.5). Ela evoluiu significativamente nas últimas décadas procurando estabelecer técnicas, critérios, métodos e ferramentas para a produção de aplicações (BARBOSA et al., 2007).

Durante o processo de desenvolvimento de software, uma das principais características buscadas é a implementação de aplicações com o menor número possível de defeitos, com um alto grau de usabilidade e que estejam de acordo com as expectativas dos usuários. As necessidades das organizações são dinâmicas e diferentes setores da economia mudam a todo momento. A busca por qualidade no produto final é constante e quanto mais impactantes forem as falhas descobertas pelo usuário maior é a tendência do sistema se tornar rapidamente obsoleto.

Muitas técnicas, ferramentas e métodos foram, e continuam a ser, desenvolvidos com o objetivo de atribuir qualidade a um sistema. Uma das principais atividades no desenvolvimento de software é a depuração. Segundo ARAKI, FURUKAWA e CHENG (1991), ela dedica-se à localização e à correção de defeitos em um programa. Para Myers, Badgett e Sandler (2012, p.9), juntamente com a atividade de testes, ela é responsável pelo consumo de mais de 50% dos custos totais de um projeto.

Entretanto, a atividade de depuração não acompanhou esta evolução, sendo ainda realizada praticamente da mesma forma desde a década de 1960. Ela ocorre nas diversas fases do ciclo de desenvolvimento possuindo características diferentes entre elas (DELAMARO et al., 2007, p.293). Os desenvolvedores geralmente utilizam depuradores simbólicos para acompanhar a execução do programa e observar as alterações nas variáveis durante a depuração (JONES; HARROLD; STASKO, 2002).

Dentre as técnicas propostas ao longo das últimas décadas, o fatiamento de programas realiza a seleção de um conjunto de comandos que afetam os valores de uma ou mais variáveis em determinado ponto do programa, auxiliando o desenvolvedor a concentrar a sua atenção em uma parte reduzida do código (DELAMARO et al., 2007, p.301-p.304). A depuração delta, por sua vez, visa identificar trechos do programa que provocaram uma falha comparando o estado da aplicação em situações em que a falha ocorre com outras em que ela não ocorre (ZELLER, 2002, p.407). Essas técnicas não têm sido utilizadas em

ambientes industriais seja por gerar uma quantidade excessiva de comandos suspeitos, no caso do fatiamento de programas, seja por consumirem muito tempo e memória para obterem seus resultados. Em outras palavras, ambas as técnicas não são escaláveis para ambientes reais.

Segundo Renieris e Reiss (2003), a técnica de localização de defeitos baseada em cobertura (LDC) é definida por um conjunto de componentes, como comandos, blocos de comandos, predicados e unidades (e.g., métodos), cobertos durante a execução do teste. Ela utiliza informações de cobertura desses componentes em casos de testes que passam, e que não passam, para identificar os trechos mais suspeitos de conter o defeito. O objetivo é reduzir o tempo gasto pelos desenvolvedores na depuração. Esta técnica apresentou bons resultados em estudos realizados, sendo mais escalável para uso em ambientes reais.

As técnicas de depuração baseadas em cobertura de código utilizam os dados de cobertura, heurísticas e os resultados de execução dos testes (passou ou falhou) para atribuir valores de suspeição para cada componente do programa avaliado. Eles são calculados em geral com base nas frequências de execução de comandos nos casos de testes. Quanto maior for o número de vezes que um componente for executado por casos de teste que falharam, maior é a probabilidade do defeito estar presente nele (SOUZA, 2012).

Diante do aumento significativo da complexidade dos sistemas computacionais, ferramentas que auxiliem os desenvolvedores na atividade de depuração são de grande importância para a diminuição do tempo gasto na localização de defeitos (JONES; BOWRING; HARROLD, 2007).

Ferramentas que oferecem representações visuais das informações de suspeição dos componentes têm sido propostas (JONES; HARROLD; STASKO, 2002; MUTTI, 2014; PEREZ; ABREU, 2013). As ferramentas Tarantula (JONES; HARROLD; STASKO, 2002), GZoltar (PEREZ; ABREU, 2013) e CodeForest (MUTTI, 2014) utilizam diferentes metáforas como estratégias de visualização.

Porém, há poucos experimentos que demonstrem a eficácia e a eficiência dessas ferramentas. Em avaliações com GZoltar (PEREZ; ABREU, 2013), as informações de suspeição de componentes colaboraram para encontrar mais rapidamente um defeito semeado no programa XStream¹, solucionando-o em menos tempo do que sem a utilização da ferramenta visual. Mutti (2014) realizou um experimento exploratório para avaliar a usabilidade da ferramenta CodeForest.

¹ <http://x-stream.github.io/>

Essas ferramentas têm apresentado resultados promissores; entretanto, a maior parte delas não foi validada por meio de experimentos quanto a sua eficácia e eficiência de forma a comprovar a sua utilidade prática em situações reais de desenvolvimento (SOUZA; CHAIM; KON, 2016). Essa limitação impede essas ferramentas de serem utilizadas por desenvolvedores durante a atividade de depuração na indústria.

1.1 Justificativa

Nos últimos anos foram desenvolvidas ferramentas que auxiliam os desenvolvedores na atividade de depuração por meio de representações visuais de informações de depuração baseadas em cobertura. Elas apresentaram resultados promissores para uso em ambientes reais, com metáforas visuais (JONES; HARROLD; STASKO, 2002; MUTTI, 2014; PEREZ; ABREU, 2013).

Porém, a utilidade de ferramentas visuais como auxílio aos desenvolvedores na atividade de depuração precisa ser ainda confirmada por experimentos. Mais ainda, é preciso verificar se as ferramentas visuais adicionam valor em relação às técnicas puramente textuais. Algumas questões relevantes para o desenvolvedor são: qual dessas representações é mais eficaz e efetiva para a localização de defeitos? Qual delas é mais fácil de usar?

Dentro desse contexto, este projeto visa avaliar o uso de ferramentas visuais na atividade de depuração. A questão básica a ser respondida é:

A metáfora visual da ferramenta CodeForest aumenta a eficácia e a eficiência da localização de defeitos em relação à metáfora puramente textual da ferramenta Jaguar?

Jaguar apresenta os elementos mais suspeitos (e.g., linhas) na forma de uma lista. CodeForest, por sua vez, utiliza uma floresta tridimensional de cactus para representar os elementos mais suspeitos de um programa. A Jaguar pode ser utilizada em duas versões, a lista de linhas e a lista de métodos mais suspeitos. As duas versões foram consideradas no projeto de pesquisa.

A eficácia é avaliada pela capacidade da ferramenta direcionar ou não o desenvolvedor ao sítio do defeito em uma sessão de depuração; e a eficiência pelo tempo despendido pelo desenvolvedor para localizar o defeito utilizando a ferramenta.

Portanto, a justificativa desta pesquisa encontra-se na necessidade de desenvolver e avaliar ferramentas de depuração visuais em relação a ferramentas textuais, visando identificar qual dessas representações é capaz de melhor auxiliar os desenvolvedores durante a atividade de depuração, contribuindo assim para que sejam construídas ferramentas visuais mais escaláveis e úteis para a indústria de software.

1.2 Objetivos

O objetivo deste trabalho é avaliar a eficácia, eficiência e a usabilidade de duas ferramentas, Jaguar e CodeForest, que utilizam metáforas distintas para representar as informações de depuração.

As ferramentas foram escolhidas por utilizarem técnicas LDC e por terem sido desenvolvidas pelo grupo de pesquisa em Engenharia de Software Experimental da Universidade de São Paulo, *Software Analysis and Experimental Group (SAEG)*.

Os objetivos específicos desta pesquisa são definidos a seguir:

- Planejar e executar um experimento que compare as ferramentas avaliadas neste estudo;
- Avaliar a eficácia das ferramentas CodeForest e Jaguar;
- Avaliar a eficiência das ferramentas CodeForest e Jaguar;
- Avaliar a escalabilidade das ferramentas estudadas neste projeto de pesquisa para utilização em situações reais de desenvolvimento;
- Identificar dificuldades associadas à usabilidade das ferramentas de depuração enfrentadas pelos desenvolvedores durante a localização de defeitos.

1.3 Organização

Este capítulo dedicou-se à apresentação do contexto, justificativa e objetivos desse projeto de pesquisa.

O Capítulo 2 detalha os conceitos básicos envolvidos nas atividades de teste e depuração de software, tipos de técnicas e ferramentas existentes. Também são apresentados conceitos básicos sobre o *Technology Acceptance Model* utilizado para avaliação da usabilidade das ferramentas avaliadas.

O Capítulo 3 apresenta uma revisão da literatura sobre ferramentas de depuração. Os trabalhos abordados são separados por aplicações bidimensionais e tridimensionais.

As principais características das ferramentas CodeForest e Jaguar são detalhadas no Capítulo 4.

O Capítulo 5 descreve o projeto experimental detalhando as questões de pesquisa, os procedimentos realizados, os programas selecionados, a seleção dos participantes e a análise de dados.

O Capítulo 6 apresenta os resultados e as discussões dos resultados obtidos.

Por último, são apresentadas as conclusões, bem como as contribuições e limitações deste estudo.

2 Conceitos Básicos

Neste capítulo serão apresentados os conceitos fundamentais de teste de software, depuração, técnicas de depuração baseadas em cobertura, visualização das informações e introduzido o *Technology Acceptance Model*.

2.1 Teste de software

Teste de software é o processo de executar um programa ou sistema com a finalidade de detectar defeitos (MYERS; BADGETT; SANDLER, 2012). Juntamente com a depuração correspondem a metade dos custos totais de um projeto. Outro fator relevante é o custo com a correção de defeitos, que cresce aproximadamente 10 vezes à medida que o projeto avança (MYERS; BADGETT; SANDLER, 2012, p.11).

Para Delamaro et al. (2007, p.2), teste de software é uma atividade dinâmica realizada com o intuito de executar um programa ou modelo com algumas entradas em particular e verificar se o seu comportamento está de acordo com a especificação. Segundo SOMMERVILLE (2007, p.359), é impossível testar um programa para encontrar todos os defeitos que ele possui. Mesmo para aplicações triviais de baixa complexidade, é necessário estabelecer uma estratégia de testes capaz de avaliar se o programa tem comportamento adequado.

Diante do papel vital que a atividade de teste de software representa, nesta seção serão apresentados os seus principais conceitos e técnicas utilizadas.

2.1.1 Defeito, erro e falha

Devido ao crescimento de estudos sobre a atividade de teste de software, é bastante comum vários termos serem utilizados para o mesmo conceito, sendo confundidos no cotidiano dos profissionais e apresentados de maneiras distintas na literatura. Diante disso, há a necessidade de defini-los visto que serão utilizados ao longo de todo o trabalho. Eles foram definidos com base no glossário de termos proposto pela IEEE (1990).

- Defeito: imperfeição em um componente ou sistema que pode provocar o seu mal funcionamento. Um passo, processo ou definição de dados incorreta inserido por

alguém que realizou a escrita do código do programa. Ele é consequência de um engano cometido pelo desenvolvedor no entendimento de uma informação.

- Erro: um estado incorreto durante a execução de um programa que representa uma manifestação concreta de um defeito em um artefato de software. A existência de um defeito pode ocasionar o surgimento de um erro.
- Falha: incapacidade de um programa realizar a sua funcionalidade de acordo com o requisito que ele deveria cumprir, ou seja, é caracterizada quando o comportamento operacional do software é diferente do esperado pelo usuário. Uma falha pode ser causada por um ou vários defeitos; porém, alguns deles podem nunca desencadeá-la.

A existência de um defeito pode ocasionar o surgimento de um erro modificando o estado de um programa durante determinado instante de sua execução. Tal modificação pode desencadear o surgimento de uma falha identificada por alguém que estiver realizando a utilização do sistema. Embora distintos, utiliza-se o termo erro apresentando o mesmo significado de defeito e falha (DELAMARO et al., 2007, p.2). A ocorrência de uma falha indica a presença de um defeito, permitindo assim que seja iniciada a depuração do programa (SOUZA, 2012).

Com o objetivo de ilustrar estas definições, será apresentado na Figura 1 um programa que visa identificar o maior elemento dentro de um arranjo de números inteiros.

Figura 1 – Método Max

L	B	Comando
-	-	int max(int[] array, int length)
-	1	{
1	1	int i = 0;
2	1	int max = array[++i]; // array[i++] ;
3	2	while(i < length)
-	3	{
4	3	if(array[i] > max)
5	4	max = array[i];
6	5	i++;
-	5	}
7	6	return max;
-	6	}

Fonte: Chaim e Araujo (2012)

O método possui um defeito na segunda linha de comando devido ao pré-incremento na variável i antes de atribuir o valor na primeira posição no arranjo. Caso o maior valor esteja na primeira posição do vetor, não será retornado o valor correto ocasionando

uma falha. Além disso, se for informado somente um elemento, no momento em que a comparação for realizada, será lançada uma exceção “`ArrayOutOfBoundsException`”, ocasionando outra falha perceptível ao usuário.

2.1.2 Caso de teste, conjunto de teste e oráculo

Um caso de teste é formado pelas entradas e resultados obtidos na saída do teste (DELAMARO et al., 2007, p.1-p.7). Um conjunto de teste é constituído dos casos de testes existentes para um programa.

Por exemplo, o método Max pode ser testado com um caso de teste $t_1 = \{20, 5, 10, 2\}$. O testador representa o papel de oráculo e irá verificar se a saída fornecida corresponde ao resultado esperado ou não. Neste caso, devido o maior elemento estar na primeira posição do vetor ele não será retornado. Entretanto, se o caso de teste for informado com os seguintes valores $t_1 = \{5, 10, 20, 2\}$ ele apresentará o maior elemento corretamente.

Casos de testes são a base de todas as ferramentas avaliadas neste estudo. Elas utilizam os resultados dos casos de teste (sucesso ou falha), trechos e caminhos do programa percorridos por eles para identificar o grau de suspeição dos elementos do programa avaliado conterem defeitos.

2.1.3 Teste funcional

A atividade de teste de software pode ser realizada por três técnicas, estrutural, funcional e baseada em defeitos, sendo diferenciadas pela fonte de dados utilizada para definir os requisitos de testes (DELAMARO et al., 2007, p.9).

Segundo Pressman (2006, p.318), o teste funcional, também conhecido como teste de caixa preta, examina algum aspecto fundamental do sistema, pouco se preocupando com a estrutura lógica interna. Ele é realizado com base na especificação da aplicação, sem acesso direto ao código fonte. No teste funcional, visualiza-se apenas o lado externo do programa, ou seja, as entradas e saídas fornecidas.

Para Delamaro et al. (2007, p.9), teste funcional é uma técnica na qual o programa ou sistema é considerado uma caixa preta e para testá-lo são fornecidas entradas avaliando se as saídas geradas estão em conformidade com os objetivos especificados. Em princípio, o teste funcional deveria submeter o programa ou sistema a todas as possíveis entradas, o

que seria o chamado teste exaustivo. No entanto, o domínio de entrada pode ser infinito ou muito grande fazendo com que ele seja impraticável. Para enfrentar essa limitação, foram desenvolvidos critérios de teste para selecionar um subconjunto relevante dos domínios de entrada e de saída para teste.

Critérios de teste funcional

Segundo Delamaro et al. (2007), o teste funcional ocorre com base na identificação das funções que o software deve conter e com casos de testes capazes de verificar se elas estão sendo realizadas. A criação de casos de teste é guiada por critérios de testes funcionais. Os mais conhecidos são o particionamento de equivalência, a análise do valor limite e o teste funcional sistemático.

O particionamento de equivalência divide o domínio de entrada em classes de equivalência (partições) válidas e inválidas, de acordo com a especificação do programa. Testes são criados para verificar cada classe identificada. Este critério pode ser adotado com o objetivo de tornar a quantidade de dados de entrada finita e viável para domínios de entrada infinitos. Os dois passos a serem realizados são a identificação das classes e a geração de casos de testes por meio da seleção de pelo menos um elemento para cada partição. A adoção desse critério apresenta como principais benefícios a redução do tamanho do domínio de entrada e a criação de casos de testes baseados unicamente na especificação (DELAMARO et al., 2007, p.12-p.13).

O critério análise de valores limites estabelece que os casos de testes devem exercitar as extremidades do domínio de entrada. Ele parte da premissa de que os defeitos costumam estar próximos às extremidades das classes. Ao invés de selecionar qualquer elemento da classe, o testador deve realizar o teste em cada uma de suas fronteiras, sendo frequente a seleção do menor e maior valor possível dentro de um intervalo, um número limite de valores ou um conjunto ordenado (MYERS; BADGETT; SANDLER, 2012, p.69).

O critério de teste funcional sistemático combina o particionamento de equivalência e a análise de valores limites. Ele requer que pelo menos dois casos de testes sejam criados para cada partição para minimizar o mascaramento de falhas devido ao problema de correção coincidente. Além disso, deve ser realizada a avaliação dos limites de cada partição. O teste funcional sistemático considera, dentre outros, valores numéricos, casos

especiais, números reais, intervalos variáveis, dados do tipo texto, valores válidos e inválidos (DELAMARO et al., 2007, p.14-p.16).

2.1.4 Teste estrutural

Segundo Pressman (2006, p.318), o teste estrutural, também conhecido como teste de caixa branca, é baseado em um exame rigoroso dos detalhes procedimentais de um programa, ou seja, as condições a serem testadas são definidas com base em sua implementação. Caminhos lógicos internos e colaborações entre os componentes de um sistema devem ser testados por meio da criação de casos de testes que exercitem conjuntos específicos de condições ou ciclos. Diferentemente do teste funcional, ele é totalmente baseado na estrutura interna do sistema.

De acordo com Myers, Badgett e Sandler (2012), os casos de testes devem ser estruturados de modo que garantam que todos os caminhos independentes de um programa e condições lógicas sejam executadas pelo menos uma vez, bem como os valores limites e todas as estruturas de dados internas da aplicação.

Segundo Delamaro et al. (2007, p.48), o teste estrutural é complementar às demais técnicas existentes uma vez que cobre classes distintas de defeitos. Os critérios utilizados no teste estrutural podem ser baseados em complexidade, no fluxo de controle ou no fluxo de dados.

Definições sobre teste estrutural

O teste estrutural baseia-se no conhecimento da estrutura interna do programa, sendo aspectos de implementação fundamentais para a geração/avaliação de casos de testes. A grande maioria dos critérios de teste estrutural utiliza uma representação do programa denominada *Grafo de Fluxo de Controle* (GFC).

O GFC representa um programa P por meio de blocos (nós) e arcos. P é decomposto em um conjunto de blocos disjuntos de comandos de forma que todos eles possuem um único predecessor e sucessor, com exceção do primeiro e do último comando (DELAMARO et al., 2007, p.51). A execução do primeiro comando de um bloco acarreta a execução de todos os outros comandos desse bloco. Os arcos representam transferências de fluxo de

controle entre os blocos. Normalmente, os arcos estão associados aos comandos de fluxo de controle do programa como *while*, *for* e *switch*.

A Figura 2 apresenta o grafo de fluxo de controle gerado para o método Max apresentado na Figura 1.

Figura 2 – Grafo de fluxo de controle do método Max

Fonte: Chaim e Araujo (2012)

Para Maldonado et al. (1997), as ocorrências de uma variável em um programa podem ser consideradas uma *definição*, *indefinição* ou um *uso*. Uma definição de uma variável ocorre quando o seu valor é armazenado em uma posição de memória, isto é, quando ela está no lado esquerdo de uma atribuição ou em um comando de entrada de dados. Uma variável é indefinida quando não se tem acesso ao valor contido nela ou a sua localização deixa de estar definida em memória. A ocorrência de uma variável em todas as situações que ela não estiver sendo definida é caracterizada como um uso.

Segundo Chaim, Maldonado e Jino (2002), o uso de uma variável pode ser distinguido como *c-uso* e *p-uso*. O *c-uso* afeta diretamente uma computação que estiver sendo realizada, permitindo que o resultado obtido em uma definição seja observado enquanto o *p-uso* afeta diretamente o fluxo de controle de um programa.

Na Figura 3, o GFC do método Max (Figura 2) é anotado com inclusão das definições e dos usos (*c-usos* e *p-usos*). No primeiro nó ocorre a definição das variáveis *i*, *array*, *length* e *max*. No segundo as variáveis *i* e *length* são utilizadas, sendo caracterizadas como um *p-uso* por afetarem diretamente o fluxo de controle do programa. O mesmo ocorre no terceiro nó com as variáveis *i*, *array* e *max*. No quarto nó a variável *max* é

definida e as variáveis i e $array$ são consideradas como um c-uso por não afetarem o fluxo de controle. A mesma situação se repete no quinto e no sexto nó com as variáveis i e max .

Figura 3 – Grafo de fluxo de controle anotado

Fonte: Chaim e Araujo (2012)

Critérios de teste estrutural

Os critérios baseados em complexidade utilizam informações sobre a complexidade do programa para derivar os requisitos de testes. Um critério bastante conhecido desta classe é o critério de McCabe que utiliza a complexidade ciclomática para derivar os requisitos de testes. Ela é considerada como uma métrica de software que proporciona uma medida quantitativa da complexidade lógica de um programa (MCCABE, 1976).

O valor da complexidade ciclomática estabelece o número de caminhos linearmente independentes de um programa. Um caminho linearmente independente é definido como qualquer caminho que introduza pelo menos um novo conjunto de instruções de processamento ou uma nova condição (DELAMARO et al., 2007, p.55-p.56). Portanto, o teste baseado na medida de complexidade ciclomática estabelece que os conjuntos de casos de teste devem exercitar pelo menos uma vez cada caminho linearmente independente.

Critérios baseados em fluxo de controle utilizam apenas características de controle da execução de um programa como comandos ou desvios para determinar quais estruturas são necessárias. Para Delamaro et al. (2007, p.56), os critérios mais conhecidos desta classe são:

- *Todos os nós.* Exige que cada comando de um programa, ou seja, cada nó, seja exercitado pelo menos uma vez pelos casos de teste. É o mínimo esperado de uma boa atividade de testes.
- *Todos os arcos.* Requer que todos os desvios de um programa sejam exercitados ao menos uma vez pelo conjunto de casos de teste.
- *Todos os caminhos.* Requer que todos os caminhos possíveis de um programa sejam exercitados pelos casos de teste. Entretanto, na maioria dos casos, esta tarefa é impraticável porque o número pode ser muito grande ou mesmo infinito.

Crítérios baseados em fluxo de dados selecionam caminhos de testes com base nas associações entre a definição de uma variável e seus possíveis usos subsequentes. Uma motivação para o uso desse critério foi a indicação de que mesmo para programas pequenos o teste baseado unicamente no fluxo de controle não era eficaz para revelar a presença de defeitos simples e triviais. Diante desse problema, torna-se necessário estabelecer uma hierarquia de critérios de teste entre os critérios todos os nós (menos exigente) e todos os caminhos (extremamente exigente) para que o teste estrutural se torne mais rigoroso (DELAMARO et al., 2007, p.57).

2.1.5 Teste automatizado

O teste é uma fase dispendiosa e trabalhosa do processo de software. Devido a isso, inúmeras ferramentas que apoiam a atividade de teste de software foram desenvolvidas nas últimas décadas. Atualmente essas ferramentas são capazes de oferecer uma grande variedade de recursos e seu uso pode colaborar para a redução dos custos dos testes realizados (SOMMERVILLE, 2007, p.371).

Testes automatizados são programas ou *scripts* simples que exercitam funcionalidades do sistema que está sendo testado e fazem verificações automáticas nos resultados obtidos. Esta abordagem permite que todos os casos de testes sejam rapidamente avaliados e repetidos sempre que necessário, sem a necessidade de qualquer esforço adicional em sua execução (BERNARDO; KON, 2008).

Como um dos principais exemplos das ferramentas propostas está o JUnit¹ que é um conjunto de classes codificadas em Java que possibilita a criação de um ambiente

¹ <http://www.junit.org>

de testes automatizado. Eles devem ser escritos de maneira que indiquem se o sistema testado comportou-se conforme o esperado utilizando como base os critérios definidos na elaboração dos casos de testes (SOMMERVILLE, 2007, p.371).

2.2 Depuração

Segundo ARAKI, FURUKAWA e CHENG (1991), depuração é o processo que consiste na coleta e análise de informações em uma ou mais execuções que auxiliem na identificação das causas de comportamentos errôneos no sistema. A manifestação de um defeito e a sua causa interna podem não ter nenhuma relação óbvia uma com a outra. Ela ocorre como consequência de um teste bem sucedido. Ou seja, quando a execução de um caso de teste não foi realizada corretamente (PRESSMAN, 2006, p.421).

É possível observar grande variação na habilidade de depuração entre programadores com o mesmo nível de formação e experiência, que pode ser afetada por fatores como a ansiedade e a indisposição para que o desenvolvedor aceite que um engano foi cometido (PRESSMAN, 2006, p.421).

A depuração é uma consequência do teste bem sucedido, isto é, quando um caso de teste apresenta uma falha. Esse teste é avaliado até que uma divergência entre o resultado esperado e os valores reais obtidos seja encontrada. Em grande parte das vezes os resultados esperados representam um sintoma interno de uma causa subjacente ou defeito; porém, desconhecida (PRESSMAN, 2006, p.421). Sintomas internos são caracterizados como os valores de uma ou mais variáveis em determinado ponto de execução do programa (CHAIM; MALDONADO; JINO, 2002).

Recentemente, diante das dificuldades apresentadas para a realização da atividade de depuração ferramentas que utilizam as informações do teste foram propostas, como a Jaguar e a CodeForest que serão avaliadas neste estudo e detalhadas no Capítulo 4.

No contexto deste trabalho, é fundamental a apresentação dos principais conceitos envolvidos na atividade de depuração que são utilizados pelas ferramentas avaliadas e nos problemas enfrentados pelos desenvolvedores em seu cotidiano. Esta seção dedica-se à apresentação dos principais conceitos relacionados à atividade de depuração incluindo processos e técnicas de depuração.

2.2.1 Processo de depuração

Foram propostos vários modelos com o objetivo de guiar o processo de depuração de maneira mais eficiente. A Tabela 1 resume os principais modelos de depuração existentes.

Tabela 1 – Principais modelos de depuração

Modelos	Descrição
Modelo hipótese-validação.	Baseado na elaboração e validação interativa de hipóteses estabelecidas pelo programador que podem ser modificadas quando necessário até a correção do defeito (ARAKI; FURUKAWA; CHENG, 1991).
Modelo sistemático de depuração.	Propõe uma interpretação do modelo hipótese-validação baseado na execução em reverso da aplicação e utilizando fatiamento dinâmico do programa para restringir o espaço de busca (AGRAWAL, 1991).
Modelo de depuração de Pan.	Enfatiza a conexão entre as atividades de teste de software e depuração com ênfase no uso das informações do teste (PAN, 1993).
Modelo de depuração de Chan.	Propõe que, para a criação e validação de uma hipótese o desenvolvedor executa o programa com um caso de teste que provoque uma falha, e inspeciona os dados de entrada e saída, bem como relacionamentos entre eles, os valores das variáveis e comandos que foram executados (CHAN, 1997).
Modelo de depuração depois do teste.	É focado na localização do defeito usando informações do teste (CHAIM; MALDONADO; JINO, 2002).

Modelo caça-caçador.	Propõe um modelo de depuração em que o defeito é a caça e o programador é o caçador. O programador (caçador) procura por informações (rastros e pegadas) que o levam até o defeito (caça). Esse modelo indica que a busca pelo defeito é menos sistemática do que a proposta pelo modelo hipótese-validação (LAWRANCE; BOGART, 2013).
----------------------	---

Fonte: Fábio Pereira da Silva, 2017

Os dois principais modelos para o processo de depuração são o hipótese-validação e o caça-caçador. O primeiro é utilizado como referência para a elaboração da maioria das abordagens propostas e o segundo sugere que a busca pelo defeito é mais intuitiva do que sistemática. Lawrance e Bogart (2013) conduziram um experimento com programadores e observaram que eles combinam esses dois modelos na busca pelos defeitos, sendo que o caça-caçador é utilizado durante a maior parte do tempo.

