



UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO
MESTRADO ACADÊMICO EM SISTEMAS E COMPUTAÇÃO

PerfMiner Visualizer: uma ferramenta para análise da evolução do atributo de qualidade de desempenho em sistemas de software

Leo Moreira Silva

Natal-RN
Julho de 2017

Leo Moreira Silva

PerfMiner Visualizer: uma ferramenta para análise da evolução do atributo de qualidade de desempenho em sistemas de software

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Sistemas e Computação do Centro de Ciências Exatas e da Terra da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção do grau de Mestre em Sistemas e Computação.

Linha de pesquisa:
Engenharia de Software

Orientador

Prof. Dr. Uirá Kulesza

Coorientador

Prof. Dr. Felipe Pinto

PPGSC – PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO
CCET – CENTRO DE CIÊNCIAS EXATAS E DA TERRA
UFRN – UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

Natal-RN

Julho de 2017

PerfMiner Visualizer: uma ferramenta para análise da evolução do atributo de qualidade de desempenho em sistemas de software

Autor: Leo Moreira Silva

Orientador(a): Prof. Dr. Uirá Kulesza

Coorientador(a): Prof. Dr. Felipe Pinto

RESUMO

Durante o processo de manutenção e evolução de um sistema de software, o mesmo pode sofrer diversas modificações, as quais podem trazer consequências negativas, diminuindo a sua qualidade e aumentando sua complexidade. Essa deterioração também pode afetar o desempenho dos sistemas ao longo do tempo. Assim, sem o devido acompanhamento, o atributo de qualidade de desempenho pode deixar de ser atendido adequadamente. A área de visualização de software propõe o uso de técnicas cujo objetivo é melhorar o entendimento do software e tornar mais produtivo o seu processo de desenvolvimento. Neste contexto, este trabalho apresenta uma ferramenta de visualização de desvios de desempenho de evoluções subsequentes de um sistema de software com o intuito de auxiliar a análise da evolução do desempenho entre versões de um software. A ferramenta permite, através de visualizações de grafos de chamadas e sumarização de cenários, que desenvolvedores e arquitetos possam identificar cenários e métodos que tiveram variações no seu desempenho, inclusive as potenciais causas desses desvios através dos *commits*. O trabalho também apresenta um estudo empírico que avalia o uso da ferramenta aplicando-a em 10 versões de evolução de 2 sistemas *open source* de domínios diferentes e submetendo questionários online para obter feedback dos seus desenvolvedores e arquitetos. Os resultados do estudo conduzido trazem evidências preliminares da eficácia das visualizações providas pela ferramenta em comparação com dados tabulares. Além disso, o algoritmo de supressão de nós da visualização do grafo de chamadas foi capaz de reduzir entre 73,77% e 99,83% a quantidade de nós a serem exibidos para o usuário, permitindo que ele possa identificar mais facilmente as possíveis causas das variações.

Palavras-chave: visualização de software, evolução de software, desempenho.

PerfMiner Visualizer: a tool for the analysis of performance quality attribute evolution in software systems

Author: Leo Moreira Silva

Advisor: Prof. Dr. Uirá Kulesza

Co-advisor: Prof. Dr. Felipe Pinto

ABSTRACT

During the maintenance and evolution process of a software system, it can undergo several modifications, which can have negative consequences, reducing its quality and increasing its complexity. This deterioration can also affect system performance over time. Thus, without due monitoring, the performance quality attribute may no longer be adequately met. The software visualization area proposes the use of techniques whose objective is to improve the understanding of the software and to make its development process more productive. In this context, this work presents a tool to visualize performance deviations from subsequent evolutions of a software system to assist the analysis of performance evolution between software versions. The tool allows, through call graph and scenario summarization visualizations, that developers and architects can identify scenarios and methods that have had variations in their performance, including the potential causes of such deviations through commits. This work also presents an empirical study that evaluates the use of the tool by applying it to 10 evolutionary versions of 2 open source systems from different domains and by submitting online questionnaires to obtain feedback from its developers and architects. The results of the conducted study bring preliminary evidence of the effectiveness of visualizations provided by the tool compared to tabular data. In addition, the nodes suppression algorithm of the call graph visualization was able to reduce between 73.77% and 99.83% the number of nodes to be displayed to the user, allowing him to be able to identify more easily the possible causes of variations.

Keywords: software visualization, software evolution, performance.

Lista de figuras

1	EvoSpaces.	p. 24
2	Areas of Interest.	p. 24
3	CrocoCosmos.	p. 25
4	CodeCity.	p. 26
5	Fase 1 do PerfMiner.	p. 32
6	Fase 2 do PerfMiner.	p. 33
7	Fase 3 do PerfMiner.	p. 34
8	Abordagem completa do PerfMiner.	p. 35
9	Fase 4 do PerfMiner.	p. 36
10	Página inicial da aplicação.	p. 38
11	Página da funcionalidade de Nova Análise.	p. 39
12	Funcionamento do processamento de uma nova análise.	p. 39
13	Passos 2 e 3 da realização de uma nova análise.	p. 41
14	Diagrama de classes parcial do <i>PerfMiner</i>	p. 42
15	Diagrama de classes da extensão proposta.	p. 43
16	Visão geral da Sumarização de Cenários.	p. 45
17	<i>Tooltip</i> com maiores informações sobre determinado cenário.	p. 46
18	Funcionamento geral das visualizações.	p. 47
19	Detalhamento do passo 4 da Figura 18, na Sumarização de Cenários.	p. 48
20	Exemplo do grafo de chamadas.	p. 49
21	Seção de sumário do grafo de chamadas.	p. 50
22	Seção de histórico do grafo de chamadas.	p. 52

23	Nó que representa um método sem desvio de desempenho.	p. 52
24	Nó que representa um método com degradação de desempenho.	p. 53
25	Nó que representa um método com otimização de desempenho.	p. 54
26	Nó que representa um método adicionado.	p. 54
27	Nó que representa um método removido.	p. 55
28	Nó que representa um agrupamento de outros nós.	p. 55
29	Borda espessa de um nó, representando múltiplas execuções.	p. 56
30	Exemplo de nós agrupados durante a hierarquia de chamadas.	p. 57
31	Exemplo de nós agrupados em hierarquias adjacentes (A) e em nós filhos (B).	p. 58
32	Barra de ferramentas do grafo de chamadas.	p. 59
33	Efeito de destaque do caminho de execução de nós com desvio.	p. 59
34	Legenda do grafo de chamadas.	p. 60
35	Detalhes de um nó no grafo de chamadas.	p. 60
36	Detalhamento do passo 4 da Figura 18, no Grafo de Chamadas.	p. 62
37	Dados demográficos dos participantes.	p. 71
38	Exemplos da visualização de Sumarização de Cenários.	p. 74
39	Exemplos da visualização de Sumarização de Cenários com excesso de cenários.	p. 75
40	Gráficos <i>boxplot</i> sobre o algoritmo de redução de nós.	p. 79
41	Grafo de Chamadas do cenário C3.	p. 81
42	Grafo de Chamadas do cenário C4.	p. 82
43	Grafo de Chamadas do cenário C5.	p. 82
44	Grafo de Chamadas do cenário C7.	p. 83
45	Grafo de Chamadas do cenário C8.	p. 84
46	Opinião dos participantes sobre os aspectos da visualização do grafo de chamadas.	p. 87

47	Questões sobre os benefícios e utilidade da ferramenta.	p. 90
48	Exemplo do <i>Performance Evolution Blueprint</i>	p. 98
49	Exemplo da visualização comportamental proposta por Bergel, Robbes e Binder.	p. 102
50	Exemplo do PARCS.	p. 103
51	Exemplo do DFG.	p. 104

Lista de tabelas

1	Resumo das características visuais dos nós.	p. 58
2	Organização dos pares de releases para o Jetty e VRaptor.	p. 68
3	Distribuição dos grupos de participantes.	p. 69
4	Resumo para o Jetty.	p. 72
5	Resumo para o VRaptor.	p. 73
6	Características dos cenários exemplos de casos especiais.	p. 81

Lista de abreviaturas e siglas

SCM – *Source Code Management*

BTS – *Bug Tracking System*

JDK – *Java Development Kit*

JVM – *Java Virtual Machine*

APM – *Application Performance Management*

CASE – *Collaborative & Automated Software Engineering Research Group*

DIMAp – Departamento de Informática e Matemática Aplicada

UFRN – Universidade Federal do Rio Grande do Norte

CPU – *Central Processing Unit*

DAM – *Dynamic Analysis Model*

QAV – *Quality Attribute Visualization*

JSON – *JavaScript Object Notation*

CDI – *Contexts and Dependency Injection*

CCT – *Context Call Tree*

SINFO – Superintendência de Informática

Sumário

1	Introdução	p. 13
1.1	Apresentação do Problema	p. 14
1.2	Limitações das Abordagens Atuais	p. 15
1.3	Abordagem Proposta	p. 17
1.4	Objetivos Gerais e Específicos	p. 18
1.5	Organização do trabalho	p. 19
2	Fundamentação Teórica	p. 20
2.1	Arquitetura de Software	p. 20
2.1.1	Atributos de Qualidade	p. 21
2.1.2	Avaliação Baseada em Cenários	p. 21
2.2	Visualização de Software	p. 22
2.2.1	Visualização de Arquitetura de Software	p. 23
2.2.2	Visualização da Evolução da Arquitetura de Software	p. 25
2.2.3	Estudos Empíricos em Visualização de Software	p. 26
2.3	Ferramentas de Análise de Desempenho	p. 28
2.4	Considerações	p. 30
3	PerfMiner Visualizer	p. 31
3.1	PerfMiner	p. 31
3.1.1	Fase 1: Análise Dinâmica	p. 31
3.1.2	Fase 2: Análise de Desvio	p. 33

3.1.3	Fase 3: Mineração de Repositório	p. 34
3.2	Visão Geral do PerfMiner Visualizer	p. 36
3.2.1	Nova Análise	p. 37
3.2.1.1	Funcionamento	p. 39
3.2.1.2	Bancos de Dados	p. 41
3.3	Visualização da Sumarização de Cenários	p. 44
3.3.1	Interação	p. 46
3.3.2	Funcionamento	p. 46
3.4	Visualização do Grafo de Chamadas	p. 48
3.4.1	Sumário	p. 49
3.4.2	Histórico	p. 51
3.4.3	Grafo de Chamadas	p. 52
3.4.3.1	Nó sem desvio	p. 52
3.4.3.2	Nó degradado	p. 53
3.4.3.3	Nó otimizado	p. 54
3.4.3.4	Nó adicionado	p. 54
3.4.3.5	Nó removido	p. 55
3.4.3.6	Nó de agrupamento	p. 55
3.4.4	Interação	p. 58
3.4.5	Funcionamento	p. 61
3.5	Considerações	p. 62
4	Avaliação	p. 64
4.1	Projeto do Estudo Empírico	p. 64
4.1.1	Objetivos e Questões de Pesquisa	p. 64
4.1.2	Sistemas	p. 66
4.1.3	Procedimentos	p. 67