2.2.2 Técnicas de depuração

Várias técnicas foram propostas ao longo dos últimos anos com o objetivo de auxiliar os desenvolvedores durante a depuração. Esta seção apresenta as principais técnicas existentes.

Rastreamento de eventos e inspeção

Essa técnica, como o próprio nome diz, visa rastrear eventos e inspecionar o estado do programa, entendendo-se como estado do programa o conjunto de variáveis que ele possui e sua pilha de execução. Na forma mais básica, consiste na inclusão de comandos de escrita em pontos determinados do programa e a impressão de valores de variáveis suspeitas. Entretanto, essa técnica é na maioria das vezes implementada por meio de depuradores simbólicos.

Tipicamente, depuradores simbólicos estão presentes em Ambientes Integrados de Desenvolvimento, também conhecidos como IDE (*Integrated Development Environment*). Em geral, a depuração é realizada manualmente pelos programadores que a partir da execução dos casos de testes que falharam inserem *breakpoints* no programa para verificar os valores contidos nas variáveis ou fluxos de execução da aplicação (JONES; BOWRING; HARROLD, 2007).

O depurador simbólico mostra informações como a posição no código fonte onde a execução foi interrompida (DELAMARO et al., 2007, p.298). Outras informações como a posição da *thread* da aplicação e os valores das variáveis de interesse são exibidas para que o desenvolvedor realize a sua avaliação e possa tirar as suas conclusões (MEGA; KON, 2005). Em geral, os depuradores permitem alterações de variáveis e de fluxo de um programa em tempo de execução. Alguns depuradores simbólicos experimentais permitem execução reversa retornando o programa a um estado anterior (DELAMARO et al., 2007)

A utilização de depuradores simbólicos ainda é predominante em ambientes industriais de desenvolvimento. Uma das razões do sucesso dos depuradores simbólicos é permitir a visualização da execução dentro do esperado pelo desenvolvedor. Além disso, a maior parte dos programadores possuem uma grande familiaridade com eles devido à forma de visualização intuitiva baseada no próprio código do sistema (MEGA; KON, 2005).

Fatiamento de programas

A técnica de fatiamento de programas, em inglês *program slicing* (WEISER, 1981), realiza a seleção de um conjunto de comandos que afetam os valores de uma ou mais variáveis em determinado ponto do programa restringindo o espaço de busca durante a depuração (DELAMARO et al., 2007, p.301-p.304).

As variáveis e o ponto do programa afetado definem o *critério de fatiamento* que será utilizado. Os comandos selecionados podem ser determinados de maneira estática ou dinâmica. No fatiamento estático, os comandos selecionados podem afetar as variáveis em relação a qualquer entrada possível enquanto no fatiamento dinâmico os comandos selecionados afetam os valores das variáveis para uma determinada entrada (DELAMARO et al., 2007, p.301).

Portanto, as fatias — conjunto de comandos selecionados — são formadas para identificar quais instruções do programa afetaram ou serão afetadas direta ou indiretamente pelo valor de uma ou mais variáveis em dado ponto do programa ou da execução (critério de fatiamento) (BRINTLEY; GALLAGHER, 1996). A falha observada em um teste deve ser mapeada para um sintoma interno, como o valor incorreto em um dado ponto de execução da aplicação, para a definição da fatia do programa suspeito (CHAIM; MALDONADO; JINO, 2002).

As fatias podem ser determinadas por *forward slicing* ou *backward slicing*. No primeiro caso, são avaliadas as instruções afetadas por uma variável, retornando subconjuntos de comandos dependentes. O segundo caso avalia quais instruções do programa possuem influência sobre os valores das variáveis de interesse, retornando os comandos dos quais elas dependem (MOTA et al., 1996).

O problema com a técnica de fatiamento de programas é que o tamanho das fatias pode ser muito grande, em especial quando definidas de maneira estática. O custo de execução para a realização do fatiamento dinâmico é alto. Isto porque é necessário o rastreamento dos comandos e das posições de memória para identificar quais comandos afetam um determinado critério de fatiamento (DELAMARO et al., 2007, p.301-p.302). Devido aos problemas apresentados, o fatiamento de programas possui pouca escalabilidade para uso em ambientes industriais (CHAIM; MALDONADO; JINO, 2002).

Depuração delta

O algoritmo de depuração delta, proposto por Zeller (2002), é inspirado na busca binária. Ele pode ser utilizado com vários propósitos como reduzir o tamanho da entrada de casos de testes ou selecionar os comandos que provocam a ocorrência de uma falha. Para realizar a simplificação da entrada de um caso de teste ou identificar os comandos que causam uma falha, muitas execuções do programa são necessárias. No caso da identificação de comandos, é preciso comparar o estado do programa nas diferentes execuções.

Embora tenha apresentado resultados promissores, a depuração delta não é escalável para uso em ambientes reais devido à necessidade de muitas execuções para a obtenção de informações úteis para a depuração (DELAMARO et al., 2007, p.310-p.312).

Localização de defeitos baseada em cobertura

A localização de defeitos baseada em cobertura (LDC), em inglês *Spectrum-based Fault Localization* (SFL), é uma das abordagens mais promissoras e tem recebido muita atenção devido a sua simplicidade e eficácia (XIE et al., 2011).

LDC baseia-se nos componentes (comandos, nós ou blocos, ramos, associações definição-uso, métodos) cobertos durante a execução dos testes para identificar quais trechos do código possuem maior probabilidade de conter defeitos (RENIERIS; REISS, 2003). Heurísticas são usadas para ordenar os elementos mais suspeitos da aplicação que são mapeados em linhas do código do programa a serem avaliadas pelo desenvolvedor.

As informações sobre a cobertura de código podem ser representadas por meio de uma matriz. A Tabela 2 apresenta a matriz de cobertura do método Max descrito na Figura 1. A coluna L representa as linhas de comando do programa. As colunas t_1 a t_5 demonstram a cobertura dos casos de testes descritos na Tabela 3. Os itens assinalados indicam que a linha foi coberta pelo teste. Ao final das colunas t_1 a t_5 , o termo S significa que a sua execução foi finalizada com sucesso e F indica que ela falhou. As colunas c_{00} , c_{01} , c_{10} e c_{11} apresentam os coeficientes de cobertura dos componentes.

Os componentes c_{00} e c_{01} indicam o número de vezes que o componente c não foi executado por casos de teste bem sucedido e de falha, respectivamente, enquanto c_{10} e c_{11} indicam o número de vezes que c foi executado por casos de teste de sucesso e de falha. Por último a coluna H_1 contém o valor de suspeição de cada linha do método Max.

Tabela 2 – Cobertura do método Max

L	t_1	t_2	t_3	t_4	t_5	c_{00}	c_{10}	c_{01}	c_{11}	H_1
1	•	•	•	•	•	0	3	0	2	0,5
2	•	•	•	•	•	0	3	0	2	0,5
3	•	•	•	•		0	3	1	1	0,33
4	•	•	•	•		0	3	1	1	0,33
5	•	•	•	•		0	3	1	1	0,33
6	•	•	•	•		0	3	1	1	0,33
7	•	•	•	•		0	3	1	1	0,33
-	S	S	S	F	F	-	-	-	-	-

Fonte: Souza (2012)

Tabela 3 – Conjunto de testes para o método Max

t_1	$\{([1,2,3],3),3\}$
t_2	$\{([5,5,5],3),5\}$
t_3	$\{([2,10,1],3),10\}$
t_4	$\{([4,2,3],3),4\}$
t_5	$\{([4],1),4\}$

Fonte: Souza (2012)

O valor de suspeição da coluna H_1 foi determinado utilizando uma heurística chamada Tarantula (JONES; HARROLD; STASKO, 2002). As heurísticas são utilizadas para atribuir valores de suspeição para os componentes do programa avaliado (nós, ramos, associações definição-uso, métodos).

As heurísticas levam em consideração os componentes que foram, ou não foram, executados por casos de testes que passam ou que falham. Quanto mais um componente for executado por casos de teste que falham, maior a probabilidade dele conter um defeito. Por outro lado, quanto mais um componente é executado por casos de teste que passam, menor é a sua probabilidade de conter um defeito. As heurísticas também consideram que quando um componente não é executado por casos de teste que falham, ele tem menor probabilidade de conter um defeito. De forma contrária, um componente que não é executado por casos de teste que passam tem maior chance de conter um defeito, desde que seja executado por casos de teste que falham.

A partir dos valores calculados, as técnicas de localização de defeitos realizam a classificação em ordem decrescente dos resultados obtidos retornando uma lista com os elementos mais suspeitos de um programa (SOUZA, 2012). Tarantula determina as frequências de execução dos comandos nos casos de teste para identificar a sua probabilidade de apresentar defeitos, conforme apresentado na Figura 4.

Figura 4 – Fórmula da heurística Tarantula

$$H_T = \frac{\frac{c_{11}}{c_{11}+c_{01}}}{\frac{c_{11}}{c_{11}+c_{01}} + \frac{c_{10}}{c_{10}+c_{00}}} \quad (1)$$

Fonte: Jones, Bowring e Harrold (2007)

Segundo Tarantula as linhas 1 e 2 são as mais suspeitas. Portanto, o desenvolvedor será orientado a olhar essas duas linhas primeiro. Nesse exemplo, o defeito está localizado na linha 2.

Poderiam ter sido utilizadas outras heurísticas para a identificação dos elementos mais suspeitos do método Max, como a heurística Ochiai proposta por Abreu et al. (2008) utilizada na área de biologia molecular. A fórmula que descreve a Ochiai é apresentada na Figura 5.

Figura 5 – Fórmula da heurística Ochiai

$$O_{chiai} = \frac{c_{ef}}{\sqrt{(c_{ef} + c_{nf})(c_{ef} + c_{ep})}} \quad (2)$$

Fonte: Abreu et al. (2008)

Na fórmula, Cep representa a quantidade de vezes em que o comando não foi executado por um caso de teste de sucesso. Cnf indica a quantidade de vezes em que o comando não foi executado por um caso de teste que falhou. Cef representa a quantidade de vezes em que o comando foi executado por um caso de teste que falhou.

O crescimento de estudos sobre técnicas LDC ocorre pelo seu conceito simples, de fácil aplicação e eficácia demonstrada em avaliações iniciais colaborando para a redução do esforço necessário durante a depuração, tornando-a mais escalável em relação as outras técnicas propostas para uso em ambientes reais (XIE et al., 2011).

As ferramentas avaliadas neste projeto de pesquisa, detalhadas no Capítulo 4, utilizam representações visuais e textuais dos valores de suspeição atribuídos a linhas, métodos, classes e pacotes de programas escritos em linguagens orientadas a objetos.

2.3 Visualização de software

A visualização de informações é uma área que procura criar novos algoritmos e técnicas de visualização para grandes quantidades de informações, incluindo documentos Web, estruturas de diretórios de arquivos e outros tipos de dados abstratos que apresentem informações temporais, lineares, bidimensionais, tridimensionais, multidimensionais e hierárquicas (CEMIN, 2001).

A falta de visibilidade das informações durante o desenvolvimento de software colabora para a baixa produtividade dos programadores, ocultando a complexidade do sistema em especial em projetos que envolvam grandes equipes. A sobrecarga de informações ocorre quando o usuário tem muita informação ao seu alcance, mas não tem condições de tratá-las ou de encontrar o que realmente deseja (BALL; EICK, 1996).

Para Cemin (2001), a visualização tem como objetivo facilitar o entendimento das informações relacionadas ao processo de desenvolvimento de software e ainda atuar como uma forma de comunicação entre as pessoas que estejam manipulando a mesma informação. Técnicas de visualização proporcionam uma maneira simples e intuitiva de entender o significado dos dados apresentados.

Com ela é possível explorar o potencial da cognição humana e habilidades de percepção que não são possíveis apenas com informações textuais (LINTERN et al., 2003). A visualização de informações dos artefatos de um sistema pode ajudar os desenvolvedores a lidar com essa complexidade e assim aumentar a produtividade dos programadores (LINTERN et al., 2003).

2.3.1 Tipos de visualização

A visualização de software pode ser estática, dinâmica e evolutiva. A visualização estática busca ilustrar as estruturas, relacionamentos e propriedades das entidades que o sistema possui. Os dados de entrada são obtidos sem a necessidade de execução do programa. A visualização dinâmica é baseada em informações processadas durante a execução do software visando auxiliar na compreensão do comportamento da aplicação, incluindo dados relacionados a chamadas de métodos e troca de mensagens. A visualização evolutiva possibilita uma avaliação das informações armazenadas no repositório de dados ao longo do tempo (CASERTA; OLIVIER, 2011).

2.3.2 Estágios de visualização

O processo de visualização possui quatro etapas, obtenção e entrada de dados, transformação dos dados, aplicação do modelo gráfico e processamento visual e cognitivo. Na obtenção e entrada de dados, as informações da estrutura do software e valores de

métricas como troca de mensagens e instanciação de objetos são ligadas ao tipo de visualização adotada. Na transformação dos dados, são filtradas as informações necessárias. A aplicação do modelo gráfico transforma os dados obtidos em alguma forma de exibição. Por último, ocorre o processamento visual e cognitivo por parte do usuário (WARE, 2004, p.4). Essas etapas são descritas na Figura 6.

Figura 6 – Estágios de visualização

Fonte: Ware (2004, p.4)

2.3.3 Visualização na atividade de depuração

A utilização de ferramentas de visualização nas atividades relacionadas ao processo de desenvolvimento e evolução de um produto de software cresceu ao longo dos últimos anos. Recentemente foram propostas ferramentas visuais com o objetivo de auxiliar os desenvolvedores na atividade de depuração (JONES; HARROLD; STASKO, 2002; MUTTI, 2014; GOLVEIA; CAMPOS; ABREU, 2013). Para alcançar os objetivos definidos na Seção 1.2, no próximo capítulo será apresentada uma revisão da literatura que demonstra as evoluções que ocorreram nas últimas décadas na construção de ferramentas visuais para depuração.

2.4 Technology Acceptance Model (TAM)

A aceitação de tecnologias está sendo estudada desde a década de 1970 e os estudos resultaram em inúmeros fatores que podem influenciar o uso da Tecnologia da Informação. Foram desenvolvidos vários modelos para auxiliar e identificar níveis de uso dos sistemas (LEGRIS; INGHAM; COLLERETTE, 2003). Entretanto, dentre as diversas teorias propostas, o TAM é considerado um dos mais influentes e utilizados pelos pesquisadores para descrever a aceitação de uma dada tecnologia.

O modelo TAM proposto por Davis, Bagozzi e Warshaw (1989) foca nas razões que levam os usuários a aceitarem ou rejeitarem uma determinada tecnologia. O modelo teórico foi desenvolvido e testado levando em consideração as características de aceitação do usuário em utilizar um determinado sistema de informação.

O TAM surgiu através de um contrato entre a IBM Canadá e o Massachusetts Institute of Technology (MIT) na década de 1980 visando avaliar o potencial de mercado de novos produtos da marca e possibilitar uma explicação das determinantes que levavam os usuários à utilização de computadores. (DAVIS; BAGOZZI; WARSHAW, 1989).

O modelo representa uma adaptação do modelo da *Theory of Reasoned Action* (TRA). O TRA tem como objetivo investigar as determinantes comportamentais que indicam a probabilidade de uma pessoa decidir executar ou não uma determinada tarefa. Ele já foi extensamente pesquisado e mostrou sucesso na previsão e explicação do comportamento humano em várias áreas (DAVIS; BAGOZZI; WARSHAW, 1989).

A Figura 7 demonstra o modelo de aceitação de novas tecnologias.

Figura 7 – Technology Acceptance Model

Segundo Davis, Bagozzi e Warshaw (1989), o TAM está alicerçado em dois pilares fundamentais, a utilidade percebida e a facilidade de uso percebida. Este modelo visa representar o impacto dos fatores externos relacionados ao sistema de informação em relação aqueles internos do indivíduo como as atitudes e intenções de uso. A utilidade é dada pelo grau com que uma pessoa acredita que o uso de um sistema particular pode melhorar o seu desempenho. A facilidade de uso é medida pelo grau em que uma pessoa acredita que o uso de um sistema de informação será de livre esforço.

No modelo, a intenção de uso de um sistema é medida através do comportamento que o usuário apresenta para executar uma dada tarefa relacionada ao uso futuro do sistema (TAYLOR; TITMUSS; LEBRE, 1999).

A utilidade é o fator com maior relevância para que o usuário aceite de forma mais rápida utilizar um produto tecnológico, diante da percepção da facilidade de uso que ele possui (TAYLOR; TITMUSS; LEBRE, 1999).

Os efeitos das variáveis externas na intenção de uso são medidos através da utilidade e facilidade de uso percebida. Em outras palavras, a medição de cada fator do TAM ocorre por meio de um conjunto de afirmações para avaliar o pontencial de uso futuro das tecnologias estudadas (LEGRIS; INGHAM; COLLERETTE, 2003).

No contexto deste projeto de pesquisa, foi criado um conjunto de afirmações para cada um dos grupos e fatores do TAM que serão descritos no Capítulo 5.

2.5 Considerações finais

Neste capítulo, foram apresentados os principais conceitos sobre as atividades de teste de software e depuração. Primeiramente, foram abordados conceitos de teste de software relacionados ao objetivo deste trabalho. Em seguida, foram contextualizados os modelos que apoiam o processo de depuração e elencados os benefícios e dificuldades de algumas das técnicas existentes. Ao final, foi discutido o papel de ferramentas visuais nas atividades de Engenharia de Software e apresentadas as principais características do modelo *Technology Acceptance Model* que foi utilizado na avaliação das ferramentas.

3 Revisão da Literatura

Este capítulo apresenta uma revisão da literatura com o objetivo de identificar as evoluções que ocorreram ao longo dos últimos anos no desenvolvimento de ferramentas visuais que auxiliem os desenvolvedores na atividade de depuração.

3.1 *Materiais e métodos*

Uma revisão da literatura é o processo de busca, análise, descrição de um corpo do conhecimento que visa demonstrar o estágio atual de uma determinada área da ciência. Ela oferece uma visão abrangente de pesquisas e contribuições anteriores conduzindo ao alcance dos objetivos estabelecidos e proporciona com os resultados obtidos que sejam realizadas futuras investigações e o desenvolvimento de novos estudos (FELIZARDO et al., 2017).

Esta revisão foi dividida em quatro fases. Primeiramente, foram estabelecidos os critérios de seleção de trabalhos, condução, extração de dados e avaliação dos resultados que serão explicados nas próximas seções, incluindo um detalhamento das ferramentas bidimensionais e tridimensionais encontradas na pesquisa.

3.2 *Critérios de seleção de trabalhos*

Esta revisão da literatura tem como objetivo a identificação das evoluções ocorridas nos últimos anos no desenvolvimento de ferramentas de visualização que auxiliem os desenvolvedores na atividade de depuração. Nela são avaliados os métodos, técnicas adotadas e problemas enfrentados na utilização em ambientes reais.

Para direcionar a pesquisa foram estabelecidas algumas questões a serem respondidas ao término desta revisão.

1. Quais foram as ferramentas visuais ou gráficas propostas ao longo dos últimos anos com o objetivo de auxiliar os desenvolvedores na atividade de depuração?
2. Alguma das ferramentas apresentadas teve a sua usabilidade, eficácia e eficiência avaliada em ambientes reais?

3. Foram encontrados trabalhos que utilizam técnicas de Interação Humano Computador para a construção de ferramentas visuais alinhadas as expectativas dos seus usuários?

4. O que ainda falta para que ferramentas visuais de depuração sejam adotadas na indústria de software?

Para a realização da busca de trabalhos relacionados foram consideradas as seguintes palavras chaves: “depuração”, relacionada com os termos *debugging*, *fault localization* e *coverage*; “visualização”, associada aos termos *software visualization*, *visualization of debug information* e *debugging tool*; “avaliação”, associada aos termos *evaluation* e *assessment*; e “experiência de usuário”, associada com o termo *user experience*. Foram considerados artigos publicados em português ou inglês.

Tabela 4 – Critérios de inclusão/exclusão

Critérios de inclusão	Critérios de exclusão
Trabalhos disponíveis integralmente em bases de dados científicas ou em versões impressas.	Trabalhos que não avaliem as técnicas de depuração ou que não apresentem novas abordagens para esta atividade.
Trabalhos publicados a partir de 1990.	Trabalhos com o ano de publicação inferior a 1990.
Trabalhos que tragam propostas de melhorias à atividade de depuração, seja pelo desenvolvimento de novas ferramentas, seja por estudos com desenvolvedores durante a sua utilização.	Trabalhos que não apresentem testes estatísticos que comprovem a eficiência do método aplicado.
Trabalhos que apresentem comparativos entre os depuradores simbólicos com ferramentas visuais.	Trabalhos que não utilizem técnicas de visualização para a atividade de depuração.
Trabalhos que realizem comparativos de aplicações bidimensionais com ferramentas tridimensionais que auxiliem os desenvolvedores na atividade de depuração.	Trabalhos que não sejam relacionados ao tema proposto.

Fonte: Fábio Pereira da Silva, 2017

Buscou-se nesta revisão da literatura trabalhos que envolvam os seguintes tópicos:

- Trabalhos que abordaram o desenvolvimento de ferramentas de visualização que auxiliem os desenvolvedores durante a depuração.
- Trabalhos que realizaram análises estatísticas detalhadas que comprovem a eficiência do método aplicado.
- Trabalhos que apresentaram o uso de técnicas de Interação Humano Computador para entendimento de como as pessoas executam a atividade de depuração.
- Trabalhos com ênfase na construção de ferramentas de reengenharia que auxiliem na localização de defeitos.
- Trabalhos que abordaram o desenvolvimento de aplicações que auxiliem na atividade de depuração em sistemas distribuídos.

3.3 Condução da revisão da literatura

Seguindo os critérios de seleção apresentados, a revisão foi conduzida por um período de três meses (abril/2015 a agosto/2015) e refeita em julho de 2016 para identificar se houve algum trabalho relevante desenvolvido nesse período. Para a obtenção dos estudos primários foi formada a seguinte *string* de busca aplicada no repositório digital da IEEE com base nas palavras chaves: *(Debugging) or ((Software Visualization) and (Visualization of debug information) and (debugging tool)) or ((fault location) and (coverage)) or (user experience)*.

Ao todo foram localizados 99 trabalhos e 20 deles foram incluídos após a leitura do título e do resumo dos artigos, considerando os critérios de inclusão e exclusão. Os demais foram descartados por não atenderem aos objetivos estabelecidos ou representarem repetições de trabalhos que já haviam sido incluídos previamente. Diante do baixo número de ferramentas visuais de depuração encontradas foram incluídos também trabalhos que apresentem alguma forma gráfica de exibição de informações durante a depuração.

A Tabela 5 demonstra os trabalhos incluídos com breves comentários que resumem as principais abordagens utilizadas. Nas próximas seções serão apresentadas uma visão detalhada dos trabalhos, como eles foram distribuídos e a relevância para os resultados desta revisão.

Tabela 5 – Artigos incluídos na revisão da literatura

Título	Ano	Autores	Comentários
Visualizing Multiple Program Executions to Assist Behavior Verification.	2009	Chunying Zhao, Kang Zhang, Jie Hao e W. Eric Wong.	Abordagem sobre a utilização de diagramas UML para representações tridimensionais.
A debugging tool for software evolution.	1995	D. Abramson e R. Sosic.	Introduz uma ferramenta para auxílio em testes de regressão.
A lightweight awareness service for industrial environments.	1997	M. Mock, M. Gergeleit e E. Nett.	Propõe uma técnica estatística de monitoramento para sistemas distribuídos.
How Programmers Debug, Revisited: An Information Foraging Theory Perspective.	2010	Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, Scott D. Fleming.	Abordagem de avaliação com programadores por meio do modelo caçador.
A visual studio plug-in for CProver.	2013	Mohamed Nassim Seghir e Daniel Kroening.	Implementação do plugin CProver.
Gemini: maintenance support environment based on code clone analysis.	2002	Tobias Schwalb, Philipp Graf e and Klaus D. Muller Glaser	Descrição da ferramenta Gemini para a localização de códigos duplicados.
Monitoring user interactions for supporting failure reproduction.	2013	Tobias Roehm, Nigar Gurbanova, Bernd Brügge, Christophe Joubert e Walid Maalej.	Apresentação da ferramenta <i>Timeline</i> para a reprodução de ações de usuários para a localização de defeitos.

Visualization of test information to assist fault localization.	2002	James A. Jones, Mary Jean Harrold e John Stasko.	Demonstração da ferramenta Tarantula para visualização de comandos suspeitos durante a depuração, mencionada como trabalho correlato em vários outros artigos.
Development and debugging of distributed virtual reality applications.	2013	Marcelo de Paiva Guimarães, Diego Colombo Dias, Bruno Barberi Gnecco, Valéria Fari-nazzo Martins e Luiz Francisco Contrí.	Apresentação da ferramenta GTracer para a depuração de mensagens.
The Challenge of Helping the Programmer during Debugging.	2014	Steven P. Reiss.	Apresentação da plataforma Code Bubbles para monitoramento de todos os passos percorridos durante a depuração.
A perspective on the evolution of live programming.	2013	Steven L. Tanimoto.	Descrição da técnica de <i>Liveness</i> na construção de novas ferramentas de visualização.
Language-agnostic program rendering for presentation, debugging and visualization.	2000	Christian S. Collberg, Sean Davey e Todd A. Proebsting.	Apresentação da ferramenta <i>Art</i> para a análise da especificação do sistema.
Guided test visualization: Making sense of errors in concurrent programs.	2011	Saint Wesonga, Eric G Mercer e Neha Rungta.	Demonstração da ferramenta G-JPF para a depuração de aplicações <i>multithreads</i> .

Cues for scent intensification in debugging.	2013	Alexandre Perez e Rui Abreu.	Aplicação do modelo caça-caçador na construção da ferramenta GZoltar.
Bug Maps: A Tool for the Visual Exploration and Analysis of Bugs.	2012	Andre Hora, Nicolas Anquetil, Stephane Ducasse, Muhammad Bhatti, Cesar Couto, Marco Tulio Valente e Julio Martins.	Demonstração da ferramenta <i>Bug Maps</i> para a representação gráfica dos componentes inspecionados.
Model-Level Debugging of Embedded Real-Time Systems.	2012	Padma Iyengar, Elke Pulvermueller, Clemens Westerkamp, Michael Uelschen e Juergen Wuebbelmann.	Propõe a técnica MDD para monitoramento de aplicações distribuídas com base em representação gráfica por meio de diagramas UML.
XSUDS-SDL: a tool for diagnosis and understanding software specifications.	1999	J. J. Li e J. R. Horgan.	Apresentação da ferramenta XSUDS-SDL para a análise de especificações de sistemas.
In *Bug: Visual analytics of bug repositories.	2014	Tommaso Dal Sasso e Michele Lanza.	Descreve a ferramenta <i>In *Bug</i> para representação gráfica e estatística de defeitos encontrados.
Visual programming of MPI applications: Debugging and performance analysis.	2013	Stanislav Böhm, Marek Behálek, Ondrej Meca e Martin Šurkovský.	Apresentação da ferramenta <i>Kaira</i> para monitoramento e depuração de aplicações distribuídas.