4.1.3.1	Passo 1 - Coleta e Preparação dos Dados	p. 67
4.1.3.2	Passo 2 - Aplicação da Abordagem	p. 67
4.1.3.3	Passo 3 - Elaboração e Aplicação dos Questionários . .	p. 68
4.1.4	Caracterização dos Participantes	p. 70
4.2	Resultados	p. 71
4.2.1	Análise do Comportamento das Visualizações	p. 72
4.2.1.1	Sumarização de Cenários	p. 72
4.2.1.2	Grafo de Chamadas	p. 77
4.2.2	Questionário Online	p. 83
4.2.2.1	Grafo de Chamadas	p. 84
4.2.2.2	Sumarização de Cenários	p. 88
4.2.2.3	Questões Finais	p. 89
4.3	Considerações	p. 91
4.3.1	Ameaças à Validade	p. 92
4.3.1.1	Validades de Construção	p. 92
4.3.1.2	Validades Internas	p. 93
4.3.1.3	Validades Externas	p. 94
4.3.1.4	Validades de Conclusão	p. 95
5	Trabalhos Relacionados	p. 96
5.1	Ferramentas de Profiling	p. 96
5.2	Ferramentas APM	p. 97
5.3	Abordagens de Degradação de Desempenho	p. 98
6	Conclusão	p. 106
6.1	Revisão das Questões de Pesquisa	p. 107
6.2	Limitações	p. 108

6.2.1	Escalabilidade	p. 108
6.2.2	Automatização	p. 109
6.2.3	Dependência de Linguagem de Programação ou Plataforma . . .	p. 109
6.3	Trabalhos Futuros	p. 110
6.3.1	Atributos de Qualidade Adicionais	p. 110
6.3.2	Aplicação em um Ambiente Real de Desenvolvimento	p. 110
6.3.3	Novas Avaliações	p. 110
6.3.4	Novas Estratégias para Exercitar os Cenários	p. 111
6.3.5	Automatização	p. 111
6.3.6	Customização	p. 112
6.3.7	Disponibilização para a Comunidade	p. 112
Referências		p. 113
Apêndice A – Questionário		p. 119
A.1	Tipo 1	p. 119
A.2	Tipo 2	p. 122

1 Introdução

A crescente utilização dos softwares por usuários com diferentes características e executando em dispositivos distintos, faz com que seja alta a demanda por novas funcionalidades e tecnologias ou para reparação de erros. Caserta e Zendra [1] mencionam que um software se torna rapidamente complexo quando o seu tamanho aumenta, trazendo dificuldades para o seu entendimento, manutenção e evolução. Nesse sentido, a manutenção e evolução de sistemas de software tem se tornado uma tarefa difícil e crítica com o passar dos anos.

O processo de manutenção é conhecido como o mais caro e o que mais consome tempo dentro do ciclo de vida de um software [2]. A maior parte do tempo gasto nesse processo é dedicada a compreender o sistema, especialmente se os desenvolvedores não estiverem envolvidos desde o início do desenvolvimento. Os desenvolvedores podem gastar mais de 60% do esforço de manutenção entendendo o software [3].

Nesse contexto, a evolução de um software tem sido um dos mais importantes tópicos da engenharia de software. Geralmente, lida com grandes quantidades de dados originados de locais diferentes como repositórios de gerenciamento de código-fonte (do inglês *Source Code Management* – SCM) , sistemas de rastreamento de erros (do inglês *Bug Tracking System* – BTS) e listas de e-mail. Um dos principais aspectos da evolução do software é construir teorias e modelos que permitam compreender o passado e o presente, dessa forma, apoiando as tarefas do processo de manutenção de software [4].

A área de Visualização de Software provê representações visuais com o intuito de tornar o software mais compreensível. Essas representações se fazem necessárias para os analistas, arquitetos e desenvolvedores examinarem os softwares devido a sua natureza complexa, abstrata e difícil de ser observada [5]. Essa área pode focar em vários aspectos de sistemas de software, como padrões de projeto, arquitetura, processo de desenvolvimento, histórico de código-fonte, esquemas de banco de dados, interações de rede, processamento paralelo, execução de processos, dentre outros [6]. Entretanto, um dos componentes es-

senciais é a visualização da arquitetura de um software e seus atributos de qualidade.

1.1 Apresentação do Problema

Ghanam e Carpendale [6] afirmam que não apenas os arquitetos estão interessados em visualizar arquiteturas de software, mas também os desenvolvedores, testadores, gerentes de projetos e até mesmo os clientes. Um dos principais desafios dessas visualizações é descobrir representações visuais eficientes e eficazes para exibir a arquitetura de um software juntamente com as métricas de código envolvidas. Essa visualização se torna especialmente difícil quando é adicionada a variável tempo, passando a ser necessária a representação de sua evolução. Desse modo, a quantidade de dados envolvidos aumenta uma vez que todas as versões do software passam a ser consideradas [1][7].

Dentre os atributos de qualidade de uma arquitetura de software, o desempenho pode ser considerado um dos mais importantes para sistemas de software. Falhas de desempenho podem resultar em relações com o cliente prejudicadas, perda de produtividade para os usuários, perda de receitas, etc [8]. A maioria das falhas de desempenho se dá devido à falta de acompanhamento de problemas dessa natureza no início do processo de desenvolvimento [8].

A evolução de software se refere às mudanças dinâmicas de características e comportamento de um software ao longo do tempo [9]. Nesse processo, as modificações progressivas podem ter consequências negativas, diminuindo a qualidade do software e aumentando a sua complexidade [10][11]. Essa deterioração pode também afetar o desempenho dos sistemas ao longo do tempo [12]. Sandoval Alcocer et al.[13] reforçam que mudanças no código-fonte podem causar comportamentos inesperados em tempo de execução, como, por exemplo, o desempenho de partes da aplicação podem ser degradados em uma nova versão em comparação com a versão anterior. Nesse contexto, sem o devido acompanhamento, os atributos de qualidade, como o desempenho, inicialmente definidos a partir de decisões arquiteturais e de *design* tomadas durante o processo de desenvolvimento podem deixar de ser bem atendidos.

Diante disso, é importante a definição de uma abordagem visual para facilitar a análise da evolução do atributo de qualidade de desempenho para os desenvolvedores e arquitetos dos sistemas, de modo a ajudá-los a gerenciar essa evolução e, conseqüentemente, diminuir a probabilidade de ocorrer variações não planejadas desse atributo de qualidade durante o processo de manutenção ou desenvolvimento de uma nova funcionalidade.

1.2 Limitações das Abordagens Atuais

De acordo com a literatura, existem várias ferramentas que tratam da visualização da evolução arquitetural de softwares, inclusive relacionadas ao monitoramento de desempenho, cada uma com suas particularidades. Caserta e Zendra [1] apontam que essas ferramentas apresentam a evolução arquitetural a partir de: (i) como a arquitetura global é alterada a cada versão, incluindo mudanças no código fonte; (ii) como os relacionamentos entre os componentes evoluem; e (iii) como as métricas evoluem a cada *release*. Este trabalho considera tais aspectos focalizando especificamente o atributo de qualidade de desempenho, em termos de tempo de execução.

As ferramentas de *profiling* realizam análise desse atributo de qualidade no software, porém com características diferentes. A *VisualVM* [14], distribuída gratuitamente com o *Java Development Kit* (JDK), exibe o tempo de execução de cada método em tempo real e o usuário pode, à medida que deseja, tirar fotografias instantâneas da execução do software, os chamados *snapshots*. Essa ferramenta, no entanto, não oferece a comparação do tempo de execução do mesmo método em versões anteriores do software, tornando difícil a visualização da evolução do atributo de qualidade de desempenho, uma vez que teria que ser feita manualmente para cada método desejado.

Já o *JProfiler* [15], ferramenta paga, pode exibir o grafo de chamadas dos métodos em tempo real, com seus respectivos tempos de execução. Assim como o *VisualVM*, a ferramenta oferece a possibilidade de guardar snapshots de determinados momentos da execução. Contudo, os *snapshots* não são automáticos, o usuário precisa, deliberadamente, informar à ferramenta quando ele deve ser acionado. Uma maneira de contornar esse problema é fazendo o uso de *triggers*, onde o usuário pode configurar a ferramenta para responder a determinados eventos da *Java Virtual Machine* (JVM) e, assim, executar algumas ações. Apesar da funcionalidade ser interessante e poderosa, dependendo do que o usuário deseja, a configuração das *triggers* pode se tornar maçante. A ferramenta oferece a comparação entre os *snapshots*, porém, não é automática e necessita da ação do usuário para escolher quais *snapshots* serão comparados. A ferramenta *YourKit Java Profiler* [16] possui funcionalidades semelhantes às comentadas para o *JProfiler*. Entretanto, é uma ferramenta paga e a comparação entre os *snapshots* também não é automática.

Com exceção do *VisualVM*, as ferramentas de *profiling* mencionadas possuem uma característica em comum: a forma com que apresentam a evolução do atributo de qualidade de desempenho não é automática e tampouco é direcionada ao(s) método(s) desejado(s)

pelo usuário. É necessário selecionar manualmente os *snapshots* que se deseja comparar e as ferramentas exibem duas formas de visualização da evolução do desempenho, em geral: *call tree* e *hot spot*. Em ambas, são apresentados todos os métodos monitorados, cabendo ao usuário procurar o método desejado para, então, verificar qual a sua evolução.

Corroborando com o exposto para o *JProfiler* e o *YourKit Java Profiler*, [Sandoval Alcocer et al.](#)[13] mencionam que essas duas ferramentas, apesar de serem úteis para acompanhar o desempenho geral, comparar a diferença dos tempos dos métodos é muitas vezes insuficiente para compreender as razões para a variação de desempenho. Os autores listam algumas limitações dessas duas ferramentas, tais como: as variações de desempenho têm que ser manualmente rastreadas e as visualizações utilizadas são ineficientes. Essas ferramentas também não oferecem maiores detalhes sobre os desvios, tais como: código-fonte dos métodos, possíveis *commits* que introduziram o desvio e quais tarefas estariam relacionadas com a degradação ou melhoria encontrada. Novamente, caso o usuário queira identificar tais características para determinado método com desvio terá que pesquisar manualmente diretamente na fonte dos dados: repositório de código-fonte e sistema de gerenciamento de tarefas.

[Ahmed et al.](#)[17] realizaram um estudo para verificar se as ferramentas de gerenciamento de desempenho de aplicações (APM, do inglês *Application Performance Management*) são eficazes na identificação de regressões de desempenho. Os autores definem regressão de desempenho quando as atualizações em um software provocam uma degradação no seu desempenho [17]. As ferramentas utilizadas no estudo foram *New Relic* [18], *AppDynamics* [19], *Dynatrace* [20] e *Pinpoint* [21]. Como resultado, eles mostram que a maioria das regressões inseridas no código-fonte foram detectadas pelas ferramentas. Contudo, o processo de identificação da causa da regressão, ou seja, o método exato cujo código foi inserido, foi mais complicado, sendo necessário bastante trabalho manual: os autores inspecionavam as transações (requisições) marcadas como lentas e, manualmente, comparavam os respectivos *stacktraces* para verificar se a ferramenta indicava corretamente a regressão de desempenho. O processo, mais uma vez, não é automático e não existem visualizações adequadas que esclareçam a regressão de desempenho.

As abordagens existentes trazem melhorias no tocante à necessidade de novas visualizações da evolução do software com foco no atributo de qualidade de desempenho, bem como a automação completa (ou parcial) do processo de análise, uma vez que ainda é necessária considerável intervenção manual do usuário nas ferramentas existentes para visualizar a evolução de diferentes versões do sistema. Além da necessidade de novas vi-

sualizações, outros requisitos de uma abordagem para análise de desvios de desempenho na evolução de sistemas são mencionados por Pinto [22]:

- Deveria automatizar o processo ao máximo. Técnicas manuais consomem tempo e são custosas, além de não se mostrarem adequadas se o processo de avaliação requerer a análise de sucessivas evoluções;
- Os softwares podem ser muitos grandes para uma análise completa. Dessa forma, a ferramenta deveria focar em partes selecionadas do sistema;
- A ferramenta deveria ser capaz de medir o desempenho de determinados cenários e seus métodos a fim de identificar onde ocorreu o desvio;
- Deveria prover suporte para análise do código-fonte com o intuito de oferecer feedback detalhado sobre o código relacionado com o desvio detectado;
- Deveria ser capaz de acessar dados dos repositórios do software, como ferramentas de controle de versão e sistemas de gerenciamento de tarefas, com a finalidade de exibir as mudanças relacionadas ao código com desvio de desempenho.

1.3 Abordagem Proposta

Este trabalho apresenta uma ferramenta, chamada *PerfMiner Visualizer*, cujo objetivo é aplicar técnicas de visualização de software para ajudar desenvolvedores e arquitetos a analisar a evolução do atributo de qualidade de desempenho, em termos de tempo de execução, ao longo das versões de um software. A ferramenta propõe duas visualizações com escopos e granularidades diferentes e proporciona ao usuário mecanismos de interação para explorá-las.

O tempo de execução, neste trabalho, é o tempo que um dado método ou cenário¹ do sistema demora para executar. Pode também ser tratado como tempo de resposta. Para medir o desempenho, além do tempo de execução, outras propriedades podem ser usadas, como: consumo de memória, entrada e saída de disco, uso do processador e tráfego de rede [23]. A medição desse atributo em termos de tempo de execução foi escolhida por se tratar de uma propriedade geral e comum para a capacidade de resposta de um sistema.

¹Um cenário é definido como uma ação de alto nível que representa a maneira como os *stakeholders* interagem com o sistema.

A ferramenta foi implementada como extensão a outra já existente, chamada *PerfMiner* [22]. Essa ferramenta busca apontar quais cenários degradaram ou otimizaram o atributo de qualidade de desempenho. A escolha desse atributo de qualidade se deu pelo fato de ser uma propriedade crítica para a maioria dos sistemas de software atuais.

A extensão proposta visa oferecer um melhor entendimento da evolução desse atributo de qualidade na arquitetura de um software por parte dos arquitetos e desenvolvedores, beneficiando-os ao: (i) fornecer uma visão geral dos cenários com desvios de desempenho entre uma determinada versão do software e a anterior; (ii) possibilitar a identificação dos cenários que possuem elevado tempo de execução; (iii) saber qual desses cenários teve o maior desvio de desempenho; (iv) acompanhar a evolução de cada cenário ao longo das versões analisadas; (v) saber, para cada cenário, os métodos que foram detectados com algum tipo de desvio, além de ter conhecimento sobre os métodos que foram adicionados e removidos; e (vi) fornecer uma listagem de *commits* que possivelmente foram as causas dos desvios de desempenho identificados. Com base nisso, a equipe de desenvolvimento pode tomar ações para sanar possíveis problemas no desempenho das aplicações além de acompanhar a evolução, planejada ou não, desse atributo de qualidade.

1.4 Objetivos Gerais e Específicos

O objetivo principal deste trabalho é implementar uma ferramenta com o intuito de prover um conjunto de visualizações de modo a facilitar o entendimento da evolução do atributo de qualidade de desempenho. A implementação dessa ferramenta foi feita estendendo outra ferramenta já existente desenvolvida pelo grupo de pesquisa *Collaborative & Automated Software Engineering Research Group* (CASE), do Departamento de Informática e Matemática Aplicada (DIMAp) da UFRN.

A ferramenta estendida, chamada de *PerfMiner* [22], pode ser definida como uma abordagem automatizada baseada em cenários para identificar desvios de desempenho, em termos de tempo de execução. A ferramenta indica, também, quais trechos de código-fonte podem ter causado a variação de desempenho baseado na mineração de *commits* e tarefas (*issues*) de desenvolvimento. Técnicas de análise dinâmica e mineração de repositórios de software são usadas pela ferramenta para atingir os seus objetivos. Nesse contexto, os objetivos específicos deste trabalho são:

- Investigar as principais abordagens de visualização da evolução do atributo de qualidade de desempenho, a fim de conhecer quais técnicas são utilizadas atualmente, e

identificar lacunas que servem como motivação para o desenvolvimento do trabalho;

- Projetar e implementar a ferramenta *PerfMiner Visualizer*. A ferramenta é desenvolvida na linguagem de programação Groovy, de modo que a análise de desempenho é suportada para sistemas desenvolvidos na linguagem Java, além do Groovy;
- Conduzir estudos de avaliação da ferramenta para avaliar a utilidade e eficácia das visualizações para encontrar informações sobre os desvios de desempenho e a aplicabilidade da ferramenta como parte integrante dos processos de desenvolvimento desses sistemas.

1.5 Organização do trabalho

Este trabalho está organizado como segue: o capítulo 2 apresenta a fundamentação teórica para o entendimento do trabalho, tais como arquitetura de software, visualização de software e ferramentas de análise de desempenho. O capítulo 3 apresenta a solução proposta, mostrando as visualizações definidas e implementadas para melhorar o entendimento das análises, além de explicar o funcionamento do *PerfMiner* integrado com o conjunto de visualizações proposto. O capítulo 4 discute o estudo aplicado para a avaliação da ferramenta e suas visualizações. O capítulo 5 exhibe trabalhos relacionados, mencionando as suas limitações e comparando-os com este trabalho. Finalmente, o capítulo 6 apresenta as conclusões do trabalho, discute as limitações e propõe trabalhos futuros.

2 Fundamentação Teórica

Este capítulo apresenta conceitos importantes para a compreensão deste trabalho. A seção 2.1 explora os conceitos de arquitetura de software e atributos de qualidade, incluindo a definição de cenários. A seção 2.2 discorre sobre os conceitos de visualização de software, além de definições sobre a visualização da evolução de arquitetura de software. Na seção 2.3, são explanados os conceitos empregados nas ferramentas de análise de desempenho. Por fim, na seção 2.4 são apresentadas as considerações finais do capítulo.

2.1 Arquitetura de Software

O estudo da arquitetura de software é o estudo de como sistemas de software são projetados e construídos. Ela deve ser o coração do projeto e desenvolvimento de um sistema, estando acima dos processos, análises e do desenvolvimento [24].

A arquitetura de software pode ser definida como a estrutura ou estruturas do sistema, que compreendem os elementos de software, as propriedades externamente visíveis desses elementos e as relações entre eles [25]. Adicionalmente, de acordo com [Taylor, Medvidovic e Dashofy](#)[24], a arquitetura de um software incorpora todas as decisões de projeto tomadas pelos seus arquitetos, que podem afetar muitos dos seus módulos, incluindo sua estrutura e atributos de qualidade.

Antes da implementação do sistema, decisões arquiteturais tais como quais componentes pertencem a arquitetura do software, quais serviços ou propriedades desses componentes serão externamente visíveis e como eles estão relacionados uns com os outros, devem ser tomadas e deveriam permanecer atendidas durante todo o ciclo de vida do sistema. É igualmente importante o fato de perceber que a arquitetura do software é essencial para o sistema alcançar os requisitos relacionados aos atributos de qualidade [26].

2.1.1 Atributos de Qualidade

A medida que o domínio de um software evolui, assim como os seus requisitos, a arquitetura do software precisa ser reavaliada de modo que ainda reflita um sistema moderno que se encaixa no domínio evoluído. Uma arquitetura não é regida apenas por requisitos funcionais, mas em grande parte por atributos de qualidade. Desse modo, criar uma arquitetura apropriada não é uma tarefa trivial [27].

A qualidade não pode ser adicionada ao sistema de maneira tardia, pelo contrário, deve ser incorporada desde o início [27]. Portanto, a arquitetura de um software necessita ser reavaliada para verificar a conformidade dos requisitos funcionais e atributos de qualidade.

Os atributos de qualidade que serão analisados em um processo de avaliação dependem do contexto e domínio do sistema. Uma estratégia comum é abordar os atributos de qualidade mais críticos. Alguns desses atributos são definidos adiante [26]:

- *Desempenho*: trata da capacidade de resposta do sistema, o tempo requerido para responder a eventos ou o número de eventos processados em determinado intervalo de tempo;
- *Confiabilidade*: é a habilidade do sistema se manter operante com o passar do tempo. É geralmente medido pelo tempo médio até a falha;
- *Segurança*: mede a habilidade do sistema de resistir a tentativas de uso não autorizado e negação de serviço, enquanto continua fornecendo seus serviços a usuários autorizados;
- *Portabilidade*: é a capacidade do sistema de executar em diversos ambientes computacionais.

Embora existam outros atributos de qualidade, o atributo de interesse deste trabalho é o desempenho, em termos de tempo de execução. O desempenho será destacado como uma das principais informações mostradas nas visualizações descritas posteriormente.