Case study: Visual debugging of finite element codes.	2002	Patricia Crossno, David H. Rogers e Christopher J. Garasi.	Introduz o uso de ferramentas tridimensionais na atividade de depuração.
---	------	--	--

Fonte: Fábio Pereira da Silva, 2017

3.4 Ferramentas de visualização bidimensionais

O uso de tecnologias tridimensionais (3D) está crescendo gradativamente desde o desenvolvimento de novas técnicas de processamento gráfico ao longo das últimas duas décadas. Entretanto, abordagens que apresentam tecnologias bidimensionais (2D) ainda são bastante expressivas nas mais variadas áreas não somente devido às tendências do passado, mas porque praticamente todos os recursos computacionais trabalham em cenários bidimensionais. Tais características tornam mais simplificada a adaptação das pessoas a estas tecnologias. Ferramentas 2D para apoio à atividade de depuração ou que representam as informações de um produto de software graficamente foram mais encontradas. As ferramentas bidimensionais encontradas são apresentadas nesta seção.

3.4.1 GZoltar

A GZoltar é uma ferramenta visual com o objetivo de auxiliar os desenvolvedores na atividade de depuração por meio de representações bidimensionais construídas em HTML5 que guiam o programador até a localização do defeito (GOLVEIA; CAMPOS; ABREU, 2013). Ela é uma evolução da ferramenta Zoltar que oferece apenas representações textuais (RIBOIRA; ABREU; ABREU, 2010).

A aplicação é um *plugin* integrável com o IDE Eclipse¹ voltada para programas desenvolvidos na linguagem Java e pode ser utilizada simultaneamente com casos de testes automatizados com o *framework* JUnit (LUIS, 2012).

A GZoltar necessita das informações de cobertura dos casos de teste para realizar a localização de defeitos utilizando a heurística Ochiai (GOLVEIA; CAMPOS; ABREU,

¹ <http://www.eclipse.org>

2013). A ferramenta elabora uma lista dos elementos candidatos a conter o defeito ordenada pelo valor de suspeição de cada um deles que podem ser pacotes, classes, métodos ou linhas de um programa (GOLVEIA; CAMPOS; ABREU, 2013).

A GZoltar possui um esquema de cores que indica a probabilidade de cada elemento do programa conter defeitos. As informações da aplicação podem ser visualizadas de três maneiras: representação em anel, particionamento vertical e ordenação hierárquica. As visualizações geradas são interativas, permitindo que o usuário navegue na estrutura do projeto e exibidas a partir do esquema de coloração. A cor verde representa um baixo valor de suspeição, amarelo indica um valor médio, laranja representa um percentual alto e vermelho indica os elementos do programa com maior probabilidade de conter defeitos (GOLVEIA; CAMPOS; ABREU, 2013).

A ferramenta oferece uma visão abrangente e integrada de um projeto de software com um esquema de navegação intuitivo. Tais características permitem que os desenvolvedores encontrem as causas das falhas apresentadas (GOLVEIA; CAMPOS; ABREU, 2013). Nas Figuras 8 e 9, são demonstradas as formas de exibição que podem ser utilizadas com a ferramenta GZoltar.

Figura 8 – Representação em anel e particionamento vertical da GZoltar

Fonte: Golveia, Campos e Abreu (2013)

Figura 9 – Representação hierárquica da GZoltar

Fonte: Golveia, Campos e Abreu (2013)

Em um estudo inicial, desenvolvedores utilizaram a GZoltar para localizar um defeito. Quando o defeito foi avaliado apenas com o uso de depuradores simbólicos, somente 35% deles conseguiram localizá-lo no tempo proposto enquanto com a GZoltar todos os participantes conseguiram encontrá-lo (PEREZ; ABREU, 2013). Entretanto, é importante ressaltar que foi realizado um único experimento com somente um defeito, tornando necessária a realização de avaliações mais abrangentes.

3.4.2 Tarantula

O trabalho de Jones, Harrold e Stasko (2002) apresenta a ferramenta Tarantula. Ela tem como objetivo auxiliar os desenvolvedores na atividade de depuração por meio de uma ferramenta gráfica que utiliza cores para mapear visualmente os elementos mais suspeitos de um programa. A Tarantula identifica os casos de testes que foram executados com sucesso ou com falha e representa as informações a partir de um esquema de coloração. Com base nestas informações, os desenvolvedores podem inspecionar os trechos do programa mais suspeitos de conter defeitos. A Figura 10 apresenta a ferramenta.

Figura 10 – Representação da ferramenta Tarantula

Fonte: Jones, Harrold e Stasko (2002)

3.4.3 Code Bubbles

De acordo com Reiss (2014), a ferramenta Code Bubbles visa reduzir a dificuldade enfrentada por programadores durante a depuração. Ela apresenta o histórico da depuração por meio de diagramas de sequência, do tempo de processamento da *thread* atual e da apresentação de informações através de uma interface gráfica, o que facilita a compreensão do código pelos desenvolvedores. É detalhado onde o programa se encontra incluindo o tempo gasto até chegar ao trecho demarcado.

A Code Bubbles oferece também análises estatísticas que auxiliem os desenvolvedores na visualização de informações tanto do status anterior quanto atual oferecendo uma previsão do que ainda será executado durante a depuração. A ferramenta proposta pode ser visualizada na Figura 11.

Figura 11 – Representação da ferramenta Code Bubbles

Fonte: Reiss (2014)

3.4.4 Bug Maps

Segundo Hora et al. (2012), a ferramenta Bug Maps serve como fonte de informações para a localização de defeitos em sistemas de grande porte. Ela extrai dados sobre o defeito, vinculando-os a artefatos do software distintos. Em seguida, tenta rastreá-los em outros módulos do sistema que podem ser afetados pelos trechos defeituosos.

Ela oferece mecanismos que automatizam a localização e análise de dados de um repositório do software. Os resultados são apresentados de maneira visual apoiando a avaliação dos defeitos encontrados e na compreensão de determinados pontos de um programa. Primeiramente, a ferramenta extrai de modo temporal o número de defeitos localizados, em seguida realiza a integração dos modelos extraídos a partir do código fonte e oferece uma representação visual dos dados obtidos. Basicamente, são oferecidas duas funcionalidades principais: o mapeamento do módulo do sistema que se deseja avaliar e a posterior visualização dos resultados (HORA et al., 2012).

A Figura 12 demonstra a forma de apresentação de informações da Bug Maps onde pequenos retângulos representam as classes e os containers demonstram os pacotes considerados na avaliação.

Figura 12 – Análise gráfica da ferramenta Bug Maps

Fonte: Hora et al. (2012)

3.4.5 In*bug

Segundo Sasso e Lanza (2014), a ferramenta In *Bug oferece visualizações interativas, com subsídios para que haja uma análise detalhada das informações obtidas durante a depuração, navegação e compreensão dos relatórios de falhas, exibindo as informações na plataforma Web.

Ela permite que seja realizada a verificação do ciclo de vida do defeito, seleção de informações adicionais com relação a um ponto específico retornado durante a depuração. São fornecidos dados como a data inicial em que este módulo foi codificado no repositório, o status e a última data em que o componente da aplicação sofreu alterações. Opções adicionais de filtragem e gráficos permitem a realização de uma avaliação detalhada durante todos os períodos em que houve modificações no repositório (SASSO; LANZA, 2014). A aplicação proposta pode ser visualizada na Figura 13.

Figura 13 – Análise estatística realizada pela ferramenta In *Bug

Fonte: Sasso e Lanza (2014)

3.5 Ferramentas de visualização tridimensionais

O uso de tecnologias tridimensionais já está sendo explorado em diversas áreas da computação devido as evoluções ocorridas em processamento gráfico.

Visualizações dos artefatos de um software em formas tridimensionais permitem a exploração dos dados em ambientes cada vez mais próximos do que vivemos. Atualmente, estão sendo buscadas novas abordagens que apresentem melhorias na intuição, usabilidade e eficiência dessas ferramentas e técnicas. Elas possuem como objetivo diminuir a complexidade de analisar um produto internamente e oferecem novas formas de apresentação de informações. Entretanto, mesmo com essa evolução e com o desenvolvimento de novas ferramentas para apoio à atividade de depuração, em especial nos últimos cinco anos, grande parte delas apresenta abordagens bidimensionais ou ferramentas de apoio que exibem informações de modo textual. Ainda há um número escasso de estudos que introduzem o uso de ferramentas tridimensionais. Esta seção apresenta as ferramentas tridimensionais encontradas.

3.5.1 Visualização em múltiplos planos 3D

A ferramenta proposta por Zhao et al. (2009) tem como objetivo auxiliar os desenvolvedores na atividade de depuração combinando múltiplos planos tridimensionais.

A aplicação permite que sejam realizadas comparações entre múltiplas execuções de um programa, auxiliando os desenvolvedores para que haja uma melhor compreensão do código fonte. Os autores propuseram a comparação do tempo de execução de um programa com diferentes entradas. Isto permite melhor projeto dos testes, análise do tempo de resposta da aplicação, identificação de duplicidades nos casos de testes e auxílio na depuração (ZHAO et al., 2009).

Foram utilizados diagramas de sequência para realizar as demonstrações das execuções em 2D. Cada objeto é representado por uma pequena esfera sólida, podendo estar relacionado a outros objetos dependendo do nível de abstração utilizado. Chamadas de métodos foram representadas por bordas dispostas de acordo com a execução dos eventos. O mapeamento entre as execuções foi realçado com linhas amarelas (ZHAO et al., 2009). A Figura 14 demonstra um diagrama de sequência gerado pela aplicação.

Figura 14 – Representação gráfica através de diagramas de sequência

Fonte: Zhao et al. (2009)

Na exibição em 3D, são utilizados alvos visuais para capturar os pontos focais dos programadores, conforme demonstrado na Figura 15 nos trechos tracejados em branco.

Figura 15 – Análise dos pontos do diagrama focados pelo usuário durante a avaliação

Fonte: Zhao et al. (2009)

Os resultados obtidos em avaliações iniciais indicam que a abordagem auxilia na análise de várias execuções de uma aplicação. A ferramenta demonstra como ocorre a interação entre os objetos em tempo de execução e permite que seja analisado em qual ponto do diagrama os programadores estão focando a busca por defeitos (ZHAO et al., 2009).

3.5.2 Mapeamento tridimensional dos elementos do programa

Crossno, Rogers e Garasi (2002) desenvolveram uma ferramenta visual com o objetivo de auxiliar os desenvolvedores na atividade de depuração. Ela transforma grandes quantidades de números em formas e imagens tridimensionais. Os desenvolvedores podem utilizá-las na interpretação de dados e identificação de padrões que levem à descoberta de trechos defeituosos.

Cada elemento é um hexaedro formado por oito nós que designam as curvas deste elemento e as mudanças ocorridas em intervalos regulares de tempo exemplificando alterações de estado dos objetos. A ferramenta primeiramente divide o modelo de simulação, em seguida os processadores verificam cada uma das variáveis em cada instante de tempo e os objetos com valores diferentes do conteúdo esperado são associados. Por último, os objetos são exibidos de modo que os usuários possam identificar trechos incorretos e estabelecer relações de onde eles foram originados (CROSSNO; ROGERS; GARASI, 2002). A Figura 16 demonstra a visualização proposta pela ferramenta.

Figura 16 – Representação tridimensional do código inspecionado

Fonte: Crossno, Rogers e Garasi (2002)

3.6 Resultados

Esta seção dedica-se a apresentação dos resultados obtidos nesta revisão da literatura; ela é baseada nas contribuições das informações para a atividade de depuração.

3.6.1 Avaliações de usabilidade

Foi possível constatar que dos vinte artigos selecionados dezoito deles apresentaram algum tipo de ferramenta que oferece mecanismos visuais para auxílio à localização de defeitos; entretanto apenas as ferramentas Tarantula (JONES; HARROLD; STASKO, 2002) e GZoltar (GOLVEIA; CAMPOS; ABREU, 2013) apresentaram ferramentas visuais que utilizam técnicas LDC.

Somente os trabalhos de Jones, Harrold e Stasko (2002) e Perez e Abreu (2013), apresentaram resultados de avaliações com desenvolvedores. Enquanto os demais focaram somente nas características da aplicação sem avaliações de sua usabilidade. Tal característica sustenta o que foi mencionado na introdução deste trabalho, que além da ausência de

propostas que abordem o uso de ferramentas visuais durante a atividade de depuração, faltam experimentos com usuários que avaliem a usabilidade, a eficiência e a eficácia das ferramentas abordadas. A Figura 17 demonstra os resultados obtidos.

Figura 17 – Avaliação da usabilidade das ferramentas

Fonte: Fábio Pereira da Silva, 2017

3.6.2 Análise do período dos artigos selecionados

Foram considerados no desenvolvimento desta revisão da literatura artigos desenvolvidos a partir do ano de 1990. Foi possível verificar que houve um aumento significativo de pesquisas sobre a atividade de depuração nos últimos cinco anos. Dos vinte artigos selecionados, 60% deles foram publicados entre 2010 e 2015, 25% entre 2000 e 2009 e 15% entre 1990 e 1999, o que demonstra o crescimento recente de pesquisas sobre o desenvolvimento de ferramentas visuais.

3.6.3 Outras considerações importantes

A ferramenta Tarantula (JONES; HARROLD; STASKO, 2002) devido a avaliação de usabilidade realizada em sua representação visual foi mencionada como trabalho correlato em vários dos artigos incluídos nesta revisão.

A ferramenta GZoltar (PEREZ; ABREU, 2013), foi avaliada em um experimento com o objetivo de comprovar a sua escalabilidade para uso em ambientes reais de desenvolvimento e apresentou resultados promissores.

3.7 *Discussão*

Nesta seção, serão discutidos os resultados da revisão da literatura em relação a cada uma das questões de pesquisa descritas na Seção 3.2.

3.7.1 Ferramentas visuais propostas

Ao longo dos últimos anos foram apresentadas algumas aplicações com o objetivo de auxiliar os desenvolvedores na atividade de depuração, em especial com o desenvolvimento de ferramentas bidimensionais como a Tarantula, GZoltar, Code Bubbles, Bug Maps e In*Bug, encontradas nesta revisão da literatura.

Foram localizadas duas ferramentas que exibem os elementos mais suspeitos através de representações tridimensionais apresentadas nos trabalhos de Zhao et al. (2009) e Crossno, Rogers e Garasi (2002).

3.7.2 Avaliação em ambientes reais

Somente os artigos de Jones, Harrold e Stasko (2002) e Perez e Abreu (2013) conduziram experimentos com ferramentas de visualização para avaliar a usabilidade, eficiência e eficácia das aplicações apresentadas.

A ferramenta Tarantula (JONES; HARROLD; STASKO, 2002) foi avaliada quanto a sua capacidade de colorir os elementos que possuem maior probabilidade de apresentar

defeitos. Embora tenha apresentado resultados promissores, o experimento foi bastante restrito a avaliação da metáfora visual proposta pela ferramenta.

A avaliação da ferramenta GZoltar (PEREZ; ABREU, 2013) buscou realizar um estudo sobre o uso de ferramentas visuais em relação aos depuradores simbólicos. Dentre os participantes que utilizaram a ferramenta, todos eles conseguiram realizar a localização do defeito no tempo previsto. Considerando os desenvolvedores que não utilizaram a GZoltar, a grande maioria não chegou nem mesmo próximo ao defeito e apenas 35% dos desenvolvedores conseguiram localizá-lo.

Entretanto, a avaliação também foi bastante restrita tanto em número de defeitos avaliados (apenas um) e na comparação com depuradores simbólicos, tornando necessário o desenvolvimento de novos estudos.

As outras ferramentas propostas não apresentaram avaliações que comprovem a sua utilidade, focando somente nas funcionalidades contidas na aplicação.

3.7.3 Uso de técnicas de Interação Humano Computador

O trabalho de Lawrance e Bogart (2013), avalia como os desenvolvedores realizam a atividade de depuração usando modelo caça-caçador.

No estudo, foi identificado primeiramente como os desenvolvedores avaliam as hipóteses encontradas durante a depuração. Vídeos foram gravados com os participantes para entender como as suas ações e expressões faciais estão relacionadas com o surgimento de novas hipóteses. O estudo apresentado serve de subsídio para a identificação de fatores comportamentais, observados nos programadores, úteis para o *design* de novas ferramentas.

O artigo de Tanimoto (2013), apresenta como objetivo o uso da técnica de *Liveness* na construção de novas ferramentas visuais. A técnica proposta serve como subsídio para o entendimento de como os programadores constroem o seu modelo mental durante a depuração.

Considerando os estudos apresentados nos artigos de Jones, Harrold e Stasko (2002) e Perez e Abreu (2013), somente 20% de todos os trabalhos levaram em consideração a avaliação das experiências dos usuários durante a construção e a avaliação de ferramenta visuais ou apresentaram propostas que auxiliem o entendimento de como os programadores realizam a atividade de depuração.

3.7.4 Uso de ferramentas de depuração na prática

O aumento significativo de pesquisas sobre a atividade depuração nos últimos cinco anos indica que se trata de um tema relevante. Entretanto, poucas delas são validadas com desenvolvedores. A razão, em parte, está na dificuldade de uso em ambientes industriais, seja por possuírem um baixo grau de usabilidade, seja a menor eficácia em relação aos depuradores simbólicos, seja pelo grande consumo de tempo em sua utilização.

Com exceção dos trabalhos de Jones, Harrold e Stasko (2002) e Perez e Abreu (2013), nenhuma das ferramentas propostas apresentou avaliações com desenvolvedores, o que reforça a necessidade de avaliações que comprovem sua capacidade de auxiliá-los na atividade de depuração.

3.8 *Considerações finais*

Este capítulo apresentou uma revisão da literatura sobre ferramentas visuais, bidimensionais e tridimensionais, para apoio à depuração de software. A principal contribuição é a constatação de que, apesar do aumento considerável de estudos sobre o tema nos últimos anos, grande parte das ferramentas acabam nunca sendo utilizadas em ambientes reais. Tal situação ocorre pela falta de avaliações com desenvolvedores que comprovem a sua escalabilidade para uso em situações reais de desenvolvimento.

Este projeto de pesquisa tem por objetivo preencher essa lacuna ao propor a realização de experimentos com desenvolvedores utilizando ferramentas de depuração visuais e não-visuais. No próximo capítulo, são apresentadas as ferramentas avaliadas neste estudo.

4 Ferramentas Avaliadas

Neste capítulo, são apresentadas as principais características das ferramentas Jaguar (RIBEIRO, 2016) e CodeForest (MUTTI, 2014) que foram avaliadas neste projeto de pesquisa. Primeiramente, será introduzida a ferramenta Jaguar, utilizada como ferramenta padrão (*baseline*), por ser textual. Em seguida, será detalhada a ferramenta CodeForest. As ferramentas foram desenvolvidas pelo grupo de pesquisa em Engenharia de Software Experimental da Escola de Artes, Ciências e Humanidades da Universidade de São Paulo *Software Analysis and Experimental Group (SAEG)*.

4.1 Jaguar

A Jaguar (JAVa coveraGe faUlt locAlization Ranking) proposta por Ribeiro (2016) é uma ferramenta que tem como objetivo auxiliar os desenvolvedores na atividade de depuração através do uso de técnicas LDC baseadas em cobertura de fluxo de controle e de dados. A partir dos resultados obtidos, é apresentada ao desenvolvedor uma lista em ordem decrescente dos elementos mais suspeitos de um programa.

A ferramenta é um *plugin* integrável com a IDE Eclipse. A Jaguar fornece serviços que permitem que programas em execução na JVM (*Java Virtual Machine*) possam ser instrumentados. Instrumentar um programa consiste em inserir código adicional permitindo que sejam coletadas as informações da cobertura da aplicação durante a sua execução.

A Jaguar coleta informações dos testes realizados e gera um arquivo XML com o valor de suspeição de cada linha do programa coberta pelos testes. Ela coleta informações da cobertura do código invocando testes de unidade do programa e recolhe estas informações para cada elemento avaliado. A suspeição de cada linha pode ser definida por dez heurísticas diferentes (DRT, Jaccard, Kulczynski2, McCon, Minus, Ochiai, Op, Tarantula, Wong3, Zoltar).

O usuário tem a opção de selecionar o diretório de qualquer projeto Java que possua testes automatizados com o JUnit¹. Ela disponibiliza como opções para o desenvolvedor a seleção do tipo de cobertura de código que será utilizada que pode ser de fluxo de dados ou fluxo de controle. Com base no conteúdo selecionado, a ferramenta se encarrega de realizar a avaliação do código da aplicação. A Figura 18 demonstra a sua arquitetura.

¹ <http://www.junit.org>

Figura 18 – Arquitetura da ferramenta Jaguar

Fonte: Ribeiro (2016)

A Jaguar utiliza as ferramentas JaCoCo² e BA-DUA (ARAÚJO, 2014) para a implementação das técnicas SFL. A JaCoCo é uma ferramenta de cobertura de código aberto para a plataforma Java capaz de determinar a cobertura de linhas, métodos, classes e outras informações de um programa. A BA-DUA (ARAÚJO, 2014) é uma ferramenta de cobertura de fluxo de dados. A JaCoCoPlusBadua (RIBEIRO, 2016) é um módulo que se comunica com a JaCoCo e a BA-DUA para coleta das coberturas de fluxo de controle e de dados.

Como demonstrado na Figura 18, a Jaguar recebe um objeto com todos os dados de cobertura dos casos de teste bem como a informação se um dado caso de teste falhou ou foi bem sucedido. Com essas informações, uma das dez heurísticas disponíveis é utilizada para atribuir valores de suspeição para cada componente (nós ou blocos, ramos, associações definição-uso ou métodos) coberto durante os testes. Esses componentes são então mapeados para linhas a serem inspecionadas pelo desenvolvedor.

Os resultados obtidos são por último apresentados com o objetivo de facilitar a localização de defeitos. Os valores de suspeição são salvos em um arquivo XML para

² <http://www.eclemma.org/jacoco/>

cada heurística, permitindo que outras aplicações possam utilizar os dados gerados pela ferramenta.

Após a geração do relatório o desenvolvedor tem a opção de utilizar a JaguarView que apresenta as informações obtidas com base nos valores calculados pela ferramenta, conforme demonstrado na Figura 19.

Figura 19 – Informações para inspeção apresentadas pela Jaguar

Fonte: Ribeiro (2016)

4.2 CodeForest

A CodeForest proposta por Mutti (2014) tem como objetivo auxiliar os desenvolvedores na atividade de depuração através de representações visuais tridimensionais em uma floresta de cactus que demonstra os elementos mais suspeitos de um programa.

A CodeForest recebe listas de classes, métodos e blocos suspeitos (conjunto de instruções executadas em sequência) para construir uma floresta de cactus tridimensional representando o programa inspecionado. Na CodeForest, as classes são representadas como cactus, os métodos como galhos e as linhas do programa avaliado são representadas como espinhos de um galho. Os elementos mais suspeitos da floresta são pintados de vermelho.

Quanto mais um elemento (classe, método ou declaração) for executado por um caso de teste que falhou mais suspeito ele é.

O posicionamento, a coloração e a espessura de cada cactu da floresta são determinados com base em seu valor de suspeição. Os cactus são posicionados de maneira ordenada da esquerda para a direita da base onde encontram-se os elementos mais suspeitos do programa.

A Figura 20 ilustra o processo descrito. Primeiramente, o cactu com menor valor pintado de verde está posicionado no canto superior esquerdo. Outro com probabilidade média está localizado no centro. O cactu com maior probabilidade de apresentar defeitos está posicionado no canto inferior à direita da base.

Figura 20 – Posicionamento dos cactus na CodeForest

Fonte: Mutti (2014)

Os elementos da floresta são coloridos de acordo com o seu valor de suspeição, considerados no intervalo entre 0 e 1. A cor verde representa a menor probabilidade dele apresentar defeitos. A cor amarela representa um estado intermediário. A cor laranja indica uma alta probabilidade. Por último, a cor vermelha representa os elementos com maior valor de suspeição.

A CodeForest funciona de maneira integrada e totalmente desacoplada da Jaguar possibilitando que ela seja apenas responsável pela geração da floresta de cactus com base nas informações dos testes. Primeiramente, a CodeForest realiza a interpretação dos arquivos gerados pela Jaguar. Os relatórios possuem as informações do valor de suspeição de cada elemento do programa avaliado, que representa a sua probabilidade de apresentar defeitos.

As informações do arquivo são armazenadas e recuperadas no momento em que for solicitada a visualização com a ferramenta. Quando ela for requisitada o arquivo é interpretado e suas informações são apresentadas através da metáfora visual. A seguir, são detalhados todos os passos necessários para a utilização da CodeForest, detalhados no Apêndice C.

- O desenvolvedor cria casos de testes da aplicação.
- Os testes são automatizados com *framework* JUnit.
- A Jaguar é invocada para gerar as informações de suspeição das classes, métodos e linhas com o uso de técnicas SFL.
- A CodeForest é invocada.
- O desenvolvedor seleciona o relatório da heurística desejada e solicita a geração da floresta de cactus.
- Por último, a floresta é apresentada de acordo com o arquivo selecionado.

A CodeForest permite que o desenvolvedor aumente ou diminua o valor de suspeição que ele deseja avaliar. O filtro pode ser feito somente por métodos que possuam um valor de suspeição alto com grande probabilidade de conterem defeitos. A Figura 21 demonstra a visualização da metáfora criada pela ferramenta.

Figura 21 – Representação visual da CodeForest

Fonte: Fábio Pereira da Silva, 2017

O desenvolvedor tem a opção de interagir clicando no galho da metáfora. Ao clicar a janela, será exibida, como na Figura 22, o código fonte, com todas as suas linhas coloridas de acordo com o valor de suspeição calculado para cada uma delas com a floresta de cactus minimizada.

Figura 22 – Interação com o código do programa

Fonte: Fábio Pereira da Silva, 2017

Este projeto de pesquisa incluiu a implementação de inúmeras melhorias na metáfora da ferramenta CodeForest com o objetivo de facilitar a sua usabilidade para a condução dos experimentos. Dentre as melhorias implementadas estão a integração com a ferramenta Jaguar e melhorias na representação da floresta.

Na integração com a Jaguar, a CodeForest se tornou capaz de interpretar todos os arquivos XMLs com as informações de suspeição dos elementos do programa para cada heurística gerados pela Jaguar sem a necessidade de qualquer configuração manual. Foi melhorado o posicionamento dos elementos na floresta de cactus. Estão disponíveis opções em que o desenvolvedor pode aumentar ou diminuir o número de cactus (classes) que ele deseja que seja apresentado na floresta de acordo com a sua necessidade e também aumentar ou diminuir o valor de suspeição para a apresentação dos elementos na floresta de cactus, facilitando o entendimento das funcionalidades da ferramenta.

4.3 Considerações finais

Neste capítulo, foram apresentadas as principais características das ferramentas Jaguar e CodeForest. Foram detalhados os aspectos fundamentais adotados em seu funcionamento e como os resultados dos testes são utilizados para a exibição de informações de depuração. No próximo capítulo, serão apresentados os detalhes da avaliação experimental conduzida com as ferramentas.

5 Planejamento do Experimento

Para atingir os objetivos deste trabalho definidos na Seção 1.4 foram realizados experimentos com as ferramentas CodeForest e Jaguar visando avaliar a eficácia, a eficiência e a usabilidade de cada ferramenta para a localização de defeitos. Também foi avaliado o potencial de uso destas ferramentas com o *Technology Acceptance Model (TAM)*. O desenho experimental aplicado ao trabalho e aprovado no Comitê de Ética da Universidade de São Paulo é descrito neste capítulo.

5.1 Questões de pesquisa

Este trabalho teve como objetivo responder às seguintes questões de pesquisa:

1. Ferramentas visuais melhoram a eficácia e a eficiência da atividade de depuração em comparação com ferramentas baseadas apenas em informações textuais?

Conduziram-se experimentos com as ferramentas CodeForest e Jaguar para avaliar a eficácia e eficiência na localização de defeitos. Por eficácia, entende-se a capacidade da ferramenta direcionar o desenvolvedor até o sítio do defeito e por eficiência em avaliar se com o uso da ferramenta houve diminuição no tempo gasto até encontrá-lo.

2. Ferramentas visuais melhoram a usabilidade da atividade de depuração em comparação com ferramentas baseadas apenas em informações textuais?

A segunda questão de pesquisa avalia a usabilidade das ferramentas estudadas. Para isso elaborou-se um questionário com base no TAM visando identificar fatores que levam os desenvolvedores a utilizar as ferramentas apresentadas e qual delas possui o maior grau de usabilidade.

As variáveis de interesse dependentes deste estudo são:

- Eficácia, medida pela verificação se o defeito foi localizado ou não com o uso de uma das ferramentas.
- Eficiência, medida pelo tempo gasto pelos desenvolvedores até a localização do defeito.
- Usabilidade das ferramentas avaliadas, medida por meio do TAM.

Foram consideradas como variáveis independentes neste estudo:

- Participantes que realizaram os experimentos.
- Nível de experiência com a linguagem Java, a IDE Eclipse e com o *framework* JUnit.
- Defeitos e programas selecionados.

As ferramentas CodeForest e Jaguar foram escolhidas por elas ainda não terem sido introduzidas em ambientes industriais, evitando que os resultados dos experimentos fossem afetados pelos conhecimentos prévios dos participantes. Ambas utilizam a técnica localização de defeitos baseada em cobertura (LDC), para a construção de suas representações. Além disso, as ferramentas foram desenvolvidas pelo *Software Analysis & Experimentation Group (SAEG)*, grupo de pesquisa de Engenharia de Software da Escola de Artes, Ciências e Humanidades da Universidade de São Paulo.

5.2 Ambiente do Experimento

Esta seção apresenta o ambiente do experimento, definido após a submissão e a aprovação do projeto Comitê de Ética e Pesquisa da Universidade de São Paulo, detalhadas no Anexo A e no Anexo B.

5.2.1 Participantes

Durante o segundo semestre de 2016, realizou-se contatos com empresas para a busca de profissionais da área de desenvolvimento para o projeto de pesquisa. Entretanto, diante da dificuldade de encontrar profissionais voluntários foram realizados contatos com a *Faculdade de Tecnologia da Zona Leste (Fatec)* que possibilitou que a atividade fosse realizada na instituição como um curso presencial de curta duração sobre “Depuração Baseada em Cobertura de Código”.

Optou-se por realizar o estudo na Fatec por ela possuir em sua grade curricular uma disciplina de Teste de Software em que os estudantes adquirem familiaridade com o *framework* JUnit. O curso é totalmente focado no ensino da linguagem Java em todas as disciplinas de programação.

Os experimentos foram realizados com 119 estudantes matriculados a partir do 4º semestre do curso de Análise e Desenvolvimento de Sistemas. Com base na grade

curricular, nesta etapa eles já deveriam ter os conhecimentos fundamentais necessários para a atividade.

A definição do número de participantes ocorreu com base no simulador *Sample Size Calculator* (STATISTICS, 2016; UX, 2014). A avaliação proposta possui similaridades com a psicologia e usabilidade pois considerou o comportamento dos desenvolvedores durante a depuração.

O *Al-Therapy Statistics* retornou um número de 27 participantes. Informou-se como parâmetros que seria realizado um teste t , com grupos de amostras que possuem o mesmo objetivo, efeito de 0,5 e significância estatística de 0,05.

O Blink UX retornou um número de 57 participantes. Foram fornecidos como parâmetros a utilização de quatro grupos para a realização dos experimentos, com o mesmo objetivo, o progresso não foi medido ao longo do tempo, a avaliação ocorreu em um único *design* e através de um estudo quantitativo.

Segundo Nielsen (2012), o número mínimo de participantes necessário para a realização de um estudo quantitativo é de 20 pessoas para a obtenção de uma amostra que permita a realização de testes estatísticos. Em uma avaliação similar de um *framework Open Source* Steinmacher et al. (2016) realizaram experimentos em duas iterações, a primeira com 14 participantes e a segunda com 51 pessoas.

Com base nas ideias propostas, conseguimos alcançar um total de 119 pessoas para a realização dos experimentos. Uma quantidade maior do que os valores estimados pelo *Al-Therapy Statistics*, Blink UX e do valor mínimo proposto por Nielsen.

5.2.2 Preparo do ambiente

Para a realização dos experimentos criou-se um ambiente nos laboratórios de informática da instituição que incluiu todos os recursos necessários para o estudo, conforme detalhado a seguir.

- IDE Eclipse.
- *Plugin* da ferramenta Jaguar.
- *Plugin* da ferramenta CodeForest.
- Dois programas, cada um deles com um defeito.
- Material de treinamento sobre cada uma das aplicações.

- Instruções para a realização do experimento.
- Questionário a ser respondido após o término da atividade.
- Organização dos “plugins” das ferramentas e programas em grupos para garantir a aleatoriedade na distribuição dos participantes.

Ao final da avaliação, os participantes responderam um questionário em que se buscou entender aspectos relacionados a sua experiência profissional e usabilidade de cada ferramenta. O material de treinamento sobre as ferramentas CodeForest e Jaguar está disponível no Apêndice C e o questionário aplicado pode ser consultado no Apêndice D.

5.2.3 Programas e defeitos selecionados

Optou-se pela escolha de dois programas de código aberto desenvolvidos em Java: JSoap e XStream. JSoap¹ é uma biblioteca Java para análise, manipulação e extração de dados HTML. XStream² é uma biblioteca para a serialização de objetos XML. Os defeitos escolhidos possuem grau de dificuldade médio, sem qualquer relação a eventuais regras aplicáveis aos programas selecionados e que possam ser encontrados com ou sem o uso das ferramentas CodeForest e Jaguar.

O defeito no programa JSoup é um defeito real encontrado no código fonte do programa disponível no repositório. Trata-se da comparação de dois elementos utilizando *equals()* ao invés de “==”. O defeito encontra-se na linha 454 da classe *Element*, no método *indexInList()*.

O defeito no programa XStream foi criado no experimento realizado com a ferramenta GZoltar (GOLVEIA; CAMPOS; ABREU, 2013). Trata-se de uma comparação com o operador lógico errado em uma estrutura de decisão. Ao invés de utilizar “*targetType==null*” deveria ser utilizado “*targetType!=null*”. O defeito semeado está localizado na linha 574 da classe *AnnotationMapper*, no método *cacheConverter()*.

Durante o treinamento os estudantes utilizaram o programa JTopas³, um analisador de arquivos de textos que verifica a estrutura de arquivos nos formatos HTML e XML. Foram criados dois defeitos no programa. No primeiro defeito havia um incremento e um decremento que não deveriam estar presentes nos parâmetros passados na instância de

¹ <https://github.com/jhy/jsoup>

² <https://github.com/x-stream/xstream>

³ <https://sourceforge.net/projects/jtopas/>

uma classe. No segundo defeito um atributo estava recebendo uma posição de um array inválida. Os defeitos estavam semeados nas linhas 1559 e 1562 da classe *AbstractTokenzer*, no método *isKeyword()*.

Como os participantes não tiveram contato com os programas selecionados antes da atividade conhecimentos prévios dos estudantes não afetaram os resultados do estudo.

5.3 Divisão dos experimentos

O Experimento 1 foi realizado no dia 20/04/2017 com 76 estudantes da disciplina de Engenharia de Software III, do 4º semestre do curso nos períodos vespertino e noturno. Trata-se de uma disciplina com vários estudantes do penúltimo e último semestre do curso em que grande parte dos alunos já haviam cursado a disciplina de Teste de Software, com conhecimentos em conceitos básicos sobre depuração e no *framework* JUnit. Pela grande quantidade de estudantes matriculados ela possibilitou que o estudo fosse conduzido com uma amostra significativa de participantes.

O Experimento 2 foi realizado nos dias 06/05/2017 e 12/05/2017 com 34 alunos da disciplina de Teste de Software, do 5º semestre do curso nos períodos vespertino e noturno. Esta disciplina possibilitou que o estudo fosse conduzido em um componente curricular totalmente ligado aos conceitos explicados no treinamento sobre o uso das ferramentas. Nela os estudantes adquirem contato com conceitos fundamentais de teste de software, depuração e no *framework* JUnit, facilitando assim o entendimento das ferramentas.

O Experimento 3 foi realizado no dia 23/05/2017 com 9 alunos das disciplinas de Inglês VI e Ética do último semestre no período noturno, devido a ausência de disponibilidade em disciplinas mais aderentes. O estudo ocorreu com uma amostra reduzida de alunos; porém trouxe resultados muito importantes para a pesquisa, considerando que todos os 9 estudantes já tinham alguma experiência profissional na área de desenvolvimento.

5.4 Procedimentos

Conduziram-se dois experimentos pilotos com 3 estudantes em cada um deles nos meses de março a abril de 2017 com o objetivo de validar o desenho experimental e mitigar possíveis riscos envolvidos.

Após os experimentos pilotos, realizaram-se contatos com professores da Fatec Zona Leste para verificar a disponibilidade para que fosse dedicada uma aula de cada disciplina durante o 1º semestre de 2017 para o estudo. Em seguida com a confirmação dos professores de disciplinas do 4º, 5º e 6º semestres do curso e agendamento das datas do experimentos, os estudantes das disciplinas foram convidados via mensagem eletrônica.

A atividade teve duração de 4 horas e incluiu uma apresentação sobre “Depuração Baseada em Cobertura de Código”, treinamento sobre o uso de cada ferramenta, leitura do termo de consentimento, um intervalo de 10 minutos e por último a execução das tarefas. Os estudantes realizaram duas tarefas durante a depuração utilizando as ferramentas CodeForest e Jaguar. A Jaguar foi utilizada em duas versões, a lista de linhas e a lista de métodos mais suspeitos. O termo de consentimento pode ser visualizado no Apêndice A e o conteúdo utilizado na apresentação pode ser consultado no Apêndice B.

Em nenhum momento ocorreu a identificação dos alunos durante o experimento. A partir da leitura do termo de consentimento eles receberam um identificador alfanumérico. Com o objetivo de garantir a aleatoriedade na distribuição dos participantes, os estudantes foram divididos em grupos conforme pode ser visualizado na Tabela 6.

Tabela 6 – Distribuição dos participantes

ID	Jaguar Método	Jaguar Linha	CodeForest
1	1XStream		2JSsoup
2		2XStream	1JSoup
3		1JSoup	2XStream
4	2JSoup		1XStream
5		1XStream	2JSoup
6	2XStream		1JSoup
7	1JSoup		2XStream
8		2JSoup	1XStream

Fonte: Fábio Pereira da Silva, 2017

Na Tabela 6, a coluna 1 se refere ao identificador (ID) do grupo e as colunas 2, 3 e 4 a ferramenta utilizada durante a sessão de depuração. Nas linhas de cada grupo estão a ordem de uso de cada ferramenta. O número preenchido antes do nome do programa corresponde à ordem de uso durante a depuração.

No grupo 1 da Tabela 6 os estudantes utilizaram primeiro durante a depuração o programa XStream com a Jaguar Método e em seguida o programa JSoup com a CodeForest. No grupo 2, os estudantes realizaram a depuração primeiro com o JSoup com a CodeForest e em seguida do XStream com a Jaguar Linha. Tal organização garante a aleatoriedade na distribuição dos participantes e mitiga a influência de fatores nos resultados do experimento como o cansaço, o programa usado com cada ferramenta e a ordem de uso das ferramentas durante a depuração.

Depois do treinamento, os estudantes tiveram 30 minutos para finalizar cada tarefa, que consistiu na localização de um defeito em cada programa seguindo a distribuição da Tabela 6. Além da ferramenta, outros recursos da IDE Eclipse puderam ser utilizados.

Ao término da atividade, os participantes responderam um questionário disponível em um formulário Web. A coleta dos dados sobre o uso das ferramentas ocorreu por meio de arquivos de “log” gerados durante a depuração para verificar se o participante conseguiu ou não realizar a localização do defeito.

No questionário, buscou-se obter informações como o tempo de experiência com a linguagem Java, IDE Eclipse e o *framework* JUnit. Os estudantes informaram se o defeito foi localizado ou não, qual era o defeito e em qual linha do programa ele se encontrava. O questionário incluiu questões relacionadas ao TAM para avaliar o potencial das tecnologias estudadas serem utilizadas em situações reais de desenvolvimento.

5.5 Análise de dados

Considerando o baixo número de participantes que conseguiram realizar a localização do defeito, conforme será detalhado no Capítulo 6, a análise dos resultados do experimento não pôde ser baseada apenas na quantidade de participantes que localizaram e corrigiram o defeito. Sendo assim, a análise dos dados ocorreu em três etapas, avaliação dos “logs”, avaliação das respostas do TAM e testes estatísticos.

Primeiramente implementou-se uma ferramenta de extração de informações para a avaliação de como os participantes realizaram a atividade por meio da análise dos “logs” gerados durante a tarefa. Em seguida, ocorreu uma análise estatística do questionário, visando avaliar por meio da percepção dos participantes qual ferramenta foi mais eficaz, eficiente e com o maior grau de usabilidade. Por último, realizaram-se testes estatísticos

não paramétricos com o objetivo de responder às questões de pesquisa. Nesta seção são descritos os métodos utilizados para a realização da análise de dados.

5.5.1 Análise dos arquivos *log*

As ferramentas CodeForest e Jaguar permitem o registro de todas as atividades realizadas pelos participantes durante a depuração por meio de arquivos de “log”. Com base nos “logs” obtidos foram selecionadas 42 questões para avaliação, sendo divididas em nove grupos detalhados na Tabela 7.

Tabela 7 – Divisão dos grupos de questões avaliadas

Nome do Grupo	Finalidade
Uso das ferramentas durante a atividade	Verificou-se, dentre outras questões, o tempo que os participantes gastaram para a execução de toda a atividade e por quanto tempo a ferramenta foi usada durante a tarefa.
Interrupções no uso das ferramentas	Foi avaliado o tempo total de períodos de inatividade durante a tarefa ou ausência de interações com a ferramenta superiores a 3 minutos e quantas vezes estas situações ocorreram para cada participante. A cada 3 minutos de inatividade foi considerado um “gap” no uso da ferramenta.
Interações com a ferramenta	Foi avaliado quantas vezes as representações propostas pela CodeForest e Jaguar foram utilizadas. Buscou-se entender a frequência de uso de cada uma das funcionalidades das ferramentas pelos participantes.
Linhas inspecionadas	Ocorreu a identificação de quantas linhas diferentes o participante interagiu durante a tarefa e o número de interações que ocorreram com cada linha.
Interações com depuradores e o JUnit	Foram verificados quantos cliques ocorreram entre a ferramenta, os testes de unidade (JUnit) e depurador e quantas vezes os testes de unidade ou o depurador foram utilizados durante a atividade.

Inclusão de “breakpoints”	Foi avaliado quantos “breakpoints” foram incluídos durante o uso da ferramenta, após uma interação em um dado trecho do programa ou em qualquer método a partir de um clique na ferramenta.
Inclusão de “breakpoints” no trecho do defeito	Buscou-se entender se em algum momento durante a depuração o participante inseriu “breakpoints” no método ou linha do defeito.
Valores de suspeição	Avaliaram-se quantos valores de suspeição (“scores”) diferentes o participante interagiu na atividade e quantas linhas de cada uma das cores propostas pelas ferramentas o desenvolvedor interagiu durante a tarefa.
Interações com o trecho do defeito	Foram avaliadas, dentre outras questões, quantas vezes o participante interagiu com a classe, método ou linha do defeito e quanto tempo durante a atividade foi necessário para que a primeira interação com o trecho do defeito ocorresse. Ainda verificou-se no caso do programa JSoup, se o método ou linha que realizou a chamada ao método do defeito foi clicado pelos participantes.

Fonte: Fábio Pereira da Silva, 2017

A ferramenta de extração de informações gerou um relatório com os resultados obtidos para cada participante para as 42 questões avaliadas. Com base neste relatório e na análise dos arquivos de “log” identificaram-se situações que não seriam possíveis de serem visualizadas apenas com a análise individual dos 238 “logs” gerados pelos participantes durante a depuração.

A Tabela 8 demonstra todas as questões avaliadas e o grupo na qual elas pertencem.

Tabela 8 – Questões avaliadas

Questão	Grupo
Quanto tempo foi gasto na tarefa?	Uso das ferramentas durante a atividade

Quanto tempo foi gasto realizando interações com a ferramenta?	Uso das ferramentas durante a atividade
Qual o percentual de participantes que iniciou a tarefa utilizando a ferramenta?	Uso das ferramentas durante a atividade
Qual foi o tempo total de “gaps” em média para todas as interações durante a atividade? (Usando ou não a ferramenta)	Interrupções no uso das ferramentas durante a atividade
Qual foi a quantidade total de “gaps” para todas as interações durante a atividade? (Usando ou não a ferramenta)	Interrupções no uso das ferramentas durante a atividade
Qual foi o tempo total de “gaps” para todas as interações utilizando a ferramenta?	Interrupções no uso das ferramentas durante a atividade
Qual foi a quantidade total de “gaps” para todas as interações utilizando a ferramenta?	Interrupções no uso das ferramentas durante a atividade
Qual foi o percentual de participantes que utilizaram a ferramenta durante a tarefa?	Interações com a ferramenta
Quantas vezes a representação da ferramenta foi usada?	Interações com a ferramenta
Quantas vezes a opção de “zoom” da CodeForest ou o “slider” da Jaguar foram utilizados?	Interações com a ferramenta
Quantas vezes o filtro de busca textual foi utilizado?	Interações com a ferramenta
Quantas vezes ocorreram cliques em classes durante o uso da CodeForest?	Interações com a ferramenta
Quantas vezes ocorreram cliques em métodos durante o uso da CodeForest?	Interações com a ferramenta
Quantas vezes ocorreram cliques em linhas durante o uso da CodeForest?	Interações com a ferramenta
Quantas interações ocorreram entre a Jaguar e o editor?	Interações com a ferramenta
Quantas linhas foram inspecionadas durante o uso da ferramenta?	Linhas inspecionadas
Quantas vezes cada classe, método e linha foi inspecionada durante o uso da ferramenta?	Linhas inspecionadas

Quantas linhas foram inspecionadas através do editor com a Jaguar?	Linhas inspecionadas
Quantos cliques ocorreram a partir de uma interação com a ferramenta entre o JUnit ou depurador?	Interações com depuradores e o JUnit
Quantas vezes o JUnit foi utilizado na tarefa?	Interações com depuradores e o JUnit
Quantas vezes o depurador foi utilizado na tarefa?	Interações com depuradores e o JUnit
Quantos “breakpoints” foram adicionados após a inspeção com a ferramenta? (No mesmo método ou linha clicado)	Inclusão de “breakpoints”
Quantos “breakpoints” foram adicionados nos métodos/linhas da ferramenta? (Considerando todos os métodos ou linhas do programa)	Inclusão de “breakpoints”
Quantos “breakpoints” foram adicionados na tarefa? (Considerando todos os breakpoints incluídos)	Inclusão de “breakpoints”
Quantos “breakpoints” foram adicionados no método do defeito?	Inclusão de “breakpoints” no método ou linha do defeito
Quantos “breakpoints” foram adicionados na linha do defeito?	Inclusão de “breakpoints” no método ou linha do defeito
Quanto tempo foi gasto para clicar no método do defeito?	Interações com o trecho do defeito
Quanto tempo foi gasto para clicar no linha do defeito?	Interações com o trecho do defeito
Quantas vezes o método do defeito foi inspecionado?	Interações com o trecho do defeito
Quantas vezes a linha do defeito foi inspecionada?	Interações com o trecho do defeito
Qual o percentual de participantes que encontraram o defeito imediatamente após um clique na ferramenta?	Interações com o trecho do defeito

Se o programa inspecionado foi o JSoup, quantas vezes o método mais suspeito que chama o método do defeito foi clicado?	Interações com o trecho do defeito
Se o programa inspecionado foi o JSoup, quantas vezes a linha do método mais suspeito que chama o método do defeito foi clicada?	Interações com o trecho do defeito
Quantas vezes a linha do defeito foi inspecionada através do editor com a Jaguar?	Interações com o trecho do defeito
Quantas vezes o método do defeito foi inspecionado através do editor com a Jaguar?	Interações com o trecho do defeito
Quanto valores de suspeição diferentes foram inspecionados?	Valores de suspeição
Quanto métodos/linhas foram inspecionados em sequência?	Valores de suspeição
Quanto métodos/linhas em verde foram inspecionados?	Valores de suspeição
Quanto métodos/linhas em amarelo foram inspecionados?	Valores de suspeição
Quanto métodos/linhas em laranja foram inspecionados?	Valores de suspeição
Quanto métodos/linhas em vermelho foram inspecionados?	Valores de suspeição
Qual o percentual de participantes que inspecionaram primeiro o método mais suspeito?	Valores de suspeição

Fonte: Fábio Pereira da Silva, 2017

5.5.2 Questões do TAM

Foram elaboradas 10 questões para avaliar o potencial de uso das tecnologias com base no modelo TAM apresentado na Seção 2.3.3. Considerou-se para a utilização do questionário os dois pilares do modelo, utilidade percebida e facilidade de uso percebida interligados pela intenção de uso futuro das tecnologias. As definições de cada grupo ocorreram com base nas ideias propostas por Davis, Bagozzi e Warshaw (1989).

Avaliou-se a utilidade percebida como o grau que a pessoa acredita que o uso das ferramentas CodeForest e Jaguar irá melhorar o seu desempenho durante a depuração. Selecionou-se para avaliação neste grupo as variáveis de usabilidade, produtividade, eficácia e utilidade da ferramenta.

A facilidade de uso percebida é definida como o grau que a pessoa acredita que o uso das ferramentas CodeForest e Jaguar não envolverá esforço em seu aprendizado. Considerou-se para este grupo as variáveis de facilidade de aprendizado, esforço mental, facilidade de manuseio e facilidade da tarefa.

A intenção de uso determina a intenção comportamental de uso das ferramentas CodeForest e Jaguar. Foram consideradas como variáveis de interesse neste grupo a consideração de uso e a intenção de uso futuro das ferramentas.

As questões aplicadas sobre a percepção de uso de cada ferramenta são apresentadas na Tabela 9, bem como o grupo do TAM que cada questão pertence. Tanto para a Jaguar quanto para a CodeForest utilizou-se o mesmo grupo de questões.

Tabela 9 – Questões do TAM

Questão	Grupo do TAM	Fator Influenciado
Supondo que eu tive acesso à ferramenta, eu pretendo usá-la	Intenção de uso	Intenção de uso futuro
Dado que eu tive acesso à ferramenta, eu imagino que a usaria	Intenção de uso	Consideração de uso
Usar a ferramenta melhorou o meu desempenho durante a tarefa de depuração	Utilidade	Usabilidade
Usar a ferramenta durante a tarefa aumentou a minha produtividade	Utilidade	Produtividade
Usar a ferramenta melhorou a minha eficácia durante a atividade de depuração	Utilidade	Eficácia
De modo geral, eu achei a ferramenta útil para realizar a atividade de depuração	Utilidade	Utilidade da ferramenta
Aprender a ferramenta foi fácil para mim	Facilidade de uso	Facilidade de aprendizado
Interagir com a ferramenta não exigiu muito do meu esforço mental	Facilidade de uso	Esforço mental

De modo geral, eu achei a ferramenta fácil de usar	Facilidade de uso	Facilidade de manuseio
Eu achei fácil contar com a ferramenta para realizar a tarefa de depuração	Facilidade de uso	Facilidade da tarefa

Fonte: Fábio Pereira da Silva, 2017

Adotou-se como padrão de respostas para o questionário a escala *Likert* amplamente utilizada por sua confiabilidade, validade e sensibilidade (CUMMINS; GULLONE, 2000), considerando valores de 1 a 7, em que quando assinalado o valor 1 o participante concordou totalmente com a afirmação e 7 discordou totalmente da afirmação.

5.5.3 Testes estatísticos

Com o objetivo de responder às questões de pesquisa foram realizados testes estatísticos para avaliar a eficácia, a eficiência e a usabilidade das ferramentas CodeForest e Jaguar por meio do software R⁴.

Para entender se os dados seguiam uma distribuição normal utilizou-se o teste de *Shapiro-Wilk* (SHAPIRO; WILK, 1965) devido a sua eficiência para diferentes distribuições e tamanhos de amostras se comparado aos resultados observados em outros testes.

Nos testes estatísticos, avaliaram-se as informações extraídas dos arquivos de “log” dos participantes. Para avaliar a eficácia e a eficiência das ferramentas CodeForest e Jaguar, foram considerados, respectivamente, a quantidade de interações com o método ou linha do defeito e o tempo gasto pelos participantes para localizar o método ou linha do defeito. Para a usabilidade optou-se por utilizar as respostas do TAM.

Ao aplicar o teste de Shapiro-Wilk e analisar a representação dos dados por meio de histogramas foi verificado que os dados não seguiam uma distribuição normal e assim para todos os outros testes estatísticos realizados adotou-se uma abordagem não-paramétrica.

Como os dados não seguiam uma distribuição normal, realizaram-se testes de *Wilcoxon*, uma abordagem não paramétrica para o teste t pareado. O teste de Wilcoxon verifica se as medianas das distribuições são iguais ou diferentes entre si. Ou seja, ele avalia a diferença entre cada valor da amostra e o valor hipotético da mediana (KAZMIER,

⁴ <https://www.r-project.org/>

2007). A aceitação ou a rejeição de uma determinada hipótese é determinada por meio do *p-value* (GREENLAND et al., 2016). Na avaliação conduzida optou-se pela utilização de um intervalo de confiança de 95%, conforme descrito na Tabela 10.

Tabela 10 – Teste de hipóteses

Valor de p	Interpretação
p maior do que 0,05	Não é possível rejeitar a hipótese nula (H_0)
p menor ou igual a 0,05	Nesta condição a hipótese nula foi rejeitada e aceita a hipótese alternativa (H_1)

Fonte: Fábio Pereira da Silva, 2017

Com o teste de Wilcoxon, pode-se identificar qual das ferramentas avaliadas neste projeto obteve melhores resultados em cada uma das questões de pesquisa. Entretanto, também é importante considerar o tamanho do efeito que os resultados obtidos representam sobre a amostra. O tamanho do efeito, em inglês *Effect Size*, avalia a magnitude da diferença entre os grupos (SULLIVAN; FEINN, 2012). O nível de significância estatística é afetado por diversas características; entretanto, o tamanho da amostra é a mais determinante (SNYDER; LAWSON, 1993)

Visando verificar o tamanho que o efeito dos resultados obtidos no teste de Wilcoxon representam sobre a amostra, utilizou-se o *Delta de Cliff*, uma medida que permite quantificar a magnitude da diferença entre grupos de amostras não paramétricas. O Delta de Cliff independe da média, pois considera propriedades ordinais de um conjunto de dados (CLIFF, 1993).

Seguindo as ideias propostas por Romano et al. (2006), os valores obtidos no Delta de Cliff podem variar entre -1 e 1. Valores iguais a 1 indicam que o resultado observado no grupo 1 é maior do que o resultado observado no grupo 2. Valores iguais a 0 indicam que os grupos estão completamente sobrepostos. Valores iguais a -1 indicam que o resultado observado no grupo 1 é menor do que o resultado do grupo 2.

A Tabela 11 apresenta com base nas ideias propostas por Romano et al. (2006) e Cohen (1988) a escala utilizada para a avaliação do tamanho do efeito.