2.1.2 Avaliação Baseada em Cenários

Algumas abordagens de avaliação arquitetural foram propostas para lidar com questões relacionadas com a qualidade em arquiteturas de software, dentre elas, a avaliação baseada em cenários é considerada bastante madura [28]. O propósito dessa avaliação é

exercitar os cenários com a finalidade de determinar se a arquitetura é adequada para um conjunto de atributos de qualidade.

Um cenário é definido pela interação entre os *stakeholders* e o sistema. Eles são particularmente úteis para ajudar os arquitetos a entender como os atributos de qualidade podem ser abordados. Os *stakeholders* podem ter diferentes perspectivas dos cenários. Por exemplo, um usuário pode imaginar um cenário como uma tarefa que ele precisa fazer no sistema. Por outro lado, um desenvolvedor, que irá implementar o cenário, pode focar em uma visão arquitetural e usar a arquitetura do software para guiar o processo de desenvolvimento [22].

Nesse contexto, os cenários se tornam especialmente úteis quando os programadores e arquitetos precisam obter uma melhor compreensão sobre os atributos de qualidade, uma vez que especificam todos os tipos de operações que o sistema terá que executar para atender a determinadas funcionalidades. Assim, uma análise detalhada do modo como essas operações serão implementadas, executadas ou até mesmo se elas podem falhar ou não, ajuda os avaliadores a extrair informações importantes sobre os atributos de qualidade, por exemplo, o desempenho.

2.2 Visualização de Software

A área visualização de software é parte da visualização da informação e pode ser definida, de maneira ampla, como a visualização de artefatos relacionados ao software e seu processo de desenvolvimento. Adicionalmente ao código-fonte do programa, esses artefatos incluem documentação de requisitos e projeto, mudanças no código-fonte e relatório de *bugs*, por exemplo [29]. Ainda de acordo com Diehl[29], a visualização de software é a arte e ciência de gerar representações visuais de vários aspectos de um software e de seu processo de desenvolvimento. O objetivo principal é ajudar a compreender sistemas de software e aumentar a produtividade do processo de desenvolvimento.

Essas representações são necessárias para que os analistas, arquitetos e desenvolvedores examinem os sistemas de software devido à sua natureza complexa, abstrata e difícil de observar [5]. Tais dificuldades são ainda piores em sistemas de grande escala.

Existem três tipos de aspectos do software que a visualização pode abordar [29]:

- (i) *Estático*: neste tipo, a visualização do software apresenta o software como ele é codificado, lidando com informações que são válidas para todas as suas possíveis

execuções. A estrutura do software em vários níveis de abstração pode ser obtida através desse aspecto, incluindo sua arquitetura;

- (ii) *Dinâmico*: provê informações sobre uma execução particular do sistema e ajuda a entender o seu comportamento. Pode exibir quais instruções são executadas e como o estado do programa se altera. Exemplos: visualização dinâmica da arquitetura, algoritmos animados e depuração e testes visuais;
- (iii) *Evolução*: adiciona a variável tempo a visualização dos aspectos estáticos ou dinâmicos do software.

Com relação ao que os usuários visualizam, [Gomez-Henriquez\[30\]](#) menciona que a visualização de software é principalmente usada para: (i) exibição do comportamento do programa, normalmente para fins pedagógicos; (ii) depuração lógica; e (iii) depuração de desempenho. Entretanto, a lista pode ser estendida para: atividades de desenvolvimento, depuração, testes, manutenção e detecção de falhas, reengenharia, engenharia reversa, gerenciamento de processo de desenvolvimento e marketing [\[31\]](#). [Diehl\[29\]](#) sumariza a lista em apenas seis itens: projeto, implementação, testes, depuração, análise e manutenção.

2.2.1 Visualização de Arquitetura de Software

Um dos mais importantes tópicos na área de visualização de software é a visualização da arquitetura do software [\[6\]\[32\]\[33\]\[34\]](#). Geralmente, os sistemas orientados a objetos são estruturados de maneira hierárquica, com pacotes contendo subpacotes, recursivamente, e com classes estruturadas por métodos e atributos. Visualizar a arquitetura consiste em observar a hierarquia e os relacionamentos entre os componentes do software [\[1\]](#). No entanto, estudos indicam que, além dos módulos do software, com suas estruturas, relacionamentos e métricas, existe um interesse crescente em visualizar a evolução desses módulos [\[35\]](#).

As representações visuais mais comuns para visualizar a estrutura hierárquica da arquitetura de um software é uma estrutura em árvore [\[7\]](#). Exemplos de outras formas de se representar a arquitetura são o *Treemap* retangular [\[36\]](#) e circular [\[37\]](#) e o *Icicle Plot* [\[38\]](#). Já para a visualização dos relacionamentos, podem ser mencionados: *Dependency Structure Matrix* [\[39\]](#), *DA4Java* [\[40\]](#), *EvoSpaces* [\[41\]](#) e *Clustered Graph Layout* [\[42\]](#). Para a visualização das métricas de um software, são exemplos as soluções: *Metric View* [\[43\]](#), *Areas of Interest* [\[44\]](#) e *CodeCrawler* [\[45\]](#).

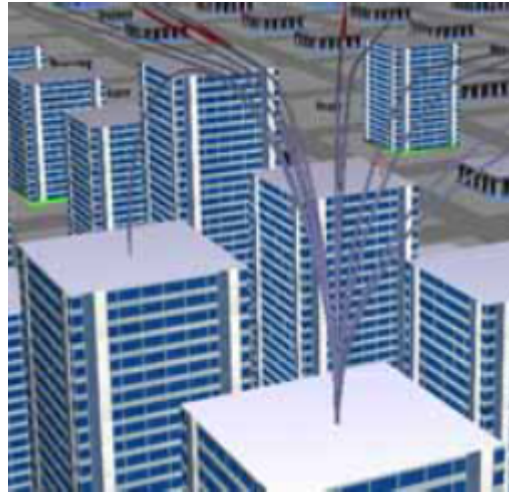


Figura 1: EvoSpaces [41].

O EvoSpaces [41], mostrado na Figura 1, é uma ferramenta que provê a visualização da arquitetura do software em um ambiente virtual. Aproveita o fato de que os sistemas são, muitas vezes, estruturados hierarquicamente para sugerir o uso de uma metáfora de cidades. As entidades, junto com suas relações, são representadas como glifos residenciais (casa, apartamento, escritório, etc), ao passo que as métricas dessas entidades são exibidas como posições e escalas visuais (tamanho, valor da cor, etc). A ferramenta possui diferentes modos de interação, como *zoom* e capacidades de navegação.

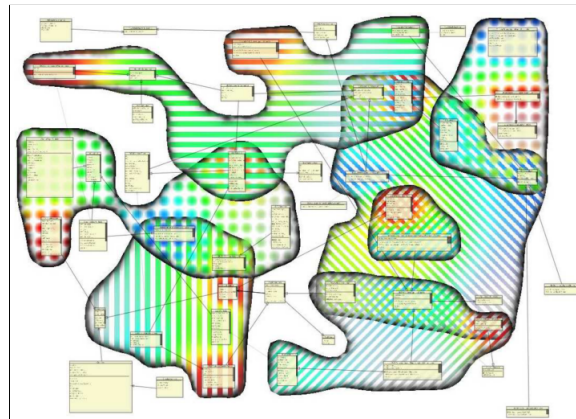


Figura 2: Areas of Interest [44].

O Areas of Interest [44], mostrado na Figura 2, é uma técnica que consiste em aplicar um algoritmo que agrupa entidades de software com propriedades comuns, as engloba com um contorno e adiciona cores para descrever métricas de software. Para distinguir áreas de sobreposição, cada área tem uma textura diferente, como linhas horizontais, verticais, diagonais e círculos. Além disso, técnicas de sombreamento e transparência são usadas para melhorar a distinção entre várias áreas.

2.2.2 Visualização da Evolução da Arquitetura de Software

A evolução de um software tem sido destacada como um dos tópicos mais importantes na engenharia de software [4]. Trata-se de uma tarefa complexa por gerar grande quantidade de dados e lidar com isso é complicada: estima-se que 60% do esforço na etapa de manutenção é para entender o software [3]. A visualização de software visa ajudar os *stakeholders* a melhorar a compreensão do software, no entanto, elaborar metáforas visuais que representem efetivamente a dimensão tempo com todas as informações relacionadas a evolução do software é uma tarefa difícil [46].

No contexto da visualização da evolução da arquitetura de um software, um dos grandes desafios, além da quantidade de dados provenientes da evolução, é o crescente tamanho e complexidade do software. Apesar da dificuldade, é importante prover visualizações gerais da arquitetura, dos relacionamentos entre os módulos e das métricas, para cada versão [7].

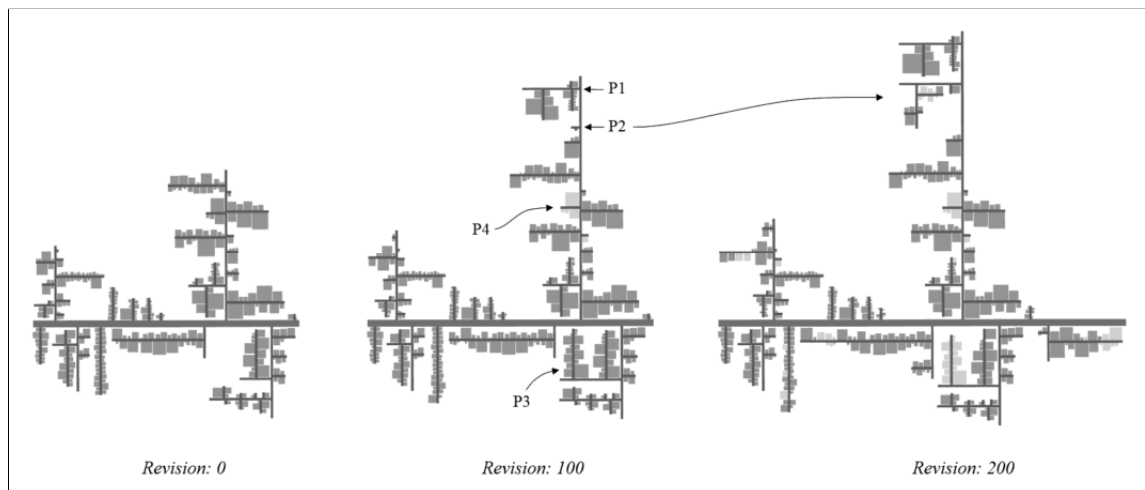


Figura 3: CrocoCosmos [47].