Tabela 11 – Escala de avaliação do tamanho do efeito

Valor	Interpretação
Inferior a 0,147	Diferença negligenciável, ou seja, não há diferença entre os grupos
Maior ou igual do que 0,147 e inferior a 0,33	Diferença pequena entre os grupos
Maior ou igual a 0,33 e inferior a 0,474	Diferença média entre os grupos
Maior ou igual a 0,474	Diferença grande entre os grupos

Fonte: Fábio Pereira da Silva, 2017

Com o objetivo de avaliar a consistência e o grau de confiabilidade das respostas decorrentes do questionário para avaliação do TAM utilizou-se o coeficiente alpha de Cronbach (CRONBACH, 1951). Pode-se dizer que há confiabilidade no significado de um instrumento de medida se ele oferece sempre os mesmos resultados quando aplicado a alvos estruturalmente iguais (CRONBACH, 1951). O coeficiente alpha de Cronbach define o coeficiente médio de todas as estimativas de consistência de um instrumento (MAROCO; MARQUES, 2006).

A avaliação do TAM incluiu uma análise fatorial e cálculo da correlação com o objetivo de verificar as questões do TAM que estão relacionadas de acordo com as respostas dos participantes e como os diferentes grupos do TAM se relacionam entre si. A análise fatorial visa resumir a estrutura de correlações por meio da redução da dimensionalidade de um grande número de variáveis (SPEARMAN, 1904).

5.6 Considerações finais

Este capítulo apresentou os procedimentos realizados para a condução dos experimentos, questões abordadas neste projeto de pesquisa, variáveis de interesse e metodologia de análise de dados utilizada. Foram detalhadas a definição da instituição de ensino onde aconteceu o estudo, a preparação do ambiente, os programas selecionados, os materiais envolvidos no treinamento sobre as ferramentas, a divisão dos experimentos e a distribuição dos participantes.

Com base na metodologia de análise de dados, avaliaram-se 238 arquivos de “log” gerados pelos participantes durante a atividade e a percepção dos estudantes sobre a usabilidade das ferramentas para a localização de defeitos que serão descritos no próximo capítulo.

6 Resultados do Experimento

Neste capítulo serão apresentados os resultados obtidos durante a execução dos experimentos para que fossem respondidas as questões de pesquisa descritas no Capítulo 5. Primeiramente, serão apresentados os resultados em relação à análise dos “logs” para cada um dos grupos de questões.

Em seguida, serão apresentados os resultados dos testes estatísticos para a avaliação da eficácia, eficiência e usabilidade das ferramentas. Será detalhada a análise estatística feita no questionário para avaliar a percepção dos estudantes em relação à usabilidade das ferramentas CodeForest e Jaguar para a localização de defeitos. Os resultados são discutidos, indicando a experiência profissional dos estudantes, a distribuição dos participantes, quantas pessoas realizaram a localização da classe, do método e da linha do defeito e opiniões sobre as demais funcionalidades das ferramentas.

Por último, serão discutidos os resultados obtidos e respondidas as questões de pesquisa.

6.1 *Resultados dos arquivos de log*

Dado que apenas cinco estudantes conseguiram realizar a localização e a correção do defeito com base nas respostas do questionário aplicado ao término do experimento, não foi possível avaliar a capacidade das ferramentas direcionarem ou não o desenvolvedor à localização do defeito apenas por meio dos resultados do questionário. Para realizar essa avaliação foram utilizados os arquivos de “log” dos estudantes. As Tabelas 12 a 19 apresentam os resultados obtidos com relação às 42 questões da Tabela 8. A ausência de avaliação de alguns itens ocorreu devido as diferenças existentes entre os “logs” gerados pelas ferramentas.

Tabela 12 – Uso das ferramentas durante a atividade

Questão	Média CodeForest	Média Jaguar
Quanto tempo foi gasto na tarefa?	00:33:21	00:36:44
Quanto tempo foi gasto realizando interações com a ferramenta?	00:25:31	00:28:03
Qual o percentual de participantes que iniciou a tarefa utilizando a ferramenta?	85,22%	94,78%

Fonte: Fábio Pereira da Silva, 2017

Ao avaliar o uso das ferramentas durante a atividade, verificou-se que a Jaguar foi mais utilizada do que a CodeForest tanto considerando toda a tarefa quanto no uso da ferramenta. Enquanto com a Jaguar ocorreram interações por um período médio de 00:28:03, na CodeForest ocorreram interações em média por um período de 00:25:31.

Tabela 13 – Interrupções no uso das ferramentas durante a atividade

Questão	Média CodeForest	Média Jaguar
Qual foi o tempo total de “gaps” em média para todas as interações durante a atividade? (Usando ou não a ferramenta)	00:03:38	00:05:34
Qual foi a quantidade total de “gaps” para todas as interações durante a atividade? (Usando ou não a ferramenta)	0,53	0,67
Qual foi o tempo total de “gaps” para todas as interações utilizando a ferramenta?	00:10:27	00:22:44
Qual foi a quantidade total de “gaps” para todas as interações utilizando a ferramenta?	1,43	2,91

Fonte: Fábio Pereira da Silva, 2017

Considerando as interrupções no uso das ferramentas durante a atividade, a CodeForest apresentou em média 00:03:38 de inatividade menor do que a Jaguar que obteve em média 00:05:34. A CodeForest também apresentou menores períodos de inatividade superiores a 3 minutos em relação a Jaguar.

Tabela 14 – Interações com a ferramenta

Questão	Média CodeForest	Média Jaguar
Qual foi o percentual de participantes que utilizaram a ferramenta durante a tarefa?	98,26%	99,13%
Quantas vezes a representação da ferramenta foi usada?	42,42	36,88
Quantas vezes a opção de “zoom” da CodeForest ou o “slider” da Jaguar foram utilizados?	4,8	0,44
Quantas vezes o filtro de busca textual foi utilizado?	2,49	0,24
Quantas vezes ocorreram cliques em classes durante o uso da CodeForest?	6,17	Não avaliado
Quantas vezes ocorreram cliques em métodos durante o uso da CodeForest?	18,09	Não avaliado
Quantas vezes ocorreram cliques em linhas durante o uso da CodeForest?	18,15	Não avaliado
Quantas interações ocorreram entre a Jaguar e o editor?	Não avaliado	10,4

Fonte: Fábio Pereira da Silva, 2017

A CodeForest obteve maior número de interações do que a Jaguar. Enquanto na CodeForest ocorreram em média 42,42 interações, com a Jaguar a quantidade média de interações foi de 36,88.

Tabela 15 – Linhas inspecionadas

Questão	Média CodeForest	Média Jaguar
Quantas linhas foram inspecionadas durante o uso da ferramenta?	13,35	11,87
Quantas linhas foram inspecionadas através do editor com a Jaguar?	Não avaliado	152,74

Fonte: Fábio Pereira da Silva, 2017

A Jaguar levou os participantes a inspecionarem um número menor de linhas. Enquanto com a Jaguar os estudantes inspecionaram em média 11,87 linhas, com a CodeForest os participantes analisaram em média 13,35 linhas.

Tabela 16 – Interações com depuradores e o JUnit

Questão	Média CodeForest	Média Jaguar
Quantos cliques ocorreram a partir de uma interação com a ferramenta entre o JUnit ou depurador?	1,23	0,12
Quantas vezes o JUnit foi utilizado na tarefa? (Sem o uso de depuradores)	2,64	3,45
Quantas vezes o depurador foi utilizado na tarefa?	0,47	1,01

Fonte: Fábio Pereira da Silva, 2017

O JUnit foi utilizado em média 3,45 vezes com a Jaguar e 2,64 vezes com a CodeForest. Considerando a média da quantidade de interações com depuradores, a Jaguar foi superior obtendo um valor médio de 1,01 interação enquanto a CodeForest obteve 0,47.

Tabela 17 – Inclusão de breakpoints

Questão	Média CodeForest	Média Jaguar
Quantos “breakpoints” foram adicionados após a inspeção com a ferramenta? (No mesmo método ou linha clicado)	0,11	0,38
Quantos “breakpoints” foram adicionados nos métodos/linhas da ferramenta? (Considerando todos os métodos ou linhas)	0,55	0,54
Quantos “breakpoints” foram adicionados na tarefa? (Considerando todos os breakpoints incluídos)	0,83	1,12

Fonte: Fábio Pereira da Silva, 2017

Em relação ao uso de “breakpoints” durante a atividade, as ferramentas apresentaram resultados similares. A média da quantidade de “breakpoints” utilizados com a Jaguar

também foi superior, obtendo para a Jaguar um valor médio de 1,12 e para a CodeForest um valor de 0,83.

Tabela 18 – Inclusão de breakpoints no método ou linha do defeito

Questão	Média CodeForest	Média Jaguar
Quantos “breakpoints” foram adicionados na tarefa no método do defeito?	0,32	0,14
Quantos “breakpoints” foram adicionados na tarefa na linha do defeito?	0,01	0

Fonte: Fábio Pereira da Silva, 2017

Quando considerada a inclusão de “breakpoints” no método ou linha do defeito, 6,96% dos participantes incluíram “breakpoints” no método do defeito com a Jaguar, enquanto com a CodeForest este percentual foi de 6,09%. Entretanto, quando avaliada a inclusão de “breakpoints” na linha do defeito nenhum estudante realizou a inclusão com a Jaguar enquanto com a CodeForest 1,74% dos estudantes inseriram “breakpoints” diretamente na linha do defeito.

Tabela 19 – Valores de suspeição

Questão	Média CodeForest	Média Jaguar
Quantos valores de suspeição diferentes foram inspecionados?	5,35	5,38
Quantos métodos/linhas foram inspecionados em sequência?	0,53	1,77
Quantos métodos/linhas em verde foram inspecionados?	1,13	1,01
Quantos métodos/linhas em amarelo foram inspecionados?	0,55	1,08
Quantos métodos/linhas em laranja foram inspecionados?	1,40	4,56
Quantos métodos/linhas em vermelho foram inspecionados?	2,26	5,56

Qual o percentual de participantes que inspecionaram primeiro o método mais suspeito?	46,96%	79,13%
---	--------	--------

Fonte: Fábio Pereira da Silva, 2017

A Jaguar foi capaz de restringir o espaço de busca dos participantes para avaliar linhas mais suspeitas do que a CodeForest, obtendo um valor médio de 5,56 enquanto a CodeForest obteve um valor médio de 2,26.

Tabela 20 – Interações com o trecho do defeito

Questão	Média CodeForest	Média Jaguar
Quanto tempo foi gasto para clicar no método do defeito? (Considerando apenas os participantes que interagiram com o método do defeito)	00:08:09	00:06:39
Quanto tempo foi gasto para clicar no linha do defeito? (Considerando apenas os participantes que interagiram com a linha do defeito)	00:14:21	00:15:31
Quantas vezes o método do defeito foi inspecionado?	4,64	4,02
Quantas vezes a linha do defeito foi inspecionada?	0,05	0,23
Qual o percentual de participantes que encontraram o defeito imediatamente após um clique na ferramenta?	1,74%	0%
Se o programa inspecionado foi o JSoup, quantas vezes o método mais suspeito que chama o método do defeito foi clicado?	5,52	3,44
Se o programa inspecionado foi o JSoup, quantas vezes a linha do método mais suspeito que chama o método do defeito foi clicada?	0,55	1,67
Quantas vezes a linha do defeito foi inspecionada através do editor com a Jaguar?	Não avaliado	1,28

Quantas vezes o método do defeito foi inspecionado através do editor com a Jaguar?	Não avaliado	40,82
--	--------------	-------

Fonte: Fábio Pereira da Silva, 2017

A partir da análise dos “logs” dos participantes, verificou-se que com a CodeForest os estudantes realizaram um maior número de interações com o método do defeito. Entretanto, a Jaguar levou a um maior número de interações com a linha do defeito. Em média, o número de interações com o método do defeito foi de 4,64 interações para a CodeForest e 4,02 interações para a Jaguar. Quando considerada a quantidade de interações com a linha do defeito a CodeForest obteve em média 0,05 interações e a Jaguar 0,23 interações.

Quando avaliado o consumo de tempo gasto para a primeira interação com o método ou linha do defeito considerando apenas as pessoas que encontraram o defeito, a Jaguar direcionou os participantes ao método do defeito em menor período de tempo do que a CodeForest. Entretanto, a CodeForest direcionou os participantes à linha do defeito em menor período de tempo. Enquanto para a Jaguar os participantes demoraram em média 00:06:39 para interagir pela primeira vez com o método do defeito, com a CodeForest o tempo médio foi de 00:08:09. Quando considerada a linha do defeito os participantes demoraram em média 00:15:31 para a Jaguar enquanto para a CodeForest demoraram em média 00:14:21. As Figuras 23 e 24 apresentam os resultados obtidos.

Figura 23 – Tempo médio para interação com o método e linha do defeito

Fonte: Fábio Pereira da Silva, 2017

Figura 24 – Quantidade média de interações com o método e linha do defeito

Fonte: Fábio Pereira da Silva, 2017

Considerando todos os participantes, tendo interagido ou não com o método do defeito, e adotando o limite de 30 minutos, ou seja, toda a tarefa para os participantes que não interagiram com o método ou linha do defeito a Jaguar obteve valores inferiores aos da CodeForest tanto para o método quanto para a linha do defeito. Com a Jaguar os participantes demoraram em média 00:13:33 para interagir com o método do defeito e 00:27:44 para interagir com a linha do defeito. Com a CodeForest o tempo médio foi de 00:13:51 para o método do defeito e 00:29:11 para a linha do defeito.

A avaliação dos “logs” dos estudantes permitiu que fossem obtidas informações detalhadas das interações realizadas pelos participantes na tarefa. Entretanto, para responder as questões de pesquisa é preciso avaliar se os resultados obtidos possuem *significância estatística para a eficácia e eficiência das ferramentas e qual o tamanho do efeito sobre a amostra*. Na próxima seção serão apresentados os testes estatísticos realizados com o objetivo de responder as questões de pesquisa.

6.2 Testes estatísticos

Os resultados obtidos com o programa de extração de informações possibilitaram a análise apenas em relação aos valores médios observados para cada uma das questões. Com o teste pareado de Wilcoxon comparou-se a capacidade das ferramentas direcionarem o participante ao método ou linha do defeito, o tempo despendido até a localização do método ou linha do defeito e a usabilidade das ferramentas medida por meio do TAM.

6.2.1 Teste de hipóteses para avaliação da eficácia

Comparou-se a eficácia da localização de defeitos entre as ferramentas CodeForest e Jaguar em relação à quantidade e à proporção de interações com o método ou linha do defeito. Quando considerada a quantidade de interações foi avaliado o número total de interações que ocorreram com o método ou linha do defeito. Por outro lado, quando avaliada a proporção de interações, avaliou-se quantas vezes o participante interagiu com o método ou linha do defeito em relação ao total do número de interações que ele realizou durante toda a tarefa.

Primeiramente, buscou-se entender se houve diferença entre as ferramentas. As hipóteses avaliadas são listadas abaixo. A Tabela 21 demonstra os resultados obtidos.

H_0 : há igualdade para a eficácia entre as ferramentas CodeForest e Jaguar.

H_1 : há diferenças para a eficácia entre as ferramentas CodeForest e Jaguar.

Tabela 21 – Avaliação da igualdade para a eficácia

Descrição do teste	Valor de p observado
Quantidade de interações com o método do defeito	0,114
Quantidade de interações com a linha do defeito	0,003075
Proporção de interações com o método do defeito	0,3101
Proporção de interações com a linha do defeito	0,00537

Fonte: Fábio Pereira da Silva, 2017

Na avaliação, conforme detalhado no Capítulo 5, optou-se pela utilização de um intervalo de confiança de 95%. No teste de Wilcoxon, verificaram-se diferenças estatísticas para a quantidade de interações com a linha do defeito e para a proporção de interações com a linha do defeito. Sendo assim, rejeitou-se a hipótese nula de que há igualdade para a eficácia entre as ferramentas nestas situações.

A partir da diferença identificada, foram realizados testes de hipóteses com o objetivo de verificar qual ferramenta obteve maior eficácia. As hipóteses consideradas são listadas abaixo. A Tabela 22 apresenta os resultados obtidos.

H_0 : a CodeForest foi mais eficaz do que a Jaguar para a localização de defeitos.

H_1 : a CodeForest não foi mais eficaz do que a Jaguar. Ou seja, a Jaguar foi mais eficaz.

Tabela 22 – Avaliação das diferenças para a eficácia

Descrição do teste	Valor de p observado
Quantidade de interações com a linha do defeito	0,001538
Proporção de interações com a linha do defeito	0,002685

Fonte: Fábio Pereira da Silva, 2017

A hipótese nula foi rejeitada e aceita a hipótese alternativa. Ou seja, *a Jaguar foi mais eficaz do que a CodeForest* considerando um intervalo de confiança de 95%.

6.2.2 Teste de hipóteses para a eficiência

Avaliou-se a eficiência da localização de defeitos das ferramentas CodeForest e Jaguar considerando o tempo gasto pelos participantes para realizar a localização do defeito. Em outras palavras, se a ferramenta foi capaz de levar o desenvolvedor a interagir com o método ou linha do defeito em um menor período de tempo. Na comparação, foram verificadas as seguintes hipóteses:

H_0 : há igualdade para a eficiência entre as ferramentas CodeForest e Jaguar.

H_1 : há diferenças para a eficiência entre as ferramentas CodeForest e Jaguar.

Neste caso, realizou-se a conversão do tempo gasto na localização do defeito em um total de 1800 segundos, ou seja, toda a tarefa que teve duração de 30 minutos para os participantes que não interagiram com o método ou linha do defeito. A Tabela 23 demonstra os resultados obtidos.

Tabela 23 – Avaliação da igualdade para a eficiência

Descrição do teste	Valor de p observado
Tempo médio para a primeira interação com o método do defeito	0,4532
Tempo médio para a primeira interação com a linha do defeito	0,06575
Proporção de tempo para a primeira interação com o método do defeito	0,3674
Proporção de tempo para a primeira interação com a linha do defeito	0,01696

Fonte: Fábio Pereira da Silva, 2017

Verificaram-se diferenças estatísticas para a proporção de tempo gasto pelos participantes para realizar a primeira interação com a linha do defeito. Sendo assim, a hipótese nula foi rejeitada e aceita a hipótese alternativa de que há diferenças entre as ferramentas.

Foram realizados testes de hipóteses para verificar qual ferramenta direcionou o participante a interagir com a linha do defeito em menor período de tempo, ou seja, mais eficiente, com base nas seguintes hipóteses:

H_0 : a CodeForest foi mais eficiente do que a Jaguar.

H_1 : a CodeForest não foi mais eficiente do que a Jaguar. Ou seja, a Jaguar foi mais eficiente.

Quando avaliado o tempo total gasto pelos participantes para interagir com a linha do defeito, não foi possível rejeitar a hipótese nula de igualdade. A Tabela 24 demonstra os resultados obtidos para a proporção de tempo para a primeira interação com a linha do defeito.

Tabela 24 – Avaliação das diferenças para a eficiência

Descrição do teste	Valor de p observado
Proporção de tempo para a primeira interação com a linha do defeito	0,00848

Fonte: Fábio Pereira da Silva, 2017

A hipótese nula foi rejeitada e aceita a hipótese alternativa. Ou seja, *a Jaguar foi mais eficiente do que a CodeForest* considerando um intervalo de confiança de 95%.

6.2.3 Teste de hipóteses para avaliação da usabilidade

Avaliou-se a usabilidade entre as ferramentas também usando os testes pareados de Wilcoxon com base nas respostas obtidas das questões do TAM.

Os testes foram realizados para cada uma das questões e soma das respostas dos estudantes de cada ferramenta, seguindo a mesma abordagem utilizada da avaliação eficácia e eficiência. As hipóteses estudadas são listadas abaixo. A Tabela 25 demonstra os resultados obtidos.

H_0 : há igualdade para a usabilidade entre as ferramentas CodeForest e Jaguar.

H_1 : há diferenças para a usabilidade entre as ferramentas CodeForest e Jaguar.

Tabela 25 – Avaliação da igualdade para a usabilidade

Descrição do teste	Valor de p observado
Supondo que eu tenha acesso à ferramenta, eu imagino que a usaria	0,06721
Dado que eu tive acesso à ferramenta, eu imagino que a usaria	0,01173
Usar a ferramenta melhorou meu desempenho durante a tarefa de depuração	0,01006

Usar a ferramenta durante a tarefa depuração aumentou minha produtividade	0,02485
Usar a ferramenta melhorou minha eficácia durante a atividade de depuração	0,06208
De modo geral, eu achei a ferramenta útil para realizar a tarefa de depuração	0,04758
Aprender a usar a ferramenta foi fácil para mim	0,03745
Interagir com a ferramenta não exigiu muito do meu esforço mental	0,07724
De modo geral, eu achei a ferramenta fácil de usar	0,07659
Eu achei fácil contar com a ferramenta para realizar a tarefa de depuração	0,04537
Resultados completos considerando a soma das respostas dos participantes	0,02054

Fonte: Fábio Pereira da Silva, 2017

Verificaram-se diferenças estatísticas em 6 das 10 questões do TAM. Quando considerados os resultados completos (com todas as respostas), também foram observadas diferenças estatísticas. Em outras palavras, considerando que a hipótese nula foi rejeitada avaliou-se qual ferramenta obteve maior usabilidade na percepção dos estudantes. Adotaram-se as hipóteses descritas a seguir. A Tabela 26 demonstra os resultados obtidos para cada uma das questões que apresentaram diferenças estatísticas.

H_0 : a CodeForest obteve maior usabilidade do que a Jaguar.

H_1 : a CodeForest não obteve maior usabilidade do que a Jaguar. Ou seja, a Jaguar obteve maior usabilidade.

Tabela 26 – Avaliação das diferenças para a usabilidade

Descrição do teste	Valor de p observado
Dado que eu tive acesso à ferramenta, eu imagino que a usaria	0,005863
Usar a ferramenta melhorou meu desempenho durante a tarefa de depuração	0,00503

Usar a ferramenta durante a tarefa depuração aumentou minha produtividade	0,01243
De modo geral, eu achei a ferramenta útil para realizar a tarefa de depuração	0,02379
Aprender a usar a ferramenta foi fácil para mim	0,01873
Eu achei fácil contar com a ferramenta para realizar a tarefa de depuração	0,02268
Resultados completos considerando a soma das respostas dos participantes	0,01027

Fonte: Fábio Pereira da Silva, 2017

Em todas as questões, rejeitou-se a hipótese nula e aceitou-se a hipótese alternativa de que a Jaguar apresentou maior usabilidade do que a CodeForest. Quando avaliados os resultados completos do TAM ela também foi superior. Ou seja, *a Jaguar obteve maior usabilidade do que a CodeForest*.

6.2.4 Tamanho do efeito

Como mencionado no Capítulo 5, além de verificar a diferença estatística também é necessário avaliar a representatividade desta diferença sobre a amostra.

Tendo em vista esta necessidade, optou-se por utilizar a medida Delta de Cliff para avaliar o tamanho do efeito nas questões em que rejeitou-se a hipótese de igualdade entre as ferramentas. Consideraram-se nas análises os valores apresentados na Tabela 11 do Capítulo 5. A Tabela 27 mostra os resultados obtidos nas questões do TAM que apresentaram diferenças estatísticas.

Tabela 27 – Representatividade para o TAM

Descrição do teste	Valor observado
Dado que eu tive acesso a ferramenta, eu imagino que a usaria	0,21504
Usar a ferramenta melhorou meu desempenho durante a tarefa de depuração	0,18865

Usar a ferramenta durante a tarefa depuração aumentou minha produtividade	0,14967
De modo geral, eu achei a ferramenta útil para realizar a tarefa de depuração	0,1324
Aprender a usar a ferramenta foi fácil para mim	0,16514
Eu achei fácil contar com a ferramenta para realizar a tarefa de depuração	0,13498

Fonte: Fábio Pereira da Silva, 2017

Em quatro das seis afirmações que apresentaram diferenças estatísticas observou-se uma diferença pequena entre os grupos. Nas afirmações “De modo geral, eu achei a ferramenta útil para realizar a tarefa de depuração” e “Eu achei fácil contar com a ferramenta para realizar a tarefa de depuração” a diferença observada foi insignificante.

Tabela 28 – Representatividade para os resultados

Descrição do teste	Valor observado
Quantidade de interações com a linha do defeito	0,14183
Proporção de interações com a linha do defeito	0,14268
Proporção de tempo para a primeira interação com a linha do defeito	0,13862
Resultados completos considerando a soma das respostas dos participantes	0,17515

Fonte: Fábio Pereira da Silva, 2017

Mesmo tendo sido apresentadas diferenças estatísticas entre as ferramentas CodeForest e Jaguar para a quantidade e proporção de interações com a linha do defeito (eficácia), proporção de tempo gasto para a primeira interação com a linha do defeito (eficiência) *a diferença entre os grupos observada foi insignificante sobre a amostra*. Entretanto, quando avaliada a soma das respostas do TAM (usabilidade) *o tamanho do efeito que esta diferença representa foi pequeno entre os grupos*.

6.3 Resultados do TAM

6.3.1 Coeficiente de Cronbach

Para o cálculo do coeficiente de Cronbach comparou-se o número de questões com a quantidade de respostas. A avaliação incluiu os três grupos do TAM, intenção de uso, utilidade percebida e facilidade de uso.

A Tabela 29 apresenta os resultados calculados para o coeficiente de Cronbach considerando todas as respostas do questionário.

Tabela 29 – Análise de Cronbach

Ferramenta	Utilidade percebida	Facilidade de uso	Intenção de uso
Jaguar	0,96	0,95	0,95
CodeForest	0,96	0,94	0,98

Fonte: Fábio Pereira da Silva, 2017

É possível perceber que as variáveis obtiveram valores bem superiores do que 0,7, apresentando alto índice de confiabilidade em todas as situações, chegando a alcançar o valor de 0,98 para o grupo de intenção de uso da CodeForest.

6.3.2 Análise fatorial

Para a realização da análise fatorial, foram comparadas a utilidade e facilidade de uso visando avaliar o quanto eles influenciam no comportamento de uso das ferramentas.

As Tabelas 30 e 31 mostram os resultados da análise fatorial considerando todas as respostas obtidas no experimento. Como mencionado na Tabela 9, cada uma das afirmações representa um fator influenciado no TAM.

Tabela 30 – Análise fatorial para a Jaguar

Fator	Utilidade percebida	Facilidade de uso
Utilidade	0,867	0,338
Produtividade	0,901	0,331
Eficácia	0,919	0,304
Usabilidade	0,758	0,446

Facilidade de aprendizado	0,368	0,879
Esforço mental	0,285	0,861
Facilidade de manuseio	0,367	0,913
Facilidade da tarefa	0,648	0,603

Fonte: Fábio Pereira da Silva, 2017

Tabela 31 – Análise fatorial para a CodeForest

Fator	Utilidade percebida	Facilidade de uso
Utilidade	0,889	0,376
Produtividade	0,922	0,322
Eficácia	0,910	0,306
Usabilidade	0,695	0,524
Facilidade de aprendizado	0,357	0,839
Esforço mental	0,243	0,825
Facilidade de manuseio	0,340	0,887
Facilidade da tarefa	0,571	0,715

Fonte: Fábio Pereira da Silva, 2017

Considerando que quanto maior o valor obtido na análise fatorial maior a relação com o fator considerado, é possível perceber que as questões de utilidade e facilidade de uso possuem maior relação entre o grupo de questões do TAM no qual elas pertencem.

Ao inverso, quando considerado o grupo no qual elas não pertencem, o valor obtido é bastante inferior, indicando a existência de uma baixa relação.

6.3.3 Análise da correlação

Para entender como os diferentes grupos do TAM se relacionam entre si calculou-se a correlação entre eles, que pode ser visualizada na Tabela 32.

Tabela 32 – Análise da correlação

Grupo do TAM	Jaguar	CodeForest
Utilidade x Facilidade de uso	0,7774469	0,7347146

Utilidade x Intenção de uso	0,7787792	0,850972
Facilidade de uso x Intenção de uso	0,6275027	0,6563972

Fonte: Fábio Pereira da Silva, 2017

É possível perceber que a utilidade e a facilidade de uso juntamente com a utilidade e a intenção de uso possuem uma alta correlação nas respostas obtidas no questionário.

Em outras palavras, o grau com que as pessoas acreditam que o uso das ferramentas CodeForest e Jaguar irá melhorar o seu desempenho durante a depuração possui alta relação com a facilidade e intenção de uso das ferramentas. A facilidade de uso das ferramentas CodeForest e Jaguar também possui uma alta relação com a intenção de uso.

As Figura 25 e 26 mostram como cada uma das variáveis estudadas nos grupos de utilidade e de facilidade de uso se relacionam entre si. Quanto mais escuro o círculo demarcado na representação maior é a relação que a variável estudada possui. Quanto mais claro menor é a relação que as variáveis possuem.

Figura 25 – Correlação entre as variáveis para a Jaguar

Fonte: Fábio Pereira da Silva, 2017

Figura 26 – Correlação entre as variáveis para a CodeForest

Fonte: Fábio Pereira da Silva, 2017

Ao avaliar a correlação entre as variáveis quando considerada a Jaguar, é possível perceber que as variáveis de esforço mental e facilidade de aprendizado possuem uma alta correlação com a variável de facilidade de manuseio. A facilidade da tarefa possui uma alta correlação com a usabilidade. A utilidade possui uma alta relação com as variáveis de produtividade e eficácia.

No caso da CodeForest, a variável de esforço mental possui uma alta relação com a variável de facilidade de manuseio. A facilidade de aprendizado possui uma alta relação com a facilidade de manuseio e facilidade da tarefa. A variável de eficácia possui uma alta relação com a utilidade e a produtividade. A usabilidade possui uma alta relação com a utilidade, produtividade e facilidade da tarefa.

Novamente, como na análise fatorial, houve uma alta correlação com as questões relacionadas ao grupo no qual as variáveis pertencem.

6.3.4 Representação dos dados

A análise incluiu a elaboração de *boxplots* para todos os grupos do TAM visando verificar a existência de *outliers* nas respostas obtidas no questionário. Também foi

avaliado se houve diferenças estatísticas em relação a mediana do conjunto de respostas dos participantes.

Apenas o grupo de facilidade de uso apresentou diferenças na linha central que determina a mediana do conjunto de dados para as ferramentas CodeForest e Jaguar, conforme pode ser visualizado nas Figuras 27 e 28. A partir da representação é possível perceber a inexistência de *outliers* nas respostas obtidas no questionário em relação a este grupo do TAM.

Figura 27 – Facilidade de uso da ferramenta Jaguar

Fonte: Fábio Pereira da Silva, 2017

Figura 28 – Facilidade de uso da ferramenta CodeForest

Fonte: Fábio Pereira da Silva, 2017

Considerando a Jaguar, no item de facilidade de uso, a mediana da maior concentração do conjunto de dados está sempre com o valor igual a dois em todos os fatores considerados. Quando avaliado o uso da CodeForest, neste mesmo item a mediana é menor do que três em relação ao esforço mental e facilidade de uso.

6.3.5 Distribuição das respostas

Foi calculada a moda de cada fator avaliado no conjunto de dados. A Tabela 33 apresenta os resultados obtidos considerando todas as respostas do experimento.

Tabela 33 – Distribuição das respostas

Fator	Jaguar	CodeForest
Usabilidade	1	1
Produtividade	3	3
Eficácia	3	1
Utilidade da ferramenta	1	1
Facilidade de aprendizado	1	1
Esforço mental	1	2
Facilidade de manuseio	1	1
Facilidade da tarefa	1	1

Intenção de uso	2	1
Consideração de uso	2	1

Fonte: Fábio Pereira da Silva, 2017

É possível perceber que o valor que mais se repetiu nas respostas dos participantes foi um, ou seja, o valor máximo de concordância com a afirmação da questão dentro da escala Likert considerada.

A partir destes valores, também podem ser realizadas comparações em relação à cada fator avaliado sobre o potencial de uso futuro das ferramentas. Quando considerado os fatores de eficácia, intenção de uso e consideração de uso, a CodeForest teve uma melhor avaliação em relação a Jaguar. No entanto, quando considerado o fator de esforço mental a Jaguar obteve melhor avaliação. Nos outros fatores considerados, as ferramentas obtiveram o mesmo valor em relação a moda do conjunto de dados.

6.4 Resultados do questionário

Nesta seção serão descritos os resultados obtidos em relação as demais questões do questionário aplicado aos participantes ao término do experimento, fornecendo maior embasamento tanto na distribuição dos participantes durante a atividade, quanto na quantidade de pessoas que conseguiram realizar a localização da classe, método e linha do defeito.

6.4.1 Experiência na área de desenvolvimento

Como mencionado no Capítulo 5, foram conduzidos experimentos com estudantes do 4º, 5º e 6º semestres do curso de Análise e Desenvolvimento de Sistemas da Faculdade de Tecnologia da Zona Leste (Fatec). Ao todo, 30,3% dos alunos estavam matriculados no 4º semestre do curso, 39,5% no 5º semestre e 30,3% no 6º semestre. A Figura 29 apresenta informações relacionadas a experiência profissional dos estudantes que participaram da atividade.

Figura 29 – Experiência profissional na área de desenvolvimento

Fonte: Fábio Pereira da Silva, 2017

A ampla maioria dos estudantes possuem experiência profissional de menos de um ano, 18,5% possuem um ano de experiência profissional, 10,9% possuem dois anos, 2,5% possuem três anos e apenas 1,7% possuem quatro anos de experiência profissional.

Quando perguntado quantos anos a pessoa programa em Java, profissionalmente ou não, 26,9% responderam que programam há um ano, 39,5% responderam que programam há dois anos, 17,6% programam em Java há três anos, 6,7% programam há quatro anos e 3,4% programam há cinco anos.

Quando perguntado o nível de conhecimento dos participantes na linguagem Java, a maior parte dos estudantes informou que possui conhecimentos básicos, ou seja, 63% dos participantes, 34,5% responderam que possuem conhecimentos intermediários e apenas 1,7% responderam conhecimentos avançados.

Foi perguntado também há quantos anos o estudante realiza testes automatizados com o JUnit ou outro framework de testes automatizados. Ao todo, 63,9% dos participantes responderam que utilizam algum framework há menos de um ano, 29,4% responderam que utilizam algum framework de testes automatizados há um ano, 5,9% há dois anos e 0,8% há três anos.

Com o objetivo de obter maior entendimento de como é executada a atividade de depuração no cotidiano dos programadores foi perguntado quais recursos são mais utilizados para realizar a depuração de programas. Ao todo, 58,8% dos participantes informaram no questionário que utilizam na atividade de depuração “breakpoints”, 35,3% responderam que utilizam comandos de escrita, 47,9% responderam que utilizam os resultados dos testes e 26,1% responderam a pilha de execução (“Stack Trace”). Nesta questão, apenas 4,2% afirmaram que utilizam o JUnit.

6.4.2 Distribuição dos participantes

Os estudantes foram divididos em 8 grupos com o objetivo de garantir a aleatoriedade na condução do experimento, de modo que os resultados fossem independentes da ferramenta utilizada, do programa avaliado e da ordem de utilização durante a atividade. A Figura 30 demonstra como ficou a distribuição dos participantes.

Com base nos resultados obtidos é possível perceber que não houve uma diferença significativa entre os valores na distribuição de cada grupo nos experimentos.

Figura 30 – Distribuição dos participantes

6.4.3 Localização de defeitos

Perfil dos estudantes que encontraram o defeito

Ao verificar a experiência profissional dos estudantes que conseguiram realizar a localização e a correção do defeito três deles informaram que possuem um ano de experiência profissional e estavam matriculados no 6º semestre do curso. Um possui um ano de experiência profissional e estava matriculado no quinto semestre do curso. Apenas um deles informou que não possui experiência profissional e estava matriculado no 6º semestre do curso.

JSoup

Nenhum participante conseguiu realizar a localização do defeito no programa JSoup. Alguns conseguiram chegar próximo ao defeito, conforme verificado no resultados descritos sobre os “logs”. A Tabela 34 sumariza com base nas respostas dos estudantes em relação a quantidade de alunos que conseguiram localizar, respectivamente, a classe do defeito, o método do defeito ou linha do defeito para o programa avaliado.

Tabela 34 – Localização do defeito - JSoup

Descrição	Quantidade
Classe do defeito	58
Método do defeito	4
Linha do defeito	0

Fonte: Fábio Pereira da Silva, 2017

Dado que nenhum participante interagiu com o método e linha do defeito, verificaram-se as interações que ocorreram no método *nextElementSibling()*, responsável por fazer a chamada ao método do defeito. Considerando o questionário, 44 pessoas responderam que o defeito se encontrava neste método, ou seja, 36,97% confirmando esta tendência observada no programa de extração de informações, de um alto número de interações no método que chamava o método do defeito.

Quando perguntado em uma escala de 1 a 7 qual a dificuldade de realizar a localização do defeito no programa JSoup, 31,1% responderam 7, ou seja, muito difícil.

XStream

Ao todo cinco pessoas conseguiram realizar a localização do defeito no programa XStream. Todos os participantes que realizaram a localização do defeito, conseguiram corrigi-lo. A Tabela 35, demonstra a quantidade de pessoas que conseguiram localizar a classe, método ou linha do defeito.

Tabela 35 – Localização do defeito - XStream

Descrição	Quantidade
Classe do defeito	46
Método do defeito	35
Linha do defeito	5

Fonte: Fábio Pereira da Silva, 2017

Quando perguntado qual a dificuldade de se realizar a localização do defeito no programa XStream, 48,7% responderam valores iguais a 7, muito difícil para a maioria dos participantes.

Considerando a soma das localizações realizadas com os programas XStream e JSoup, 104 pessoas conseguiram localizar a classe do defeito, 87,39% dos participantes, 39 pessoas localizaram o método do defeito, representando 32,77% dos participantes e 5 estudantes, totalizando 4,20% localizaram e corrigiram o defeito. A Figura 31 apresenta os resultados obtidos.

Figura 31 – Participantes que encontraram a classe, método ou linha do defeito

Fonte: Fábio Pereira da Silva, 2017

6.4.4 Opinião sobre a Jaguar

O questionário aplicado incluiu algumas questões sobre a opinião dos participantes em relação ao uso da ferramenta. Quando perguntado se a Jaguar ajudou o participante a encontrar o defeito, 31,9% responderam que a Jaguar foi útil para encontrar o defeito. Considerando a rejeição à ferramenta, 23,5% responderam que a Jaguar foi pouco útil.

Foram elaboradas também outras questões para entender a opinião dos participantes em relação aos recursos disponíveis na Jaguar avaliada através da escala Likert com opções para que o participante informasse valores de 1 a 7. Em que 1 indica que a ferramenta não foi útil e 7 indica que a ferramenta foi útil. A Tabela 36 detalha os resultados obtidos.

Tabela 36 – Avaliação das funcionalidades da Jaguar

Questão	Valor mais observado	Percentual
A organização das informações fornecida pela Jaguar foi útil?	7	33,6%

Qual é a sua opinião sobre o esquema de cores fornecido pela Jaguar?	7	67,2%
Qual é a sua opinião sobre a lista fornecida pela Jaguar?	7	40,3%
Qual é a sua opinião sobre o controle slider fornecido pela Jaguar?	7	31,1%
Qual é a sua opinião sobre o filtro de busca textual fornecido pela Jaguar?	7	29,4%

Fonte: Fábio Pereira da Silva, 2017

A maioria das respostas obtidas foi igual a 7 para todas as questões, ou seja, o valor máximo dentro da escala Likert considerada. A questão que recebeu o maior percentual de respostas iguais a 7 estava relacionada à utilidade do esquema de cores fornecida pela ferramenta.

6.4.5 Opinião sobre a CodeForest

O questionário aplicado também incluiu questões com o objetivo de obter a opinião dos participantes em relação à CodeForest. Quando perguntado se a CodeForest ajudou o participante a encontrar o defeito, 32,8% responderam que a ferramenta foi útil. Considerando a rejeição à ferramenta, 30,3% responderam que a ferramenta foi pouco útil para encontrar o defeito. O questionário incluiu perguntas para que fossem obtidas as opiniões dos participantes em relação aos recursos disponíveis na ferramenta com base na escala Likert, os valores obtidos podem ser visualizados na Tabela 37.

Tabela 37 – Avaliação das funcionalidades da CodeForest

Questão	Valor mais observado	Percentual
A organização das informações fornecida pela CodeForest foi útil?	6	26,1%
Qual é a sua opinião sobre a representação visual fornecida pela CodeForest?	7	26,1%
Qual é a sua opinião sobre a interação com o código fonte por meio da floresta de cactus fornecida pela CodeForest?	7	26,1%
Qual é a sua opinião sobre o ranking fornecido pela CodeForest?	7	27,7%
Qual é a sua opinião sobre o filtro de busca textual fornecido pela CodeForest?	7	25,2%

Fonte: Fábio Pereira da Silva, 2017

Apenas em uma das questões a maioria das respostas obtidas não foi igual a 7. A pergunta que recebeu o maior percentual de respostas iguais a 7 foi relacionada o ‘ranking’ dos elementos mais suspeitos.

Observando os resultados obtidos, os valores da Jaguar foram bastante superiores a CodeForest. Enquanto a opção melhor avaliada na Jaguar teve o percentual de 67,2% respostas iguais a 7 a melhor avaliada na CodeForest teve o percentual de apenas 27,7%.

Comparando os valores médios obtidos, a Jaguar obteve valores iguais a 7 para 40,33% dos participantes enquanto a CodeForest obteve valores iguais a 7 para 25,55% dos participantes.

6.5 Discussão

Esta seção dedica-se à apresentação das discussões dos resultados obtidos sobre o uso das ferramentas CodeForest e Jaguar na atividade de depuração tanto com base na avaliações dos “logs” dos participantes quanto na avaliação da percepção ao uso das ferramentas.

Primeiramente, serão apresentadas considerações para a avaliação dos “logs” dos participantes sobre os resultados do experimento. Em seguida, sobre a análise estatística do questionário e demais respostas dos estudantes nas outras questões abordadas não relacionadas ao TAM. Por último, serão apresentadas as considerações em relação a cada uma das questões de pesquisa.

6.5.1 Síntese dos resultados do experimento

Realizou-se uma avaliação nos “logs” de todos os participantes através do programa de extração de informações que permitiu que 42 tópicos de interesse fossem avaliados com base nas informações obtidas durante toda a tarefa.

As questões mencionadas serviram de subsídio para a análise estatística realizada visando responder as questões de pesquisa. A Tabela 38 apresenta as considerações para os 9 grupos em que as 42 questões extraídas das ferramentas foram divididas.

Tabela 38 – Síntese dos resultados dos logs

Grupo	Ferramenta melhor avaliada	Considerações
Uso das ferramentas durante a atividade	Jaguar	Jaguar foi mais utilizada do que a CodeForest
Interrupções no uso das ferramentas durante a atividade	CodeForest	A Jaguar obteve maior quantidade de gaps em sua utilização em relação a CodeForest.
Interações com a ferramenta	CodeForest	A CodeForest obteve um maior número de interações com a ferramenta.
Linhas inspecionadas	Jaguar	A Jaguar foi capaz de restringir melhor o espaço de busca dos participantes.
Interações com depuradores e o JUnit	Jaguar e CodeForest	Houve maior quantidade de interações com depuradores e com o JUnit através do uso da Jaguar. Entretanto a utilização do JUnit e depuradores foi maior quando realizada imediatamente após uma interação com a CodeForest.

Inclusão de “break-points”	Jaguar	A Jaguar foi capaz de restringir mais o espaço de busca do programador com mais “breakpoints” incluídos no método ou linha clicada.
Inclusão de “break-points” no método ou linha do defeito	CodeForest	A CodeForest direcionou o programador a incluir “breakpoints” no método ou linha do defeito mais do que a Jaguar.
Valores de suspeição	Jaguar	A Jaguar possibilitou maior restrição do espaço de busca do programador durante a depuração.
Interações com o trecho do defeito	CodeForest e Jaguar	A CodeForest possibilitou maior interação com o método do defeito em relação a Jaguar. Entretanto, a Jaguar permitiu que o defeito fosse localizado em menor período de tempo em relação a CodeForest. Considerando todos os estudantes, a Jaguar foi capaz de direcionar os participantes ao trecho do defeito em menor período de tempo em relação a CodeForest, tanto para o método quanto para a linha do defeito.

Fonte: Fábio Pereira da Silva, 2017

6.5.2 Síntese dos resultados do TAM

Realizou-se a avaliação das respostas dos participantes em relação às questões do TAM no questionário aplicado ao término do experimento.

Esta avaliação foi fundamental para entendimento da percepção dos participantes em relação à usabilidade das ferramentas para a localização de defeitos. Juntamente com a análise das interações que ocorreram através do programa de extração de informações, permitiu entender a capacidade das ferramentas serem introduzidas em situações reais

de desenvolvimento e o quanto elas podem auxiliar os desenvolvedores na atividade de depuração.

A Tabela 39 demonstra considerações para cada um dos grupos do TAM avaliados em relação à média de respostas obtidas dos participantes.

Tabela 39 – Síntese dos resultados por grupo do TAM

Grupo	Ferramenta melhor avaliada	Consideração
Intenção de uso	Jaguar	A Jaguar foi melhor avaliada do que a CodeForest. Com base nos resultados da Tabela 26, em uma das duas questões deste grupo foi observada diferença estatística.
Utilidade da ferramenta	Jaguar	A Jaguar foi melhor avaliada do que a CodeForest. Três das quatro questões que avaliaram a utilidade apresentaram diferenças estatísticas considerando um intervalo de confiança de 95%.
Facilidade de uso	Jaguar	A Jaguar foi melhor avaliada do que a CodeForest. Em duas das quatro questões que avaliaram o grupo apresentaram diferenças estatísticas.

Fonte: Fábio Pereira da Silva, 2017

A mediana da quantidade de respostas observadas entre os participantes teve resultados muito similares para as ferramentas. Apenas no grupo de facilidade de uso observou-se diferenças nos fatores de facilidade de aprendizado e facilidade da tarefa que obtiveram melhor avaliação com a Jaguar.

Entretanto, é importante considerar que quando avaliada a moda do conjunto de dados à CodeForest obteve melhor avaliação do que a Jaguar nos fatores de eficácia, intenção de uso e consideração de uso enquanto a Jaguar foi superior a CodeForest apenas no fator de esforço mental.

Em termos da análise fatorial entre os grupos do TAM as respostas dos participantes tiveram claramente maior relação no grupo do TAM no qual elas pertencem. Os maiores valores de correlação entre os grupos do TAM ocorreram entre utilidade e facilidade de uso, juntamente com a utilidade e intenção de uso. No entanto, também verificou-se uma alta correlação entre os grupos de facilidade de uso e intenção de uso.

Avaliou-se a consistência do questionário usando o coeficiente de Cronbach que teve como resultado uma alta confiabilidade na análise estatística realizada.

6.5.3 Síntese dos resultados do questionário

Ao término do experimento apenas cinco participantes conseguiram realizar a localização e a correção do defeito. Dois deles utilizaram a CodeForest e os outros três participantes utilizaram a Jaguar. Considera-se que o percentual de participantes que localizaram e corrigiram o defeito foi extremamente baixo, pois apenas um percentual de 4,20% dos estudantes conseguiram encontrá-lo.

Através destes resultados realizou-se uma avaliação abrangente das interações que ocorreram com as ferramentas, em especial no método ou linha do defeito. De fato, a utilização das ferramentas CodeForest e Jaguar foi capaz de levar os estudantes a interagirem com o método ou linha do defeito. O principal fator que pode ter causado o baixo desempenho dos estudantes é a baixa experiência profissional na área de desenvolvimento, pois 66,64% dos estudantes que fizeram a atividade não tinham nenhuma experiência profissional.

Avaliando os estudantes que localizaram a classe do defeito, 104 pessoas conseguiram encontrá-la, o que representa um percentual de 87,39%. Quando avaliado o método do defeito apenas 32,77% dos estudantes encontraram o método do defeito, um percentual muito superior aos 4,20% que localizaram e corrigiram a linha do defeito.

6.5.4 Questões de pesquisa

1. Ferramentas visuais melhoram a eficácia e eficiência da atividade de depuração em comparação com ferramentas baseadas apenas em informações textuais?

Para a primeira questão de pesquisa, verificou-se que o uso de ferramentas textuais de depuração quando utilizada a Jaguar foi mais eficaz e eficiente para a localização de defeitos do que ferramentas visuais de depuração quando utilizada a CodeForest, como detalhado nas Tabelas 22 e 24.

Entretanto, ao avaliar o tamanho do efeito sobre a amostra entre as ferramentas todos os valores obtidos foram inferiores a 0,147, mínimo para a representatividade de um efeito pequeno sobre a amostra, conforme apresentado na Tabela 28.

Para se ter uma ideia da representatividade dos resultados do Delta de Cliff, foi avaliado por meio do software R a probabilidade de igualdade e superioridade entre os grupos.

Excluindo todas as comparações em pares das amostras das ferramentas CodeForest e Jaguar a probabilidade de um elemento aleatório do grupo de valores da Jaguar possuir igualdade a um elemento aleatório do grupo de valores da CodeForest é 80,5% e 79,95%, respectivamente, para a quantidade e proporção de interações com a linha do defeito. Ao avaliar a eficiência das ferramentas a probabilidade de igualdade é de 79,95% das amostras, para a proporção de tempo para a primeira interação com a linha do defeito.

Apesar da diferença estatística observada em relação a eficácia e eficiência no teste de Wilcoxon entre as ferramentas para a capacidade de levar os participantes a interagirem com a linha do defeito e o tempo gasto até localizá-lo, o tamanho do efeito sobre a amostra obtido no Delta de Cliff é insignificante.

2. Ferramentas visuais melhoram a usabilidade da atividade de depuração em comparação com ferramentas baseadas apenas em informações textuais?

Com base nas respostas obtidas pelos estudantes no TAM, o uso de ferramentas textuais de depuração, quando utilizada a ferramenta Jaguar, teve maior grau de usabilidade do que o uso de ferramentas visuais de depuração, quando utilizada a CodeForest. Observou-se significância estatística em 6 dos 10 fatores do TAM representados por cada afirmação que fazia parte da composição do questionário. Em nenhuma situação a CodeForest obteve melhor avaliação do que a Jaguar.

Além da significância estatística apresentada pelo teste de Wilcoxon, também verificou-se um efeito pequeno sobre a amostra em quatro das seis afirmações e para a soma das respostas do TAM com valores acima do que 0,147, conforme pode ser visualizado nas Tabelas 27 e 28.

Os valores observados no Delta de Cliff para a soma das respostas do TAM de cada participante, excluindo todas as comparações em pares entre as amostras das ferramentas CodeForest e Jaguar, indicam que a probabilidade de um elemento aleatório do grupo da Jaguar ter valor igual a um elemento aleatório do grupo da CodeForest é 2,42%. Nesta mesma situação, a probabilidade de ser obtido um valor que indique que a Jaguar foi melhor avaliada do que a CodeForest na percepção dos participantes é 55,31%.

Com base nos resultados obtidos no teste de Wilcoxon e Delta de Cliff, há indícios de que a Jaguar obteve maior usabilidade do que a CodeForest através da percepção dos estudantes tanto estatisticamente quanto no tamanho do efeito sobre a amostra.

6.6 Ameaças à validade

Os resultados de um projeto de pesquisa são influenciados por fatores que afetam à validade externa, interna e de construto.

A validade externa indica a capacidade dos resultados de um projeto de pesquisa serem generalizados para outros trabalhos. Os resultados obtidos sobre o uso de metáforas visuais e textuais de depuração limitam-se às ferramentas CodeForest e Jaguar. Existem outras ferramentas de depuração que não foram avaliadas neste projeto de pesquisa, como a Tarantula (JONES; HARROLD; STASKO, 2002) e a GZoltar (GOLVEIA; CAMPOS; ABREU, 2013). Apesar de não existir uma quantidade significativa de ferramentas visuais e textuais de depuração, a experimentação com várias ferramentas seria extremamente complexa de ser realizada.

Considera-se ainda que na literatura não foram encontradas avaliações sobre o uso de ferramentas visuais e textuais de depuração. Apenas a ferramenta GZoltar foi avaliada no trabalho de Golveia, Campos e Abreu (2013).

A validade interna avalia as condições básicas aplicáveis ao estudo. Neste trabalho, à validade interna é altamente influenciada pela experiência profissional das pessoas que fizeram o experimento. A condução do estudo ocorreu apenas com alunos de curso de graduação que em geral possuem pouca ou nenhuma experiência profissional na área de desenvolvimento de software. Os resultados obtidos nos experimentos poderiam ter sido melhores se tivessem sido aplicados com profissionais com maior experiência. Buscou-se

uma amostra significativa de pessoas que já atuam na indústria; porém, não foi possível obter uma quantidade viável para ser aplicada neste projeto de pesquisa.

Os defeitos e programas selecionados podem ser muito complexos para serem aplicados com estudantes de curso de graduação que ainda estão começando a adquirir familiaridade com desenvolvimento de software. No entanto, foram utilizadas outras medidas, como a quantidade e proporção de cliques no método e linha do defeito, que permitiram avaliar a eficácia e a eficiência das ferramentas.

Em relação ao questionário TAM muitas das percepções que os estudantes tiveram talvez não se refletissem caso o experimento fosse conduzido com profissionais mais experientes na área de desenvolvimento.

A validade de construto avalia o quanto uma medida está correlacionada com o grupo de variáveis que podem influenciar os resultados do estudo. A eficácia, eficiência e usabilidade das ferramentas CodeForest e Jaguar foram verificadas por meio de vários testes estatísticos com uma amostra significativa de participantes.

6.7 Considerações finais

Neste capítulo, foram apresentadas as interações com as ferramentas CodeForest e Jaguar ocorridas no experimento, a análise estatística, a avaliação do TAM e as questões respondidas pelos participantes. Os resultados apresentados foram discutidos em detalhes e serviram de subsídio para que fossem respondidas as questões de pesquisa.

7 Conclusões

Este trabalho teve como objetivo avaliar a eficiência, a eficácia e a usabilidade de duas metáforas visuais de depuração. A metáfora visual foi representada pela ferramenta CodeForest enquanto a metáfora textual foi representada pela ferramenta Jaguar.

Os mais variados aspectos de utilização das ferramentas CodeForest e Jaguar foram avaliados. A avaliação incluiu uma análise detalhada do uso das ferramentas durante a atividade, verificando-se as interações executadas pelos participantes, o uso de outros meios disponíveis no ambiente para a localização do defeito como testes de unidades e “breakpoints” bem como as interações que ocorreram com o trecho do defeito.

O trabalho consistiu na realização de um estudo com 119 estudantes do curso de graduação em Análise e Desenvolvimento de Sistemas da Faculdade de Tecnologia da Zona Leste onde as ferramentas avaliadas neste projeto de pesquisa foram utilizadas pelos estudantes em uma avaliação experimental para a localização de defeitos. A eficácia e a eficiência foram avaliadas, respectivamente, pela capacidade da ferramenta direcionar o desenvolvedor ao método ou linha do defeito e o tempo necessário para localizá-los. A usabilidade das ferramentas foi avaliada por meio de um questionário baseado no modelo TAM (*Technology Acceptance Model*).