O *CrocoCosmos* [47] é um exemplo de ferramenta que utiliza uma metáfora de cidades para representar a evolução estrutural da arquitetura de um software, onde ruas representam pacotes e construções refletem as classes Java. A sequência de representações visuais objetiva destacar as mudanças básicas na estrutura do software, como elementos que foram adicionados, removidos ou movidos dentro da hierarquia. A Figura 3 mostra essa ferramenta. O *Code Flows* [48] e o *Successive Inheritance Graphs* [49] são outros exemplos de visualizações de evolução estrutural de arquitetura de software.

Outra forma de se exibir a evolução de um software é através das suas métricas. Elas encapsulam, resumizam e proveem informações de qualidade sobre o código-fonte [50]. As

métricas são essenciais para o entendimento contínuo e para a análise da qualidade do sistema durante todas as fases do seu ciclo de vida [7].

O *CodeCity* [51] é uma visualização interativa em 3D que avalia a evolução estrutural de sistemas de software e as apresenta utilizando uma metáfora de cidades. As métricas são exibidas através de propriedades visuais dos artefatos da cidade: as propriedades das classes, como o número de métodos e de atributos, são mapeadas como a altura e o tamanho da base das construções, respectivamente; a profundidade dos pacotes é representada através da saturação de cores dos distritos. A Figura 4 exemplifica essa ferramenta.

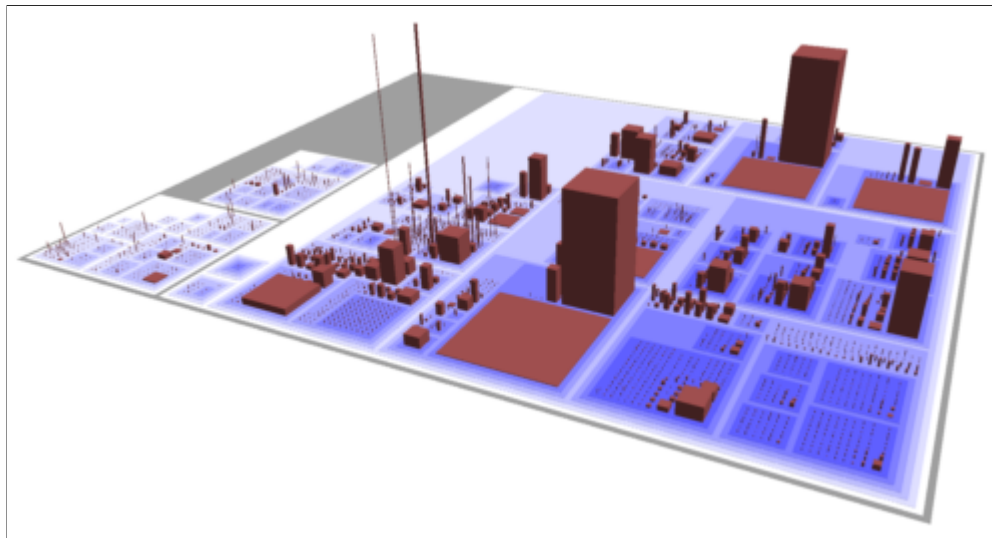


Figura 4: Codecity [51].

Outras soluções que apresentam visualizações da evolução da arquitetura de um software através das métricas são: *The Evolution Matrix* [52], *VERSO* [50] e *RelVis* [53].

2.2.3 Estudos Empíricos em Visualização de Software

As visualizações são importantes para ajudar na compreensão de vários aspectos de um software, além de ajudar a aumentar a produtividade do processo de desenvolvimento, lançando mão de metáforas visuais para representar esses aspectos. Entretanto, não significa que toda visualização de software é útil. Para cada técnica em particular, e para cada intenção de uso, é necessário avaliar a sua utilidade [29]. O objetivo principal de uma visualização é transmitir informações de forma compreensível, eficaz e fácil de lembrar. Diehl[29] agrupa as avaliações das visualizações em dois grupos: quantitativa e qualitativa.

Os métodos quantitativos de avaliação medem propriedades da visualização ou pro-

priedades do usuário interagindo com a visualização. Esse tipo de avaliação requer uma análise estatística dos resultados de um experimento controlado [29].

Já o método qualitativo coleta dados sobre a experiência dos usuários com as visualizações, verbalizados em formato de relatórios. Os métodos qualitativos são de extrema importância quando se trata da percepção humana e da interação com uma visualização [54], incluindo o fato de eles exigirem menos pessoas para o teste e cobrirem mais aspectos da visualização avaliada.

Komlodi, Sears e Stanziola[55], em sua pesquisa com cerca de 50 estudos de usuários de sistemas de visualização de informação, encontrou quatro áreas de avaliação:

- (i) *Experimentos controlados comparando elementos de design*: esses estudos podem comparar elementos específicos;
- (ii) *Avaliação de usabilidade de uma ferramenta*: esses estudos podem fornecer *feedback* sobre problemas encontrados pelos usuários com uma ferramenta e mostrar como os *designers* podem refinar o *design*;
- (iii) *Experimentos controlados comparando duas ou mais ferramentas*: geralmente tentam comparar novas abordagens com o estado da arte;
- (iv) *Estudos de caso de ferramentas em contextos realistas*: a vantagem dos estudos de caso é que eles relatam os usuários em seu ambiente natural fazendo tarefas reais, demonstrando viabilidade e utilidade no contexto. A desvantagem é que eles são demorados para conduzir e os resultados podem não ser replicáveis e generalizáveis.

Serai et al.[56] realizou um estudo de mapeamento sistemático de métodos de validação em visualização de software. Os autores definiram seis propriedades de classificação desses métodos:

- (i) *Tipo de Investigação*: determina qual o tipo de estudo empírico foi usado: experimento, estudo de caso ou questionário;
- (ii) *Tarefas*: especifica a natureza das tarefas envolvidas na avaliação. Elas são específicas quando o participante tem que resolver um problema específico, ou exploratórias quando o participante não tem uma tarefa específica para executar;
- (iii) *Fonte de Dados*: caracteriza a fonte dos dados da visualização: industrial, *open source* e/ou dados domésticos;

- (iv) *Participantes*: determina o perfil dos participantes usados na avaliação, se houver: estudantes, profissionais ou ambos;
- (v) *Medidas*: especifica se a avaliação incluiu medidas objetivas, ou seja, sem ser baseadas no julgamento dos participantes. Caso contrário, as medidas poderiam ser subjetivas ou sem medida
- (vi) *Referência de Comparação*: define se a ferramenta foi comparada a outras ferramentas.

Como resultados, os autores destacam que 78,16% dos artigos analisados utilizaram o método de estudo de caso, sendo destes, 65% puramente análise qualitativa. Além disso, 72,5% envolviam tarefas específicas na avaliação, 77% dos artigos usaram ferramentas *open source*, 70,1% das avaliações foram realizadas sem participantes. Com relação às medidas, 60,9% dos trabalhos coletaram medidas objetivas. Por fim, 77% dos artigos não incluíram nenhuma comparação com outras abordagens. Baseado nesses resultados, os autores concluíram que a análise dos tipos de experimentos feitos mostra uma tendência em relação aos estudos de caso, tarefas específicas, fonte de dados *open source*, sem participantes, com medidas objetivas e sem referência de comparação [56].

2.3 Ferramentas de Análise de Desempenho

O tamanho e a complexidade das aplicações modernas aumentaram a demanda por ferramentas que colem dados sobre o comportamento dinâmico dos programas, e portanto permitam aos desenvolvedores identificarem gargalos de desempenho nas suas aplicações com um mínimo de esforço. O processo de coleta automática e apresentação dos dados de desempenho de sistemas em execução é chamado de *profiling* [57].

Nesse contexto, o *profiling* de CPU determina quanto tempo o programa gastou executando várias partes do código. Com isso, se pode calcular o desempenho, em termos de tempo de execução, de determinada funcionalidade ou rotina. Já o *profiling* de memória determina o número, tipos e ciclos de vida dos objetos que o programa aloca [57]. Existem outras modalidades, no entanto a de CPU e memória são as mais usadas.

Das diferentes técnicas de se realizar o *profiling*, a baseada em instrumentação é a mais comum. Essa técnica trabalha inserindo, ou injetando, trechos especiais de código, chamados de código de instrumentação, na aplicação. A execução desses trechos gera eventos, como entrada/saída de métodos ou alocação de objetos. Esses dados são coletados,

processados e, eventualmente, apresentados ao usuário. Com relação ao *profiling* de CPU, essa técnica grava exatamente o número exato de eventos, ao invés de uma aproximação estatística (como acontece com o *profiling* baseado em amostragem) [57].

Uma das formas de utilizar o *profiling* baseado em instrumentação é lançando mão do paradigma de programação orientado a aspectos, que permite o aumento da modularidade de um sistema através da separação de interesses. O princípio é que alguma lógica ou funcionalidade possa agir transversalmente entre as diferentes lógicas encapsuladas no sistema usando diferentes tipos de abstrações [58]. Para a linguagem de programação Java, o suporte a programação orientada a aspectos é feito usando a biblioteca *AspectJ*.

Existem no mercado várias ferramentas que realizam a medição do atributo de qualidade de desempenho para a linguagem Java. Algumas delas são:

- *VisualVM* [14]: distribuída gratuitamente com o *Java Development Kit* (JDK), exibe o tempo de execução de cada método em tempo real e o usuário pode, à medida que deseja, tirar fotografias instantâneas da execução do software, os chamados *snapshots*;
- *JProfiler* [15]: ferramenta paga, pode exibir o *call graph* de chamadas dos métodos em tempo real, com seus respectivos tempos de execução. Assim como o *VisualVM*, a ferramenta oferece a possibilidade de guardar *snapshots* de determinados momentos da execução;
- *YoutKit Java Profiler* [16]: ferramenta paga que possui funcionalidades semelhantes às do *JProfiler*.

Além das ferramentas de *profiling*, outra maneira de se realizar a análise de desempenho de aplicações é utilizando ferramentas de gerenciamento de desempenho de aplicações, as chamadas APM. Essas ferramentas integram abordagens de mineração de dados de desempenho em ferramentas de monitoramento de desempenho disponíveis no mercado e são frequentemente utilizadas para detectar anomalias no desempenho [17]. Exemplos desse tipo de ferramentas são o *New Relic* [18], *AppDynamics* [19], *Dynatrace* [20] e *Pinpoint* [21].

2.4 Considerações

Os conceitos apresentados neste capítulo são importantes para o entendimento da proposta. Com relação ao conceito de arquitetura de software, a ferramenta *PerfMiner* é focada principalmente na implementação do sistema, e não nos componentes arquiteturais ou relacionamentos entre eles. Além disso, apesar de existirem vários atributos de qualidade, apenas o de desempenho, medido em termos de tempo de execução, é considerado. Com relação aos cenários, é a forma com a qual a avaliação da arquitetura do software é guiada e eles são definidos, no contexto deste trabalho, como sendo um caso de teste automatizado do sistema.

A respeito da visualização de software, a ferramenta proposta utiliza representações visuais para exibir aspectos dinâmicos e a evolução do atributo de qualidade de desempenho ao longo das versões do sistema. Assim, como o *PerfMiner* não trata dos componentes arquiteturais ou dos seus relacionamentos, não são usadas representações visuais para tal fim na ferramenta proposta.

Foram mencionadas neste capítulo ferramentas de análise de desempenho, como as de *profiling* e APM, pois é possível medir o desempenho de aplicações utilizando-as. Entretanto, há diferenças para a ferramenta *PerfMiner Visualizer*, principalmente no tocante a identificação da evolução do desempenho. Com relação a técnica de coleta automática de dados, o *PerfMiner* utiliza a instrumentação de código através do *AspectJ*.

3 PerfMiner Visualizer

Este capítulo apresenta o conjunto de visualizações proposto como extensão à ferramenta *PerfMiner*. A seção 3.1 destaca o funcionamento dessa ferramenta. A seção 3.2 mostra uma visão geral sobre o funcionamento da ferramenta proposta. As seções 3.3 e 3.4 descrevem as visualizações da sumarização de cenários e grafo de chamadas, respectivamente, com suas características, propriedades visuais e funcionamento. Por fim, são reportadas considerações finais sobre o capítulo na seção 3.5.

3.1 PerfMiner

A implementação das visualizações foi desenvolvida como uma extensão da ferramenta *PerfMiner*, a qual provê a avaliação de cenários de execução do sistema no que diz respeito a desvios de desempenho, de forma a minimizar a erosão de tal atributo de qualidade [22].

A principal funcionalidade do *PerfMiner* é realizar a análise de desvio de desempenho entre duas versões de um sistema, revelando de maneira automatizada, as potenciais causas para o desvio nos cenários. Para isso, técnicas de análise dinâmica e mineração de repositório são usadas para estabelecer uma abordagem baseada em cenários para a avaliação do atributo de qualidade de desempenho, medido em termos de tempos de execução [22].

Para atingir o objetivo de realizar a análise de desvio de desempenho, o *PerfMiner* define três fases, descritas nas subseções a seguir: (i) análise dinâmica; (ii) análise de desvio; e (iii) mineração de repositório.

3.1.1 Fase 1: Análise Dinâmica

A fase de análise dinâmica consiste em realizar a execução dos cenários através de uma suíte de testes automatizados. Como resultado, essa fase gera um modelo de análise dinâmica, que é persistido em um banco de dados e contém informações sobre os *traces*

de execução do sistema, modelados por um grafo de chamadas dinâmico que representa cada execução dos cenários selecionados para determinada versão [22]. A Figura 5 ilustra esta fase.

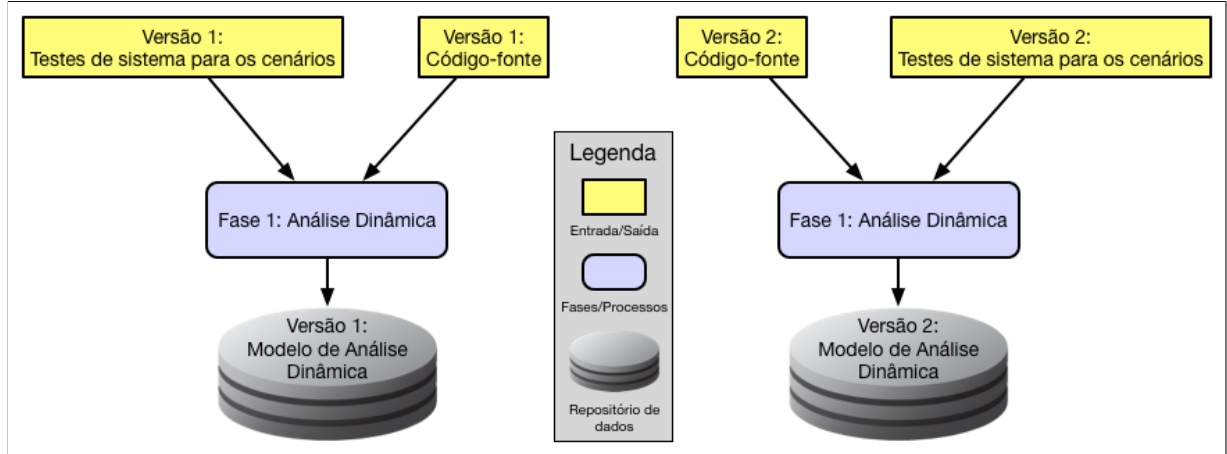


Figura 5: Fase 1 do PerfMiner.

O grafo é montado interceptando os métodos de entrada de cada cenário e instrumentando suas execuções, calculando o tempo de execução de cada nó, bem como informações sobre se o cenário falhou ou não. Esse grafo pode ser interpretado como uma estrutura em árvore onde cada nó é a execução de um método, onde os métodos de entrada representam os nós raiz [22].

Duas informações são importantes nesse processo:

- (i) É fundamental que cada versão do sistema analisado possua testes automatizados para que o sistema seja executado. Caso o sistema não tenha testes automatizados, uma estratégia alternativa é utilizar ferramentas de teste de desempenho, como o *JMeter* [59], para submeter requisições que exercitem os cenários em um sistema web;
- (ii) Cada teste de sistema executado é considerado um cenário. Dessa forma, cada cenário analisado é representado na ferramenta com o seguinte nome: “*Entry point for SimpleClassName.testMethodName*”.

A ferramenta usa *AspectJ* para definir um aspecto que instrumenta as execuções dos cenários, interceptando os métodos de entrada para montar o grafo de chamadas e calcular os tempos de execução dos cenários e dos seus métodos [22]. Vale salientar que todo o processo dessa fase deve ser executado uma vez para cada versão do sistema a ser

analisado. Dessa forma, serão gerados dois modelos de análise dinâmica que são utilizados para calcular os desvios de desempenho dos cenários e métodos na fase seguinte.

A análise dinâmica é executada no mesmo computador para todas as versões, nas mesmas condições e com todos os serviços não essenciais desabilitados (por exemplo: atualizações, antivírus, memória virtual). A suíte de testes para cada versão é executada, no mínimo, 10 vezes. Essa quantidade de execuções ajuda a ter medições de desempenho mais precisas em termos de tempo de execução.

3.1.2 Fase 2: Análise de Desvio

A segunda fase é a análise de desvio, que consiste em realizar a comparação do modelo de análise dinâmica, extraído durante a fase de análise dinâmica, para cada uma das duas versões do sistema. Essa comparação revela os cenários e métodos que foram degradados ou otimizados durante a evolução. O artefato de saída desta fase é um relatório contendo os cenários e métodos degradados ou otimizados em termos de tempo de execução [22]. Esta fase pode ser vista na Figura 6 adiante.

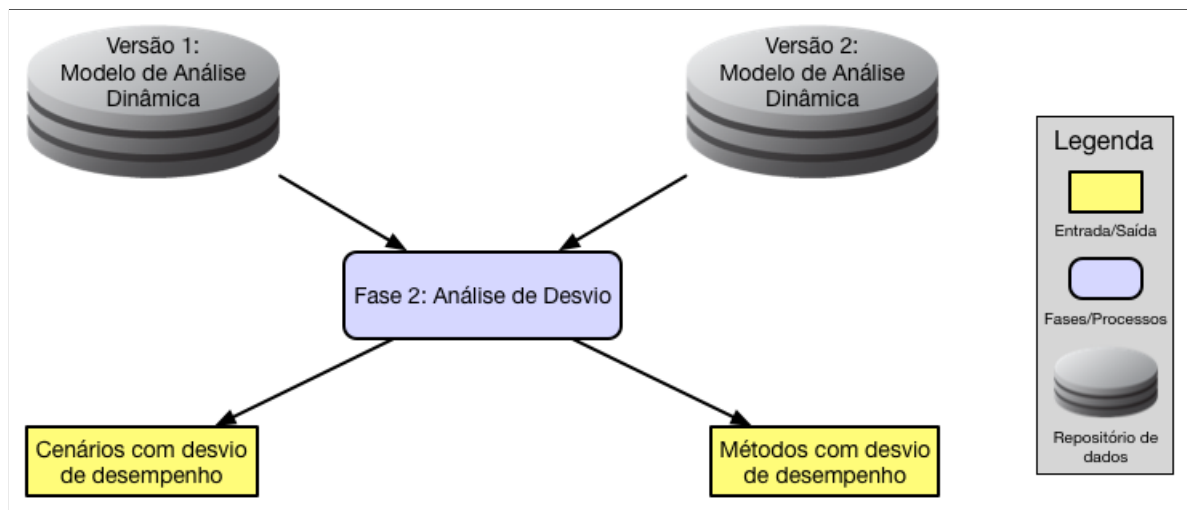


Figura 6: Fase 2 do PerfMiner.

Para realizar a comparação entre os tempos de execução, o *PerfMiner* pode utilizar duas estratégias: média aritmética e teste estatístico. A primeira compara a média do tempo de execução para cada método em ambas as versões. Se o valor da versão mais nova aumentou ou diminuiu mais do que um limiar configurado, é considerado que o método teve um desvio de desempenho. Já a segunda, usa um teste estatístico para observar se duas amostras independentes têm a mesma tendência. Neste caso, as amostras são formadas pelo conjunto dos valores dos tempos de execução para cada método comum em

cada cenário [22].

A estratégia de teste estatístico utilizado pela ferramenta é o Mann-Whitney U-Test [60]. Para esse teste, a ferramenta usa um valor padrão para o nível de significância (α) de 0,05. Dado um método, se o p -value calculado for igual ou menor do que o nível de significância, houve um desvio de desempenho para este método. Para os casos em que há o desvio, o tempo médio de execução é usado para determinar se houve uma degradação ou otimização. Embora, no geral, os desenvolvedores e arquitetos estejam interessados em degradações, a ferramenta também sinaliza as otimizações. Isso pode se tornar interessante, pois os desenvolvedores podem checar se algumas modificações esperadas realmente diminuam o tempo de execução [22].