Com base nos dados extraídos, realizaram-se testes estatísticos para avaliar qual ferramenta foi mais eficaz, eficiente e com maior usabilidade para a localização de defeitos. Com o teste de Wilcoxon, buscou-se entender qual ferramenta foi melhor para cada uma das questões de pesquisa considerando um intervalo de confiança de 95%. Com a medida Delta de Cliff, buscou-se entender, para os testes que apresentaram significância estatística, qual a representatividade sobre a amostra, ou seja, o tamanho do efeito. Os resultados obtidos indicam que a Jaguar foi mais eficaz, eficiente e com maior grau de usabilidade do que a CodeForest; entretanto, o tamanho do efeito estatístico é insignificante para a eficácia e eficiência e baixo para a usabilidade.

O experimento descrito nesta pesquisa apresentou, que seja do nosso conhecimento, o primeiro estudo na literatura com uma quantidade significativa de pessoas sobre o uso de ferramentas visuais e textuais de depuração, alcançando o objetivo de dar indicações para os desenvolvedores e pesquisadores em depuração de software sobre a eficiência, eficácia e usabilidade de ferramentas visuais e textuais de depuração.

A principal limitação deste trabalho é a ausência de profissionais com maior experiência na área de desenvolvimento, que poderia trazer uma melhoria significativa aos resultados finais do experimento. Outra limitação foi o uso apenas das ferramentas CodeForest e Jaguar para a condução dos experimentos diante da dificuldade de realizar a experimentação com outras ferramentas de depuração.

7.1 Contribuições

Este trabalho apresentou como principais contribuições.

1. Uma revisão da literatura sobre a evolução das ferramentas visuais de depuração no últimos anos.
2. Implementação de novas funcionalidades na ferramenta CodeForest, tornando-a apta para a execução dos experimentos.
3. Um desenho experimental aplicado ao uso de ferramentas visuais e textuais de depuração.
4. Implementação de um programa de extração de informações que possibilitou a realização de uma avaliação detalhada das interações dos participantes que executaram o experimento com as ferramentas.
5. Aplicação do questionário TAM (*Technology Acceptance Model*) para avaliar a percepção dos estudantes quanto à eficácia, à eficiência e à usabilidade das ferramentas CodeForest e Jaguar para a localização de defeitos.
6. Primeiro estudo experimental sobre o uso de ferramentas visuais e textuais de depuração.

7.2 Trabalhos Futuros

Estudos com usuários de ferramentas visuais e textuais de depuração são bastante escassos na literatura. Anteriormente a este projeto, apenas o trabalho de Golveia, Campos e Abreu (2013) apresentou uma avaliação sobre a eficácia e eficiência da ferramenta GZoltar para a localização de defeitos. Além disso, foram encontradas poucas ferramentas visuais de depuração na literatura, e com exceção da GZoltar, não apresentam resultados de

avaliações que demonstrem o quanto elas podem auxiliar os desenvolvedores na atividade de depuração.

Tais considerações reforçam a necessidade de novos experimentos que combinem o uso de ferramentas de depuração e a avaliação de sua eficácia, eficiência e usabilidade na localização de defeitos.

Com relação ao uso de ferramentas visuais e textuais de depuração, novos estudos podem:

- replicar o experimento com estudantes mais experientes, por exemplo, de cursos de pós-graduação;
- repetir o experimento com outras ferramentas como Tarantula e GZoltar; e
- realizar experimentos com profissionais da indústria.

Tais estudos contribuirão de maneira significativa para avaliar a utilidade dessas ferramentas na indústria de software.

Referências¹

- ABREU, R. et al. Automatic software fault localization using generic program invariants. In: *Proceedings of the 2008 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2008. (SAC '08), p. 712–717. ISBN 978-1-59593-753-7. Disponível em: <http://doi.acm.org/10.1145/1363686.1363855>. Citado na página 37.
- AGRAWAL, H. *Towards Automatic Debugging of Computer Programs*. Tese (Doutorado) — Purdue University, West Lafayette, IN 47907, September 1991. Disponível em: <http://spaf.cerias.purdue.edu/Students/spyder/TR103P.pdf>. Citado na página 31.
- ARAKI, K.; FURUKAWA, Z.; CHENG, J. A general framework for debugging. *IEE Software Magazine*, v.8, n. 3, 1991. Citado 3 vezes nas páginas 17, 30 e 31.
- ARAUJO, R. P. A. de. *Scalable data-flow testing*. Dissertação (Mestrado) — Universidade de São Paulo, 2014. Disponível em: <http://www.teses.usp.br/teses/disponiveis/100/100131/tde-14112014-155259/pt-br.php>. Citado na página 64.
- BALL, T.; EICK, S. G. Software visualization in the large. *IEEE Computer Society Press Los Alamitos, CA, USA*, Los Alamitos, CA, USA, v. 29, n. 4, p. 1, April 1996. Disponível em: <http://research.microsoft.com/en-us/um/people/tball/papers/softvis.pdf>. Citado na página 38.
- BARBOSA, E. F. et al. *Introdução ao Teste de Software*. 2007. Disponível em: <http://www.inf.ufpr.br/silvia/topicos/apostilaUSP.pdf.gz>. Citado na página 17.
- BERNARDO, P. C.; KON, F. A importância dos testes automatizados. *Engenharia de Software Magazine*, p. 2, 2008. Disponível em: <https://www.ime.usp.br/~kon/papers/EngSoftMagazine-IntroducaoTestes.pdf>. Citado na página 29.
- BRINTLEY, D. W.; GALLAGHER, K. B. Program slicing. *Academic Press 1996*, p. 1–2, 1996. Disponível em: <http://www.cs.cmu.edu/afs/cs/academic/class/17654-f01/www/refs/Binkley.pdf>. Citado na página 34.
- CASERTA, P.; OLIVIER, Z. Visualization of the static aspects of software: A survey. *IEEE Transactions on Visualization and computer graphics*, p. 2, 2011. Disponível em: https://hal.inria.fr/file/index/docid/636900/filename/visualization_static_aspects_2011.pdf. Citado na página 38.
- CEMIN, C. *Visualização de Informações Aplicada à Gerência de Software*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul, 2001. Disponível em: <https://www.lume.ufrgs.br/bitstream/handle/10183/1838/000359630.pdf?sequence=1>. Citado 2 vezes nas páginas 37 e 38.
- CHAIM, M. L.; ARAUJO, R. P. A. D. An efficient bitwise algorithm for intra-procedural data-flow testing coverage. *Information Processing Letters*, 2012. Citado 3 vezes nas páginas 23, 27 e 28.
- CHAIM, M. L.; MALDONADO, J. C.; JINO, M. *Processo de Depuração Depois do Teste: Definição e Análise*. 2002. Disponível em: <http://www.infoteca.cnptia.embrapa.br/infoteca/bitstream/doc/8684/1/bolpesq5.pdf>. Citado 4 vezes nas páginas 27, 30, 31 e 34.

¹ De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

CHAN, T. W. A framework for software debugging. *Journal of Computer Information Systems*, v. 38, n. 1, p. 67–73, 1997. Disponível em: <http://www.tandfonline.com/doi/abs/10.1080/08874417.1997.11647309>. Citado na página 31.

CLIFF, N. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, American Psychological Association, v. 114, n. 3, p. 494–509, nov. 1993. ISSN 1939-1455. Disponível em: <http://search.ebscohost.com/login.aspx?direct=true&db=pdh&AN=bul-114-3-494&site=ehost-live>. Citado na página 85.

COHEN, J. *Statistical Power Analysis for the Behavioral Sciences*. [S.l.]: Lawrence Erlbaum Associates, 1988. Citado na página 85.

CRONBACH, L. Coefficient alpha and the internal structure of tests. *Psychometrika*, v. 16, n. 3, p. 297–334, 1951. Citado na página 86.

CROSSNO, P.; ROGERS, D. H.; GARASI, C. J. Case study: Visual debugging of finite element codes. *IEEE Visualization VIS - 2002*, 2002. Disponível em: <http://www.sandia.gov/~pjcross/papers/DebugFiniteElem.pdf>. Citado 3 vezes nas páginas 57, 58 e 60.

CUMMINS, R. A.; GULLONE, E. Why we should not use 5-point Likert scales: The case for subjective quality of life measurement. In: *Proceedings, Second International Conference on Quality of Life in Cities*. [S.l.: s.n.], 2000. p. 74–93. Citado na página 84.

DAVIS, F. D.; BAGOZZI, R. P.; WARSHAW, P. R. User acceptance of computer technology: A comparison of two theoretical models. *Manage. Sci.*, INFORMS, Institute for Operations Research and the Management Sciences (INFORMS), Linthicum, Maryland, USA, v. 35, n. 8, p. 982–1003, ago. 1989. ISSN 0025-1909. Citado 3 vezes nas páginas 40, 41 e 82.

DELAMARO, M. E. et al. *Introdução ao Teste de Software*. Rio de Janeiro: Elsevier Ltda., 2007. Citado 10 vezes nas páginas 17, 22, 23, 24, 25, 26, 28, 29, 33 e 34.

FELIZARDO, K. R. et al. *Revisão sistemática da literatura em engenharia de software*. 1a. ed. São Paulo: Elsevier, 2017. Citado na página 42.

GOLVEIA, C.; CAMPOS, J.; ABREU, R. Using html5 visualizations in software fault localization. In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. [S.l.: s.n.], 2013. p. 1–10. Citado 8 vezes nas páginas 39, 48, 49, 50, 58, 74, 122 e 125.

GREENLAND, S. et al. Statistical tests, p values, confidence intervals, and power: a guide to misinterpretations. *European Journal of Epidemiology*, v. 31, n. 4, p. 337–350, Apr 2016. ISSN 1573-7284. Disponível em: <https://doi.org/10.1007/s10654-016-0149-3>. Citado na página 85.

HORA, A. et al. Bug maps: A tool for the visual exploration and analysis of bugs. In: *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*. [s.n.], 2012. p. 523–526. Disponível em: <https://doi.org/10.1109/CSMR.2012.68>. Citado na página 53.

IEEE. *IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990)*. 1990. Disponível em: http://www.mit.jyu.fi/ope/kurssit/TIES462/Materiaalit/IEEE_SoftwareEngGlossary.pdf. Citado na página 22.

JONES, J. A.; BOWRING, J. F.; HARROLD, M. J. Debugging in parallel. *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, p. 16–26, 2007. Disponível em: <http://dl.acm.org/citation.cfm?id=1273468>. Citado 3 vezes nas páginas 18, 33 e 36.

JONES, J. A.; HARROLD, M. J.; STASKO, J. Visualization of test information to assist fault localization. In: *Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: ACM, 2002. (ICSE '02), p. 467–477. ISBN 1-58113-472-X. Disponível em: <http://doi.acm.org/10.1145/581339.581397>. Citado 11 vezes nas páginas 17, 18, 19, 36, 39, 51, 58, 60, 61, 62 e 122.

KAZMIER, L. J. *Estatística aplicada a administração e economia*. 4a. ed. Porto Alegre: Bookman, 2007. Citado na página 85.

LAWRANCE, J.; BOGART, C. How programmers debug, revisited: An information foraging theory perspective. v. 39, n. 2, p. 197–215, 2013. Citado 2 vezes nas páginas 32 e 61.

LEGRIS, P.; INGHAM, J.; COLLERETTE, P. Why do people use information technology? a critical review of the technology acceptance model. *Information Management*, v. 40, n. 3, p. 191 – 204, 2003. ISSN 0378-7206. Citado 2 vezes nas páginas 40 e 41.

LINTERN, R. et al. Plugging-in visualization: Experiences integrating a visualization tool with eclipse. In: *Proceedings of the 2003 ACM Symposium on Software Visualization*. New York, NY, USA: ACM, 2003. (SoftVis '03), p. 47–ff. ISBN 1-58113-642-0. Disponível em: <http://doi.acm.org/10.1145/774833.774840>. Citado na página 38.

LUIS, F. M. M. S. *Refinamento de testes para a localização de faltas*. Dissertação (Mestrado) — Universidade de Lisboa, 2012. Disponível em: http://repositorio.ul.pt/bitstream/10451/8836/1/ulfc104272_tm_Filipe_Luis.pdf. Citado na página 48.

MALDONADO, J. C. et al. Cobertura dos critérios potenciais-usos e a confiabilidade de software. *XI Simpósio Brasileiro de Engenharia de Software*, p. 379–394, 1997. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/sbes/1997/0024.pdf>. Citado na página 27.

MAROCO, J.; MARQUES, T. G. *Qual a fiabilidade do alfa de Cronbach? Questões antigas e soluções modernas?* 2006. Disponível em: <http://publicacoes.ispa.pt/index.php/lp/article/viewFile/763/706>. Citado na página 86.

MCCABE, T. J. A complexity measure. In: *Proceedings of the 2Nd International Conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976. (ICSE '76), p. 407–. Disponível em: <http://dl.acm.org/citation.cfm?id=800253.807712>. Citado na página 28.

MEGA, G.; KON, F. God: Um depurador simbólico para sistemas de objetos distribuídos. *Simpósio Brasileiro de Engenharia de Software -2005*, p. 1–3, 2005. Disponível em: http://eclipse.ime.usp.br/projects/DistributedDebugging/sbes_2005_dd.pdf. Citado na página 33.

MOTA, L. G. et al. Particionamento seguro de código. *IV Congresso Tecnológico InfoBrasil*, p. 2, 1996. Disponível em: https://ins3rt.s3.amazonaws.com/media/papers/Particionamento_seguro_de_codigo.pdf. Citado na página 34.

MUTTI, D. *Coverage Based Debugging Visualization*. Dissertação (Mestrado) — Universidade de São Paulo, 2014. Disponível em: <http://www.teses.usp.br/teses/disponiveis/100/100131/tde-15122014-230109/en.php>. Citado 6 vezes nas páginas 18, 19, 39, 63, 65 e 66.

MYERS, G. J.; BADGETT, T.; SANDLER, C. *The Art of Software Testing*. 3a. ed. New Jersey: John Wiley & Sons, Inc., 2012. Citado 4 vezes nas páginas 17, 22, 25 e 26.

NIELSEN, J. *How Many Test Users in a Usability Study?* 2012. Disponível em: <https://www.nngroup.com/articles/how-many-test-users/>. Citado na página 73.

PAN, H. *Software Debugging with Dynamic Instrumentation and Test-Based Knowledge*. Tese (Doutorado) — Purdue University, August 1993. Disponível em: <http://spaf.cerias.purdue.edu/Students/spyder/TR145P.pdf>. Citado na página 31.

PEREZ, A.; ABREU, R. Cues for scent intensification in debugging. In: *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. [S.l.: s.n.], 2013. p. 120–125. Citado 7 vezes nas páginas 18, 19, 51, 58, 60, 61 e 62.

PRESSMAN, R. *Engenharia de Software*. 6a. ed. Porto Alegre: Mc Graw Hill, 2006. Citado 3 vezes nas páginas 24, 26 e 30.

REISS, S. P. The challenge of helping the programmer during debugging. *2014 Second IEEE Working Conference on Software Visualization (VISOFT)*, IEEE Computer Society, Los Alamitos, CA, USA, v. 00, p. 112–116, 2014. Citado na página 52.

RENIERIS, M.; REISS, S. P. Fault localization with nearest neighbor queries. In: *ASE*. IEEE Computer Society, 2003. p. 30–39. ISBN 0-7695-2035-9. Disponível em: <http://dblp.uni-trier.de/db/conf/kbse/ase2003.html#RenierisR03>. Citado 2 vezes nas páginas 18 e 35.

RIBEIRO, H. L. *On the use of control- and data-flow in fault localization*. Dissertação (Mestrado) — Universidade de São Paulo, 2016. Disponível em: <http://ppgsi.each.usp.br/dissertacoes-defendidas/>. Citado 3 vezes nas páginas 63, 64 e 65.

RIBOIRA, A.; ABREU, R.; ABREU, R. The gzoltar project: A graphical debugger interface. In: *TAIC PART*. [S.l.: Springer, 2010. (Lecture Notes in Computer Science, v. 6303), p. 215–218. Citado na página 48.

ROMANO, J. et al. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys? In: *annual meeting of the Florida Association of Institutional Research*. [S.l.: s.n.], 2006. p. 1–3. Citado na página 85.

SASSO, T. D.; LANZA, M. In*bug: Visual analytics of bug repositories. In: *CSMR-WCRE*. [S.l.: IEEE Computer Society, 2014. p. 415–419. Citado na página 54.

- SHAPIRO, S. S.; WILK, M. B. An analysis of variance test for normality (complete samples). *Biometrika*, v. 52, n. 3/4, p. 591–611, Dec. 1965. Disponível em: <http://links.jstor.org/sici?sici=0006-3444%28196512%2952%3A3%2F4%3C591%3AAAQVTF%3E2.0.CO%3B2-B>. Citado na página 84.
- SNYDER, P.; LAWSON, S. Evaluating results using corrected and uncorrected effect size estimates. *The Journal of Experimental Education*, Routledge, v. 61, n. 4, p. 334–349, 1993. Disponível em: <http://dx.doi.org/10.1080/00220973.1993.10806594>. Citado na página 85.
- SOMMERVILLE, L. *Engenharia de Software*. 8a. ed. São Paulo: Pearson Education, 2007. Citado 4 vezes nas páginas 17, 22, 29 e 30.
- SOUZA, H. A. de. *Depuração de Programas Baseada em Cobertura de Integração*. Dissertação (Mestrado) — Universidade de São Paulo, 2012. Disponível em: <http://www.teses.usp.br/teses/disponiveis/100/100131/tde-08032013-162246/pt-br.php>. Citado 4 vezes nas páginas 18, 23, 35 e 36.
- SOUZA, H. A. de; CHAIM, M. L.; KON, F. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. 2016. Disponível em: <http://arxiv.org/pdf/1607.04347v1.pdf>. Citado na página 19.
- SPEARMAN, C. “General intelligence,” objectively determined and measured. *American Journal of Psychology*, v. 15, p. 201–293, 1904. Citado na página 86.
- STATISTICS, A.-T. *Sample Size Calculator*. 2016. Disponível em: <https://www.ai-therapy.com/psychology-statistics/sample-size-calculator>. Citado na página 73.
- STEINMACHER, I. et al. Overcoming open source project entry barriers with a portal for newcomers. *Proceedings of the 38th International Conference on Software Engineering*, p. 4, 2016. Disponível em: <https://ctreude.files.wordpress.com/2016/02/icse16b.pdf>. Citado na página 73.
- SULLIVAN, G.; FEINN, R. Using effect size—or why the p value is not enough. *Journal of Graduate Medical Education*, v. 4, n. 3, p. 279–282, 2012. Disponível em: <https://doi.org/10.4300/JGME-D-12-00156.1>. Citado na página 85.
- TANIMOTO, S. L. A perspective on the evolution of live programming. In: *Proceedings of the 1st International Workshop on Live Programming*. Piscataway, NJ, USA: IEEE Press, 2013. (LIVE '13), p. 31–34. ISBN 978-1-4673-6265-8. Disponível em: <http://dl.acm.org/citation.cfm?id=2662726.2662735>. Citado na página 61.
- TAYLOR, L.; TITMUSS, R.; LEBRE, C. The challenges of seamless handover in future mobile multimedia networks. *IEEE Personal Communications*, v. 6, n. 2, p. 32–37, Apr 1999. ISSN 1070-9916. Citado na página 41.
- UX, B. *Sample Size Calculator*. 2014. Disponível em: <http://blinkux.com/usability-sample-size/>. Citado na página 73.
- WARE, C. *Information Visualization Perception for Design*. San Francisco: Morgan Kaufmann Publishers, 2004. Citado na página 39.

WEISER, M. Program slicing. In: *Proceedings of the 5th International Conference on Software Engineering*. [S.l.]: IEEE Press, 1981. (ICSE '81), p. 439–449. Citado na página 33.

XIE, X. et al. Spectrum-based fault localization: Testing oracles are no longer mandatory. *11th International Conference on Quality Software 2011*, p. 432–449, 2011. Disponível em: <https://www.utdallas.edu/~ewong/documents/UTD-CS-TR-2010-02-12.pdf>. Citado 2 vezes nas páginas 35 e 37.

ZELLER, A. *Why Programs Fail: A Guide to Systematic Debugging*. 2a. ed. San Francisco: Morgan Kaufmann Publishers, 2002. Citado na página 17.

ZHAO, C. et al. In: *SSIRI*. [S.l.]: IEEE Computer Society, 2009. p. 113–122. Citado 4 vezes nas páginas 55, 56, 57 e 60.

Apêndice A – Termo de Consentimento Livre Esclarecido (TCLE)

TERMO DE CONSENTIMENTO LIVRE E ESCLARECIDO

O Sr. (a) está sendo convidado (a) como voluntário (a) a participar da pesquisa “Visualização das Informações de Depuração – Uma avaliação experimental”. Nesta pesquisa pretendemos investigar se o uso de uma ferramenta visual é mais eficaz e eficiente do que técnicas de apresentação de informações textuais na atividade de depuração. O motivo que nos leva a estudar é a necessidade do desenvolvimento de novas ferramentas que apoiem os desenvolvedores na atividade de depuração.

Para esta pesquisa adotaremos os seguintes procedimentos: O Sr. (a) deve possuir conhecimentos na área de desenvolvimento de software para participar da avaliação. Serão convidadas em torno de 32 pessoas que virão de âmbito acadêmico e profissional. No experimento serão apresentadas as ferramentas CodeForest e Jaguar. Buscamos entender qual forma de apresentação de informações é capaz de melhor auxiliar os desenvolvedores na atividade de depuração. Será avaliado se o desenvolvedor conseguiu ou não realizar a localização do defeito com cada uma das ferramentas. Para isso, serão apresentados dois defeitos aos desenvolvedores que deverão localizá-lo com uma ferramenta distinta para cada um deles. As ações dos programadores durante a depuração serão gravadas com o objetivo de identificar questões relacionadas a usabilidade das ferramentas estudadas neste projeto de pesquisa. Não pediremos nenhuma informação pessoal dos desenvolvedores que participarem do experimento no questionário respondido ao término da avaliação. Ela será focada em aspectos técnicos do experimento, como se foi possível realizar a localização do defeito, em qual trecho do programa ele foi encontrado e sua possível correção. Perguntaremos também se ele teve algum problema com o ambiente criado ou com a utilização de cada uma das aplicações estudadas. Os riscos envolvidos na pesquisa e seus possíveis tratamentos consistem em verificar especialmente os aspectos iluminação e ruído durante a realização da avaliação tratado a partir da escolha do laboratório adequado para o experimento. Verificar se o experimento exige do participante um desempenho acima de suas habilidades, em que será oferecida total liberdade para que o desenvolvedor abandone o experimento ou utilize qualquer outro recurso no ambiente criado. Prover recursos para que o indivíduo não fique envergonhado em casos de situações de baixo desempenho, o experimento será sigiloso e nenhuma informação pessoal do programador será divulgada. A pesquisa contribuirá para avaliar se o uso de metáforas visuais irá trazer melhorias a área de Engenharia de Software e poderá ser utilizada como referência para a elaboração de novos estudos que tenham como objetivo o desenvolvimento de ferramentas visuais que auxiliem os desenvolvedores na atividade de depuração.

Para garantir o sigilo durante todo o estudo, após o Sr. (a) aceitar participar da avaliação os pesquisadores entrarão em contato através de uma mensagem eletrônica. Após o experimento o Sr. (a) deverá preencher um questionário contendo informações básicas relacionadas a sua experiência profissional na área de desenvolvimento de software e dados que nos auxiliem na avaliação dos resultados deste estudo; mas sem a solicitação de qualquer informação pessoal. O sigilo será garantido pois o Sr. (a) receberá um código de identificação que será utilizado durante todo o experimento e posterior avaliação dos resultados pelos pesquisadores.

Para participar deste estudo o Sr (a) não terá nenhum custo, nem receberá qualquer vantagem financeira. O Sr. (a) terá o esclarecimento sobre o estudo em qualquer aspecto que desejar e estará livre para participar ou recusar-se a participar. Poderá retirar seu consentimento ou interromper a participação a qualquer momento. A sua participação é voluntária e a recusa não acarretará em qualquer penalidade ou modificação na forma em que o Sr. (a) é atendido (a). Os resultados da pesquisa estarão à sua disposição quando finalizada. Em nenhum momento o Sr. (a) precisará se identificar para a realização deste estudo, pois não serão solicitadas informações pessoais do participante.

Este termo de consentimento encontra-se impresso em duas vias originais, sendo que uma será arquivada pelo pesquisador responsável, na **Universidade de São Paulo** e a outra será fornecida ao Sr. (a). Os dados e instrumentos utilizados na pesquisa ficarão arquivados com o pesquisador responsável por um período de 5 (cinco) anos, e após esse tempo serão destruídos. Em caso de dúvidas e problemas o Sr. (a) poderá se dirigir diretamente ao Comitê de Ética em Pesquisa localizado na Avenida Arlindo Bettio, 1000, localizado no bairro de Ermelino Matarazzo, São Paulo - SP, CEP: 03828-000, Salas T14 - I1. Além disso, o Sr. (a) pode entrar em contato com o Comitê de Ética da Universidade de São Paulo, no telefone 011 3091-1046 ou por email através do seguinte endereço cep-each@usp.br.

Eu, _____, fui informado (a) dos objetivos da pesquisa “Visualização das Informações de Depuração – Uma avaliação experimental”, de maneira clara e detalhada e esclareci minhas dúvidas. Sei que a qualquer momento poderei solicitar novas informações e modificar minha decisão de participar se assim o desejar.

Declaro que concordo em participar. Recebi uma via original deste termo de consentimento livre e esclarecido e me foi dada à oportunidade de ler e esclarecer as minhas dúvidas.

São Paulo, _____ de _____ de 20__.

Assinatura do Participante

Assinatura do (a) Pesquisador (a)

Nome do Pesquisador Responsável: Fábio Pereira da Silva

Endereço: Rua Agenor Rocha, 144

CEP: 08062-220

Fone (11) 98657-0334

E-mail: fabio.dasilva@usp.br

Apêndice B – Apresentação

Depuração baseada em Cobertura de Código

Fábio Pereira da Silva

Problema

Altos Custos

- As atividades de teste de software e depuração são responsáveis por mais de 50% dos custos totais de um projeto.

Poucas evoluções

- Não houve mudanças significativas na forma como a atividade de depuração é realizada ao longo das últimas décadas e, na prática, a depuração ainda ocorre manualmente.

Técnicas não escaláveis

- Recentemente, foram desenvolvidas técnicas com o objetivo de auxiliar a atividade de depuração; porém não são escaláveis para uso em ambientes industriais.

Atividade de maior consumo de tempo no processo de desenvolvimento

Consumo diário de tempo durante a depuração

Depuração

Corolário do teste bem sucedido	<ul style="list-style-type: none">• Teste que detecta a presença de defeitos.
Tarefa mais difícil	<ul style="list-style-type: none">• Localizar defeitos. Como depurar programas para localizar defeitos de maneira mais eficaz e menos custosa.
Corrigir Defeitos	<ul style="list-style-type: none">• Como alterar o código para eliminar defeitos sem introduzir outros.

Depuradores Simbólicos

- Depuradores simbólicos permitem:
 - Inspeccionar o estado da pilha de execução
 - Inspeccionar os valores das variáveis
 - Realizar a execução do programa passo a passo
 - Alterar valores das variáveis
 - Avaliar expressões
 - Monitorar posições de memória (watchpoints).

L	B	Comando
-	-	int max(int array [], int length)
-	1	{
1	1	int i = 0;
2	1	int max = array[++i]; //array[i++];
3	2	while(i < length){
-	3	{
4	3	if(array[i] > max)
5	4	max = array[i];
6	5	i = i + 1;
-	5	}
7	6	return max;
-	6	}

Casos de Testes para o método Max

T_n	caso de teste	valor esperado	valor obtido
t_1	$\max([1,2,3], 3)$	3	3
t_2	$\max([5,5,6], 3)$	6	6
t_3	$\max([2,1,10], 3)$	10	10
t_4	$\max([4,3,2], 3)$	4	3
t_5	$\max([4], 1)$	4	exceção

Tabela de Cobertura do Método Max

comando	t_1	t_2	t_3	t_4	t_5	Ochiai
int i = 0;	•	•	•	•	•	0.63
int max = array[++i]; // array[i++]	•	•	•	•	•	0.63
while(i < length)	•	•	•	•		0.35
if(array[i] > max)	•	•	•	•		0.35
max = array[i];	•	•	•			0.00
i++;	•	•	•	•		0.35
return max;	•	•	•	•		0.35
	✓	✓	✓	✗	✗	

Técnicas de Depuração

- Depuração baseada em cobertura de código
 - Associa um nível de suspeição a linhas de código e a outras entidades do programa (métodos, classes e pacotes).
- Visualização de Informações de Depuração
 - Realiza a apresentação dos resultados obtidos na depuração baseada em cobertura de código em metáforas visuais.

$$Ochiai = \frac{c_{ef}}{\sqrt{(c_{ef} + c_{nf})(c_{ef} + c_{ep})}}$$

- Cep: Quantidade de vezes em que o comando não foi executado por um caso de teste de sucesso.
- Cnf: Indica a quantidade de vezes em que comando não foi executado por um caso de teste que falhou.
- Cef: Quantidade de vezes em que o comando foi executado por um caso de teste que falhou.

Visualização de Informações

Definição

- É uma área recente que procura criar novos algoritmos e técnicas de visualização para grandes quantidades de informações.

Problema

- A falta de visibilidade das informações durante o desenvolvimento de software colabora para a baixa produtividade dos programadores.

Cognição Humana

- Com ela é possível explorar o potencial da cognição humana e habilidades de percepção que não são possíveis apenas com informações textuais.

AbstractTokenizer.java

```

@param companion information object associated with this keyword
//
public void addKeyword(String keyword, Object companion, int flags) {
    // normalize flags
    int commonFlags = getParseFlags();

    if (flags != commonFlags) {
        if ((commonFlags & F_KEYWORDS_CASE) == 0) {
            if ((flags & F_KEYWORDS_CASE) != 0) {
                flags |= F_KEYWORDS_CASE;
            }
        } else {
            if ((flags & (F_KEYWORDS_NO_CASE | F_NO_CASE)) != 0) {
                flags &= ~F_KEYWORDS_CASE;
            }
        }
    }
}

```

Problems Javadoc Declaration Method View

Class.Method	Score
AbstractTokenizer.isKeyword(int,int)	1.00
AbstractTokenizer.addKeyword(String,Object,int)	0.97
AbstractTokenizer.addKeyword(String,Object)	0.97
AbstractTokenizer.addKeyword(String)	0.88
AbstractTokenizer.addString(String,String,Object,int)	0.78
AbstractTokenizer.addString(String,String,String)	0.78
AbstractTokenizer.addString(String,String,String,Object)	0.78
SortedArray.addSpecialSequence(TokenizerProperty)	0.65
AbstractTokenizer.addSpecialSequence(TokenizerProperty)	0.61
SortedArray.searchIndex(char,int[])	0.61
SortedArray.searchBinary(String,int,boolean)	0.59
AbstractTokenizer.indexInSet(char,String)	0.56
AbstractTokenizer.addBlockComment(String,String,String,Object,int)	0.54
AbstractTokenizer.addBlockComment(String,String)	0.54

Search:

Min: 0.0 Current min score: 0.0 Max: 1.0

Java EE - jtopas-0.4/src/de/susebox/java/util/AbstractTokenizer.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help

Project Explorer AbstractTokenizer.java

New
Go Into
Show In
Copy
Copy Qualified Name
Paste
Delete
Remove from Context
Build Path
Refactor
Import...
Export...
Refresh
Close Project
Close Unrelated Projects
Mark as Deployable
Validate
Show in Remote Systems view
Run As
Debug As
Profile As
Coverage As
Restore from Local History...
Code Forest
Jaguar
Team
Compare With
Configure
Source
Properties

the current read position contains a whitespace. If so retrieve sequence of whitespaces.
then add information to this buffer
TokenizerException failure while reading data from the input stream
DO>true</CODE> if a whitespace sequence was found
DO>false</CODE> otherwise
clean test4Whitespace(Token token) throws TokenizerException {
 if (currentReadPos < currentWritePos || readMoreData() > 0) {
 whitespace(_inputBuffer[_currentReadPos]);
 whitespace(token);
 true;
 }
};
ing identified a whitespace, this method continues to read data
etects a non-whitespace.

is expected to fill in the type and the length of the whitespace
g the methods {@link Token#setType} and {@link Token#setLength}
then add information to this buffer
TokenizerException failure while reading data from the input stream
d completeWhitespace(Token token) throws TokenizerException {
 = _currentReadPos + 1; // the first whitespace we have already
 le = _currentWritePos - start;
 = readWhitespaces(rangeStart + start, available);
 }
};
g Servers Data Source Explorer Snippets Perform Analysis

Outline Task List

- getChar(int): char
- getCharUnchecked(int): char
- addToken(AbstractTokenizer): void
- switchTo(AbstractTokenizer): void
- isWhitespace(char): boolean
- readWhitespaces(int, int): int
- readString(int, int): int
- isSeparator(char): boolean
- isSequenceCommentOrString(): TokenizerProperty
- isKeyword(int, int): TokenizerProperty
- addSpecialSequence(TokenizerProperty)
- searchBinary(String): TokenizerProperty
- getBaseTokenizer(AbstractTokenizer)
- readMoreData(): int
- synchronizeAll(int): void
- test4Normal(Token): boolean
- test4Whitespace(Token): boolean
- completeWhitespace(Token): void
- isWhitespace(int): boolean
- test4Separator(Token): boolean
- completeSeparator(Token): void
- isSeparator(int): boolean
- test4SpecialSequence(Token): boolean
- completeSpecialSequence(Token): void
- isSpecialSequence(int): boolean
- completeLineComment(int): int

Perform analysis
Remove analysis
View as a Code Forest
Star Heuristic
thai Heuristic
iczynski2 Heuristic
Con Heuristic

1489023300790 - DRT Heuristic

Atividade proposta

Ambiente

- A atividade será realizada na máquina local que o participante estiver utilizando.

Programas selecionados

- Foram escolhidos dois programas reais de código aberto desenvolvidos em Java: JSoup e XStream. O JSoup é uma biblioteca Java para análise, manipulação e extração de dados HTML. O XStream é uma biblioteca para a serialização de objetos XML.

Composição

- Experiência de uso das ferramentas, realização da atividade e questionário final.

Informações

- No diretório “Materiais de Treinamento” há tutoriais para uso das ferramentas Jaguar e CodeForest.
- Sempre ao término de cada atividade clique em “Stop Jaguar” ou “Remove Analysis”
- Você poderá utilizar qualquer recurso disponível no Eclipse para a realização da atividade
- O link do questionário final está disponível no diretório “Questionários”.
- Ao término da última atividade não desligue o computador.

Apêndice C – Material de treinamento

C.1 Material de treinamento - Jaguar Linha

Sobre a Jaguar Linha

A Jaguar é uma ferramenta de localização de defeitos para o Eclipse. A Jaguar apresenta uma lista com os comandos mais suspeitos de conter defeitos. Para gerar a lista, a Jaguar usa os resultados da execução dos testes JUnit.

Figura 1: Plugin da Jaguar para o Eclipse.

A Figura 1 mostra a Jaguar no lado direito do Eclipse. A Figura 2 mostra detalhes da ferramenta - A área 1 contém duas colunas: *Comando* e *Valor*. A coluna *Comando* possui a lista com os comandos suspeitos de conter um defeito. A coluna *Valor* mostra o valor de suspeição de cada comando. Esse valor indica a probabilidade que cada comando tem de conter o defeito. Quanto maior o valor, mais suspeito ele é. Os comandos estão ordenados em ordem decrescente de valor.

As cores representam quatro níveis de valores: vermelho representa os comandos mais suspeitos, laranja são comandos mais suspeitos após os vermelhos, amarelos são os comandos com chance moderada de conter o defeito, e os comandos em verde são os menos suspeitos.

Navegação

Ao clicar em um comando na Jaguar, o arquivo que contém esse comando será aberto na área de edição do Eclipse (veja Figura 1). A Jaguar possui dois filtros: um controle *slider* (veja Figura 2, área 3) para filtrar os **comandos** por valor de suspeição; e uma busca textual

(veja Figura 2, área 2), para selecionar **comandos** que contêm um termo de interesse.

Comando	Valor
int commonFlags = getParseFlags	1.00
if (flags != commonFlags) {	1.00
if ((commonFlags & F_KEYWORDS	1.00
table = _keywords[1];	1.00
String keyword = new String(_input	0.98
if (_keywords[0] != null) {	0.98
if (prop == null && _keywords[1] !=	0.98
keyword = keyword.toUpperCase(	0.98
if ((flags & F_KEYWORDS_CASE) !=	0.97
if ((flags & (F_KEYWORDS_NO_CA	0.97
flags &= ~F_KEYWORDS_CASE;	0.97
}	0.97
}	0.97
if ((flags & Tokenizer.F_KEYWORD	0.97
if (_keywords[1] == null) {	0.97
_keywords[1] = new HashMap();	0.97
keyword = keyword.toUpperCase(	0.97
table.put(keyword, new Tokenizer	0.97
new String[] { keyword },	0.97
companion, flags));	0.97
addString(start, end, escape, null)	0.78
_array.add(idx, new SpecialSeque	0.65
} else {	0.65
if (_sequences[1] == null) {	0.61

Busca:

Min: 0.0 Valor mínimo atual: 0.0 Max: 1.0

Figura 2: Detalhes da Jaguar

Teste a Jaguar

Explore a ferramenta antes de começar a realizar os experimentos. O programa **jtopas** foi disponibilizado para uso nesse exemplo.

- Clique com o botão direito do mouse no programa **jtopas**.
- Escolha a opção **Jaguar > Run Jaguar**.
- O plugin será aberto no lado direito do Eclipse.
- Clique nos comandos para ver como funciona a Jaguar.
- O defeito está no comando da **linha 1559**:

String keyword = new String(_inputBuffer, startingAtPos - _rangeStart, length - 1);
 que fica no método **isKeyword(int, int)** da classe **AbstractTokenizer**. O valor atribuído à variável **keyword** deveria ser **length** ao invés de **length - 1**.

C.2 Material de treinamento - Jaguar método

Sobre a Jaguar Método

A Jaguar é uma ferramenta de localização de defeitos para o Eclipse. A Jaguar apresenta uma lista com os métodos mais suspeitos de conter defeitos. Para gerar a lista, a Jaguar usa os resultados da execução dos testes JUnit.

Figura 1: Plugin da Jaguar para o Eclipse.

A Figura 1 mostra a Jaguar no lado direito do Eclipse. A Figura 2 mostra detalhes da ferramenta - A área 1 contém duas colunas: *Classe.Método* e *Valor*. A coluna *Classe.Método* mostra os métodos mais suspeitos de conter um defeito. Cada método é precedido pelo nome de sua respectiva classe. A coluna *Valor* mostra o valor de suspeição de cada método. Esse valor indica a probabilidade que cada método tem de conter o defeito. Quanto maior o valor, mais suspeito ele é. Os métodos estão ordenados em ordem decrescente de valor.

As cores representam quatro níveis de valores: vermelho representa os métodos mais suspeitos, laranja são métodos mais suspeitos após os vermelhos, amarelos são os métodos com chance moderada de conter o defeito, e os métodos em verde são os menos suspeitos.

Navegação

Ao clicar em um método na Jaguar, o arquivo que contém esse método será aberto na área de edição do Eclipse (veja Figura 1). A Jaguar possui dois filtros: um controle *slider* (veja Figura 2, área 3) para filtrar os métodos por valor de suspeição; e uma busca textual (veja Figura 2, área 2), para selecionar métodos que contenham um termo de interesse.

Figura 2: Detalhes da Jaguar

Teste a Jaguar

Explore a ferramenta antes de começar a realizar os experimentos. O programa **jtopas** foi disponibilizado para uso nesse exemplo.

- Clique com o botão direito do mouse no programa **jtopas**.
- Escolha a opção **Jaguar > Run Jaguar**.
- O plugin será aberto no lado direito do Eclipse.
- Clique nos comandos para ver como funciona a Jaguar.
- O defeito está na **linha 1559**:

String keyword = new String(_inputBuffer,startingAtPos-_rangeStart,length-1);
 que fica no método **isKeyword(int,int)** da classe **AbstractTokenizer**. O valor atribuído à variável **keyword** deveria ser **length** ao invés de **length - 1**.

C.3 Material de treinamento - CodeForest

Sobre a CodeForest

A CodeForest é uma ferramenta visual de depuração. Ela demonstra os elementos mais suspeitos do programa através de uma floresta de cactus tridimensional.

Figura 1: Plugin da CodeForest para o Eclipse

A Figura 1 mostra a CodeForest ao lado direito do Eclipse. Na CodeForest as classes são representadas como cactus, os métodos como galhos e as linhas do programa avaliadas como espinhos de um galho. Quanto maior for o valor de suspeição do elemento do programa avaliado maior é a probabilidade do defeito estar presente nele.

As cores são representadas em quatro níveis de valores: vermelho representa os comandos mais suspeitos, laranja são comandos mais suspeitos após os itens apresentados em vermelho, amarelo os elementos com uma probabilidade intermediária e em verde os elementos com menor probabilidade de conter o defeito.

Para utilizar a CodeForest primeiramente clique com o botão direito no projeto, vá até o ícone da CodeForest, clique em "Perform Analysis" e selecione o relatório de erros desejado.

Figura 2: Acessando a opção de Análise de Performance

Figura 3: Selecionando o relatório de erros

Em seguida, clique novamente com o botão direito no projeto, navegue até o ícone da CodeForest e clique em “View as CodeForest” para que a floresta de cactus seja apresentada.

Figura 4: Solicitando a geração da floresta de cactus.

Figura 5: Floresta de cactus da CodeForest

Ao clicar no classe (cactu), método (galho) ou linha (espinhos de um galho) o desenvolvedor é redirecionado para o trecho solicitado no código fonte. As linhas que foram coloridas pelo teste são demonstradas de acordo com o seu valor de suspeição, conforme na Figura 1.

O desenvolvedor pode aumentar ou diminuir o número de classes (cactus) que ele deseja que sejam apresentados de acordo com a sua necessidade. Por padrão são apresentados no máximo 8 classes a partir do valor de suspeição de 0.8. Para isso basta

desmarcar os valores pré estabelecidos, modificá-los de acordo com a sua necessidade, selecionar a opção “New Inspection” e clicar em “Run”

Figura 6: Aumentando o número de classes e o valor de suspeição considerados na busca

Os cactus na CodeForest são apresentados da esquerda para a direita de acordo com o seu valor de suspeição. **O tamanho do cactu é determinado pelo número de métodos que ele possui. O tamanho do galho é determinado pelo número de linhas que ele possui.**

O desenvolvedor pode aumentar ou diminuir o tamanho dos elementos da floresta. Para isso são utilizadas as setas do teclado. **Seta para cima aumentará o tamanho da floresta. Setas para baixo diminuirá o tamanho da floresta. Setas para a esquerda irá posicionar a floresta a esquerda. Setas para direita irá posicionar a floresta para a direita.**

Figura 7: CodeForest maximizada pelo uso de setas

Por último a CodeForest oferece um Roodmap em que o desenvolvedor pode aumentar ou diminuir o filtro de valores de suspeição desejados para a localização do defeito.

Figura 8: Filtrando o intervalo de suspeição considerado

Após a aplicação do filtro no Roodmap a CodeForest irá se adequar automaticamente a requisição realizada.

Figura 9: Floresta de cactus após a aplicação do Roodmap

Apêndice D – Questionário

Experimento sobre o uso de ferramentas visuais e textuais na atividade de depuração.

1. Qual é o seu ID?

2. Em qual grupo você foi alocado para o experimento?

- ☐ Grupo 1
- ☐ Grupo 2
- ☐ Grupo 3
- ☐ Grupo 4
- ☐ Grupo 5
- ☐ Grupo 6
- ☐ Grupo 7
- ☐ Grupo 8

3. Em qual semestre do curso você está?

- ☐ 3º semestre
- ☐ 4º semestre
- ☐ 5º semestre
- ☐ 6º semestre
- ☐ Já sou graduado

4. Há quantos anos você programa? (Experiência educacional e profissional em qualquer linguagem).

- ☐ 0
- ☐ 1
- ☐ 2
- ☐ 3
- ☐ 4
- ☐ 5
- ☐ 6
- ☐ 7
- ☐ 8
- ☐ 9
- ☐ 10
- ☐ Mais de 10 Anos

5. Há quantos anos você programa profissionalmente? (Considerando qualquer linguagem).

154

- ☐ 0
- ☐ 1
- ☐ 2
- ☐ 3
- ☐ 4
- ☐ 5
- ☐ 6
- ☐ 7
- ☐ 8
- ☐ 9
- ☐ 10
- ☐ Mais de 10 anos

6. Há quantos anos você programa em Java?

- ☐ 0
- ☐ 1
- ☐ 2
- ☐ 3
- ☐ 4
- ☐ 5
- ☐ 6
- ☐ 7
- ☐ 8
- ☐ 9
- ☐ 10
- ☐ Mais de 10 anos

7. Como você avalia seu atual nível de conhecimento em Java?

- ☐ Não tenho conhecimentos.
- ☐ Básico - Uso esporadicamente para desenvolver pequenos programas.
- ☐ Intermediário - Uso com frequência moderada para o desenvolvimento de alguns programas grandes.
- ☐ Avançado - Uso constantemente para o desenvolvimento de diversos programas.

8. Há quantos anos você programa usando o Eclipse?

- ☐ 0
- ☐ 1
- ☐ 2
- ☐ 3
- ☐ 4
- ☐ 5

- ☐ 6
- ☐ 7
- ☐ 8
- ☐ 9
- ☐ 10
- ☐ Mais de 10 anos

9. Como você avalia seu atual nível de conhecimento no Eclipse ou Netbeans?

- ☐ Não tenho conhecimento.
- ☐ Básico - Uso esporadicamente para desenvolver pequenos programas.
- ☐ Intermediário - Uso com frequência moderada para o desenvolvimento de alguns programas grandes.
- ☐ Avançado - Uso constantemente para o desenvolvimento de diversos programas.

10. Há quantos anos você faz testes usando JUnit ou outro framework de testes automatizados?

- ☐ 0
- ☐ 1
- ☐ 2
- ☐ 3
- ☐ 4
- ☐ 5
- ☐ 6
- ☐ 7
- ☐ 8
- ☐ 9
- ☐ 10
- ☐ Mais de 10 anos

Programa e versão da Jaguar utilizados

11. Qual projeto você depurou usando a Jaguar?

- ☐ JSoup.
- ☐ XStream.

12. Qual versão da Jaguar você utilizou?

- ☐ Jaguar Linha.
- ☐ Jaguar Método.

13. Você pode explicar o que causou o defeito do programa jsoup? (Caso não saiba responda "Não sei").

14. Qual é o nome da classe que contém o defeito do jsoup? (Caso não saiba responda "Não sei").

15. Qual é o nome do método que contém o defeito do jsoup? (Caso não saiba responda "Não sei").

16. Qual é o número de linha onde está o defeito do jsoup? (Caso não saiba responda "Não sei").

17. Por favor descreva como você poderia corrigir o defeito do jsoup. (Caso não saiba responda "Não sei").

- 18. Qual é a dificuldade de encontrar o defeito do jsoup?**

1 2 3 4 5 6 7

Fácil ○ ○ ○ ○ ○ ○ ○ Difícil

- 19. Você já teve algum contato prévio com o código do jsoup?**

☐ Sim☐ Não

Localização de Defeitos - XStream

- 20. Você pode explicar o que causou o defeito do programa xstream?**

- 21. Qual é o nome da classe que contém o defeito do xstream?**

- 22. Qual é o nome do método que contém o defeito do xstream?**

- 23. Qual é o número de linha onde está o defeito do xstream?**

- 24. Por favor descreva como você poderia corrigir o defeito do xstream.**

- 25. Qual é a dificuldade de encontrar o defeito do xstream?**

1 2 3 4 5 6 7

Fácil ○ ○ ○ ○ ○ ○ Difícil

35. Que outra informação você gostaria de ver na Jaguar?

36. Você tem outros comentários, observações ou sugestões?

Localização de defeitos - Opinião sobre a CodeForest

37. A CodeForest ajudou você a encontrar o defeito?

- ☐ Eu encontrei o defeito sem usar a CodeForest.
- ☐ A CodeForest foi pouco útil para encontrar o defeito.
- ☐ A CodeForest foi útil para encontrar o defeito.
- ☐ A CodeForest foi muito útil para encontrar o defeito.
- ☐ Eu não encontraria o defeito sem usar a CodeForest.

38. Descreva como a CodeForest ajudou você a encontrar o defeito.

39. Qual informação fornecida pela CodeForest foi mais útil durante a depuração?

40. A organização das informações fornecida pela CodeForest foi útil?

	1	2	3	4	5	6	7	
Inútil	☐	☐	☐	☐	☐	☐	☐	Útil

41. Qual é a sua opinião sobre a representação visual fornecida pela CodeForest?

	1	2	3	4	5	6	7	
Inútil	☐	☐	☐	☐	☐	☐	☐	Útil

42. Qual é a sua opinião sobre a interação com o código fonte por meio da floresta de cactus fornecida pela CodeForest?

	1	2	3	4	5	6	7	
Inútil	☐	☐	☐	☐	☐	☐	☐	Útil

43. Qual é a sua opinião sobre o ranking fornecido pela CodeForest?

	1	2	3	4	5	6	7	
Inútil	☐	☐	☐	☐	☐	☐	☐	Útil

44. Qual é a sua opinião sobre o filtro de busca textual fornecido pela CodeForest?

[illegible]

45. Que outra informação você gostaria de ver na CodeForest?

46. Você tem outros comentários, observações ou sugestões?

Percepção de utilidade e de facilidade de uso - Jaguar

47. Supondo que eu tenha acesso à Jaguar, eu pretendo usá-la.

48. Dado que eu tive acesso à Jaguar, eu imagino que a usaria.

	1	2	3	4	5	6	7	
Concordo totalmente	☐	☐	☐	☐	☐	☐	☐	Discordo totalmente

49. Usar a Jaguar melhorou meu desempenho durante a tarefa de depuração.

	1	2	3	4	5	6	7	
Concordo totalmente	☐	☐	☐	☐	☐	☐	☐	Discordo totalmente

50. Usar a Jaguar durante a tarefa depuração aumentou minha produtividade.

[illegible]

51. Usar a Jaguar melhorou minha eficácia durante a atividade de depuração.

	1	2	3	4	5	6	7	
Concordo totalmente	☐	☐	☐	☐	☐	☐	☐	Discordo totalmente

52. De modo geral, eu achei a Jaguar útil para realizar a tarefa de depuração.

	1	2	3	4	5	6	7	
Concordo totalmente	☐	☐	☐	☐	☐	☐	☐	Discordo totalmente

53. Aprender a usar a Jaguar foi fácil para mim.

	1	2	3	4	5	6	7	
Concordo totalmente	☐	☐	☐	☐	☐	☐	☐	Discordo totalmente

54. Interagir com a Jaguar não exigiu muito do meu esforço mental.

	1	2	3	4	5	6	7	
Concordo totalmente	☐	☐	☐	☐	☐	☐	☐	Discordo totalmente

55. De modo geral, eu achei a Jaguar fácil de usar.

	1	2	3	4	5	6	7	
Concordo totalmente	☐	☐	☐	☐	☐	☐	☐	Discordo totalmente

56. Eu achei fácil contar com a Jaguar para realizar a tarefa de depuração.

	1	2	3	4	5	6	7	
Concordo totalmente	☐	☐	☐	☐	☐	☐	☐	Discordo totalmente

Percepção de utilidade e de facilidade de uso - CodeForest

57. Supondo que eu tenha acesso à CodeForest, eu pretendo usá-la.

[illegible]

58. Dado que eu tive acesso à CodeForest, eu imagino que a usaria.

	1	2	3	4	5	6	7	
Concordo totalmente	☐	☐	☐	☐	☐	☐	☐	Discordo totalmente

59. Usar a CodeForest melhorou meu desempenho durante a tarefa de depuração.

	1	2	3	4	5	6	7	
Concordo totalmente	☐	☐	☐	☐	☐	☐	☐	Discordo totalmente

60. Usar a CodeForest durante a tarefa depuração aumentou a minha produtividade.

[illegible]

61. Usar a CodeForest melhorou minha eficácia durante a atividade de depuração.

	1	2	3	4	5	6	7	
Concordo totalmente	☐	☐	☐	☐	☐	☐	☐	Discordo totalmente

62. De modo geral, eu achei a CodeForest útil para realizar a tarefa de depuração.

[illegible]

63. Aprender a usar a CodeForest foi fácil para mim.

[illegible]

64. Interagir com a CodeForest não exigiu muito do meu esforço mental.

[illegible]

65. De modo geral, eu achei a CodeForest fácil de usar.

[illegible]

66. Eu achei fácil contar com a CodeForest para realizar a tarefa de depuração.

	1	2	3	4	5	6	7	
Concordo totalmente	☐	☐	☐	☐	☐	☐	☐	Discordo totalmente

Conhecimento em depuração

67. Quais das seguintes opções abaixo você usa para depurar programas?

- ☐ Breakpoints.
- ☐ Comandos de print.
- ☐ Resultados dos testes.
- ☐ Stack traces.
- ☐ Bug reports.
- ☐ Outra: _____

68. Você tem experiência prévia usando alguma técnica de localização de defeitos? (Uma técnica de localização de defeitos é aquela que automaticamente indica trechos de código suspeitos de conter defeitos).

- ☐ Sim
- ☐ Não

69. Se você respondeu sim na questão acima, diga qual técnica de localização de defeitos você usa.

Anexo A – Submissão do projeto no Comitê de Ética e Pesquisa

USP - ESCOLA DE ARTES,
CIÊNCIAS E HUMANIDADES
DA UNIVERSIDADE DE SÃO
PAULO

COMPROVANTE DE ENVIO DO PROJETO

DADOS DO PROJETO DE PESQUISA

Título da Pesquisa: Visualização das Informações de Depuração - Uma avaliação experimental

Pesquisador: FABIO PEREIRA DA SILVA

Versão: 2

CAAE: 62033416.2.0000.5390

Instituição Proponente: UNIVERSIDADE DE SAO PAULO

DADOS DO COMPROVANTE

Número do Comprovante: 119253/2016

Patrocinador Principal: UNIVERSIDADE DE SAO PAULO

Informamos que o projeto Visualização das Informações de Depuração - Uma avaliação experimental que tem como pesquisador responsável FABIO PEREIRA DA SILVA, foi recebido para análise ética no CEP USP - Escola de Artes, Ciências e Humanidades da Universidade de São Paulo - EACH/USP em 16/11/2016 às 19:08.

Endereço: Av. Arlindo Bétio, nº 1000

Bairro: Ermelino Matarazzo

CEP: 03.828-000

UF: SP

Município: SAO PAULO

Telefone: (11)3091-1046

E-mail: cep-each@usp.br