3.1.3 Fase 3: Mineração de Repositório

A última fase realiza a mineração nos repositórios de controle de versões e gerenciador de tarefas com o intuito de encontrar os *commits* e tarefas que alteraram os métodos identificados na fase anterior. Para cada método detectado com desvio de desempenho (degradação ou otimização), esta fase recupera os *commits* do sistema de controle de versões. Se esse *commit* alterou linhas dentro do método detectado, o número da respectiva tarefa é procurado na mensagem de *commit*. O número da tarefa é usado para procurá-la no sistema de gerenciamento de tarefas em busca de informações extras, tais como o tipo da tarefa (defeito, melhoria, nova funcionalidade, etc) [22]. A Figura 7 ilustra essa fase.

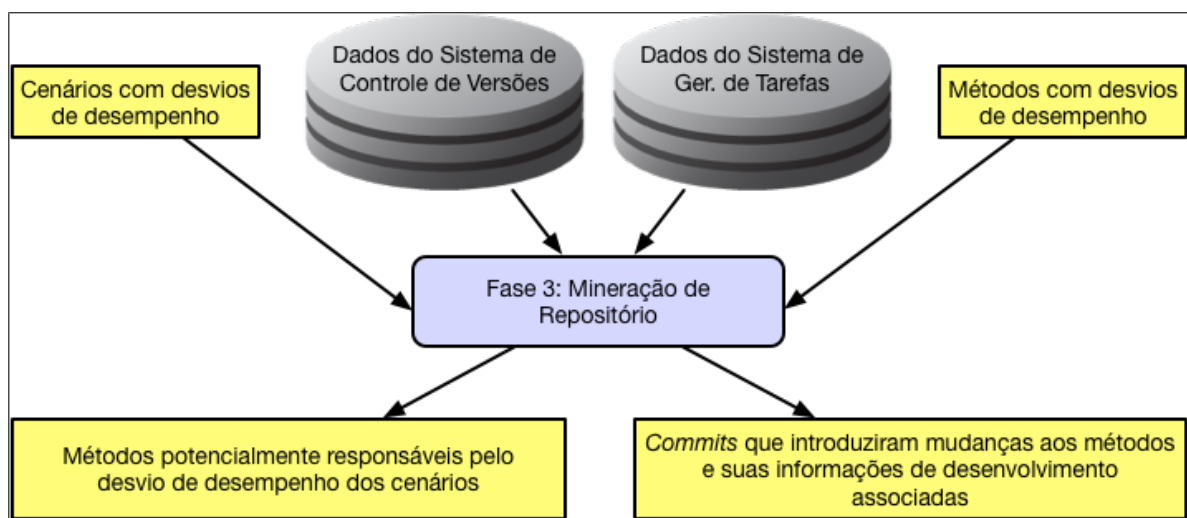


Figura 7: Fase 3 do PerfMiner.

É importante notar que os métodos identificados com desvios de desempenho nas fases anteriores, mas que não foram alterados durante a evolução, são também seleciona-

dos e armazenados, contudo, não estarão presentes no relatório final por, provavelmente, não representar causas reais do desvio de desempenho do cenário. Eles podem ter sido impactados por outros métodos na hierarquia de chamadas [22].

A abordagem completa do *PerfMiner* pode ser vista na Figura 8 adiante. Os artefatos de saída são usados como entradas para as fases seguintes, até a geração do relatório final. Os testes de sistemas em ambas as versões são considerados como os cenários para a análise e apenas os cenários e métodos comuns entre as versões são comparados.

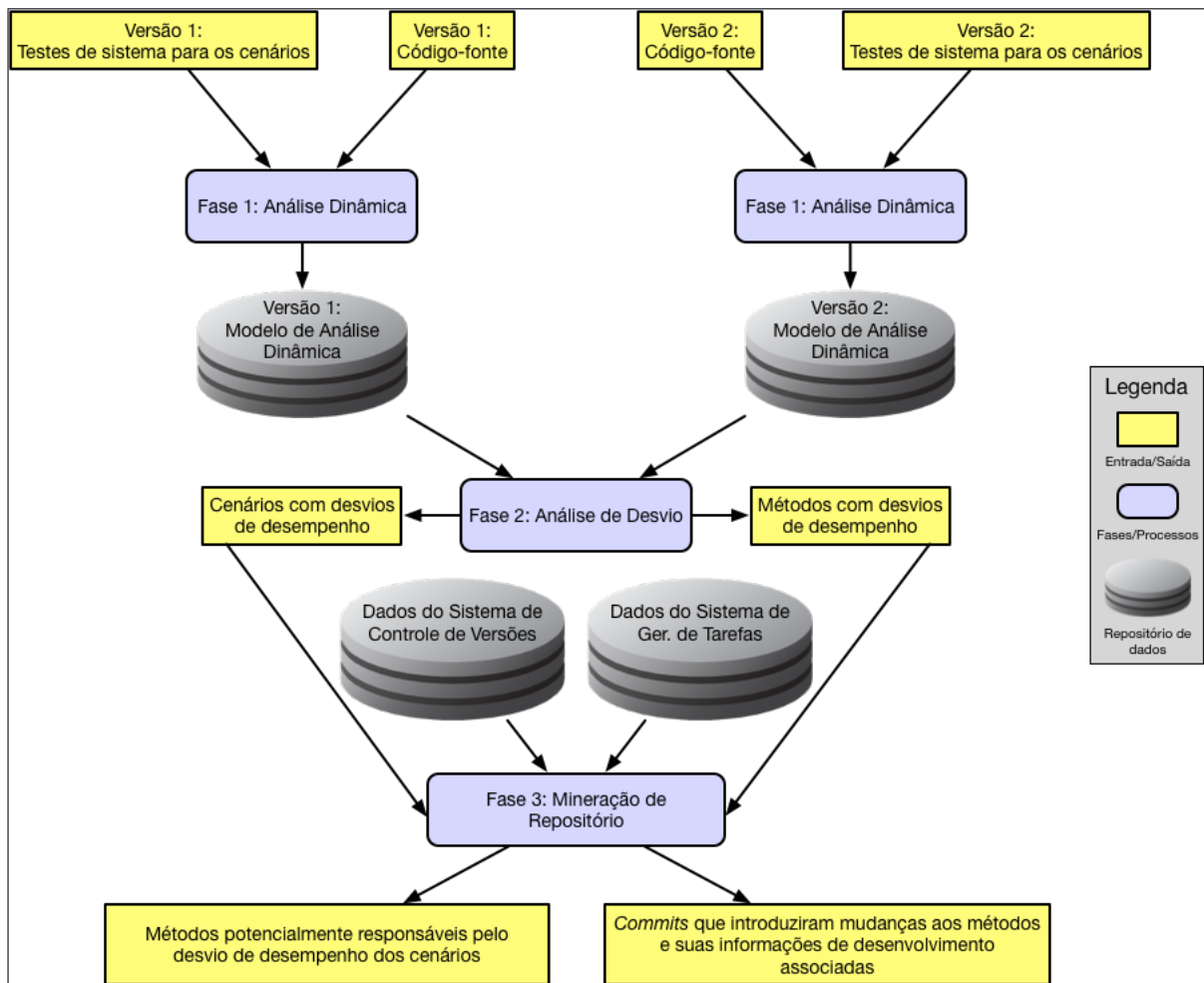


Figura 8: Abordagem completa do *PerfMiner*.

As visualizações propostas como extensão da ferramenta utilizam os artefatos de saída gerados após a execução completa da abordagem, caracterizando assim mais uma fase da ferramenta: a Fase 4. Como ilustra a Figura 9 a seguir, os artefatos de saída utilizados pelas visualizações são: (i) os modelos de análise dinâmica de ambas as versões, (ii) os relatórios de cenários e métodos com desvios de desempenho, (iii) os métodos potencialmente responsáveis pelo desvio de desempenho dos cenários e (iv) os *commits* que introduziram mudanças aos métodos, bem como suas informações de desenvolvimento associadas. Cada

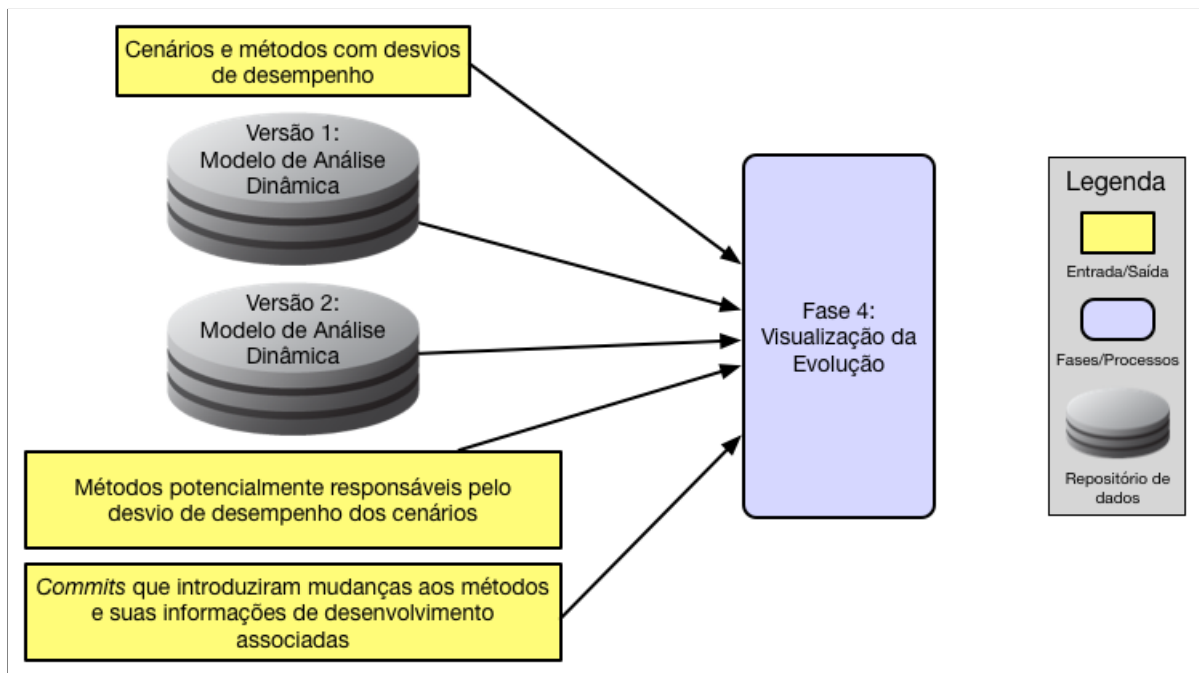


Figura 9: Fase 4 do PerfMiner.

uma das visualizações implementadas utilizam total ou parcialmente os artefatos gerados nas fases anteriores, de acordo com o seu propósito.

3.2 Visão Geral do PerfMiner Visualizer



A extensão ao *PerfMiner* foi implementada na forma de uma aplicação web, intitulada de *PerfMiner Visualizer*. O processamento do *PerfMiner* em suas fases 1, 2 e 3 não foi alterado, continuando *standalone*. A escolha por esse tipo de aplicação se deu pelo fato de sua execução ocorrer em um ambiente distribuído, onde cada parte que compõe a aplicação está localizada em locais diferentes. Por exemplo, a interface com o usuário reside em sua estação de trabalho, ao passo que o servidor e o banco de dados estão localizados em outro computador.

Por ser web, os usuários da aplicação, como os desenvolvedores e arquitetos, podem utilizá-la sem a necessidade de instalar nenhum módulo nas suas estações de trabalho. Essa é uma importante característica da extensão à ferramenta, fazendo com que esta se diferencie das outras ferramentas mencionadas neste trabalho. A ideia é que distribuição e facilidade de acesso façam com que a equipe de desenvolvimento acompanhe mais adequadamente a evolução do atributo de qualidade de desempenho.



A implementação foi feita utilizando o *framework* web Grails¹, banco de dados relacional PostgreSQL² e o *layout* das páginas foi implementado usando o *framework front-end* Bootstrap³. As visualizações foram geradas a partir de *scripts* em JavaScript através da biblioteca JointJS⁴. A estrutura das páginas web da ferramenta é dividida em três partes: canto superior, canto esquerdo e centro.

No canto superior situa-se uma barra de título, onde é apresentado o nome do sistema e o nome da página atual exibida. Na Figura 10, o nome do sistema é identificado como **PerfMiner Visualizer** ao passo que o nome da página é **Analyzed Systems**.

Já no canto esquerdo é encontrada uma barra de menus contendo dois itens:

- Nova análise (): a partir deste item o usuário pode iniciar uma nova análise, que será descrita com maiores detalhes posteriormente;
- Sistemas analisados (): é a página inicial mostrada na Figura 10 explicada adiante.

Ao centro está a área principal da aplicação, onde são exibidas os conteúdos das páginas acessadas pelo usuário. Na página inicial exibida na Figura 10, pode ser verificada uma tabela contendo todos os sistemas analisados. Cada linha da tabela representa uma análise, onde são exibidas: a versão anterior (*Version From*), versão posterior (*Version To*), *status* e ações (*Actions*). No exemplo da figura, dois sistemas foram analisados pela ferramenta: o sistema *System A* teve três análises e o sistema *System B* teve quatro análises.

Dependendo do *status* de cada análise, algumas ações pode ser tomadas. Caso o *status* seja **COMPLETED**, o usuário pode navegar até a visualização de sumarização de cenários () ou apagar a análise (). Caso seja **ERROR**, a única ação que pode ser tomada é a de apagar a análise para que esta possa ser realizada novamente. Se o *status* for **PENDING**, nenhuma ação pode ser efetuada uma vez que a análise ainda está em processamento.

3.2.1 Nova Análise

Para que o usuário da ferramenta obtenha acesso às visualizações é necessário que seja realizada uma análise dos artefatos gerados pelo *PerfMiner* onde, após o processamento,

¹<https://grails.org>

²<https://www.postgresql.org>

³<http://getbootstrap.com>

⁴<https://www.jointjs.com>

PV

≡

≡

+

+

New Analysis

PerfMiner Visualizer

Analyzed Systems

System A


Version From	Version To	Status	Actions
1.0	2.0	COMPLETED	<div><div></div><div></div></div>
2.0	3.0	COMPLETED	<div><div></div><div></div></div>
3.0	4.0	PENDING <div></div>	

System B

Version From	Version To	Status	Actions
1.0	2.0	COMPLETED	<div><div></div><div></div></div>
2.0	3.0	COMPLETED	<div><div></div><div></div></div>
3.0	4.0	COMPLETED	<div><div></div><div></div></div>
4.0	5.0	ERROR	<div><div></div></div>

Figura 10: Página inicial da aplicação.

sejam gerados os dados responsáveis por supri-las.

A partir do menu de Nova Análise () ou do botão com o mesmo nome e ícone situado na página inicial (Figura 10), o usuário pode navegar até a página responsável por essa funcionalidade, exibida na Figura 11.

Nessa página, o usuário precisa informar os dados e arquivos necessários para que uma análise seja feita. No primeiro campo, o nome do sistema ao qual se deseja realizar a análise é informado. Após isso, é informada a versão anterior do sistema, no campo **Previous Version**, e o seu respectivo arquivo de *backup* contendo os dados resultantes da análise do *PerfMiner*, no campo **Backup File - Previous Version**. Igualmente é feito para os campos **Current Version** e **Backup File - Current Version**, desta vez para a versão atual do sistema. Por fim, é necessário informar os arquivos contendo os cenários e métodos com desvio de desempenho, um para degradação de desempenho, no campo **PerfMinerDegraded Scenarios File**, e outro para otimização de desempenho, no campo **PerfMinerOptimized Scenarios File**.

Start a new analysis to generate the visualizations for given two versions of a system.

New Analysis

System

Jetty-Servlet

Previous Version

9.3.10

Current Version

9.3.11

PerfMiner Degraded Scenarios File

Escolher arquivo jetty_9.3.10_...13h42min.txt

PerfMiner Optimized Scenarios File

Escolher arquivo jetty_9.3.10_...13h42min.txt

Backup File - Previous Version

Escolher arquivo Jetty-Servlet-jetty-9.3.10.v20160621-db_10.backup

Backup File - Current Version

Escolher arquivo Jetty-Servlet-jetty-9.3.11.v20160721-db_10.backup

Cancel Submit

Figura 11: Página da funcionalidade de Nova Análise.

3.2.1.1 Funcionamento

O funcionamento geral desta funcionalidade é ilustrado na Figura 12. Vale salientar que esse processamento faz parte da fase 4 do *PerfMiner*, conforme exposto na Figura 9. No primeiro passo desse processo (passo 1), o usuário realiza a requisição solicitando que uma nova análise seja realizada, de acordo com os parâmetros e arquivos informados na página da Figura 11.

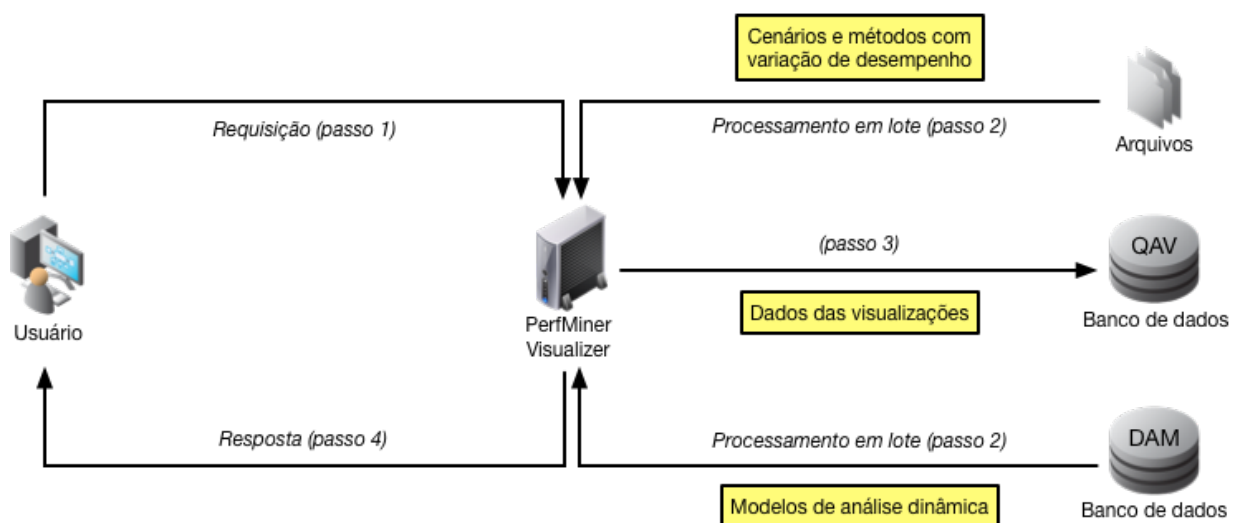


Figura 12: Funcionamento do processamento de uma nova análise.

Quando o servidor (*PerfMiner Visualizer*) recebe essa requisição, inicia-se o proces-

samento em lote (passo 2). Esse processamento recupera os arquivos contendo os cenários e métodos com desvio de desempenho que foram enviados pelo usuário, os arquivos de *backups* referentes a versão anterior e atual a ser analisada. Os bancos de dados são restaurados e, então, os modelos de análise dinâmica para cada uma das versões são recuperadas desse banco de dados (identificado por DAM – *Dynamic Analysis Model*). De posse de todos esses dados, é realizado um processamento para determinar todos os dados que dão suporte às visualizações oferecidas pela ferramenta. Após isso, no passo 3, os dados resultantes são salvos em um banco de dados (identificado por QAV – *Quality Attribute Visualization*). Ao final, o usuário recebe a resposta de que o processamento requisitado foi efetuado com sucesso, no passo 4.

Detalhadamente, no processamento em lote, iniciado no passo 2, os arquivos de saída do *PerfMiner* contendo informações sobre os cenários e métodos com desvio de desempenho são lidos e os modelos de análise dinâmica de ambas as versões são recuperados do banco de dados DAM. A partir de então, os dados para cada visualização começam a ser processados. Os passos 2 e 3 da Figura 12 são detalhados no diagrama de atividades da Figura 13.

De posse desses artefatos, os nós do grafo de chamadas são recuperados e a partir deles são determinados os métodos que foram adicionados ou removidos comparando os nós da versão atual com os da anterior. Depois disso, a partir dos métodos indicados pelos arquivos de saída do *PerfMiner*, são determinados os nós com desvios de desempenho, seja degradação ou otimização.

Na ferramenta, os nós identificados como adicionados, removidos e com desvio de desempenho são candidatos a serem exibidos. Optou-se por apresentar apenas os nós com desvio, nós adicionados, removidos e poucos nós sem desvio de desempenho, mas que ajudam a tornar o grafo legível, de modo que poucos nós são renderizados no navegador do usuário. Essa decisão levou em consideração a quantidade de nós de um cenário, que pode passar dos milhares, o desempenho da própria aplicação web e um possível ganho no entendimento da visualização do grafo de chamadas por parte do usuário.

Após determinar os nós com desvios de desempenho, são criados nós de agrupamento para representar os que não serão exibidos na visualização. Depois disso, os tempos médios dos nós a serem exibidos são calculados, levando em consideração todas as ocorrências do método no cenário.

Para a visualização da sumarização de cenários, os dados gerais de todos os cenários são calculados e, juntamente com os dados da visualização do grafo de chamadas calcu-

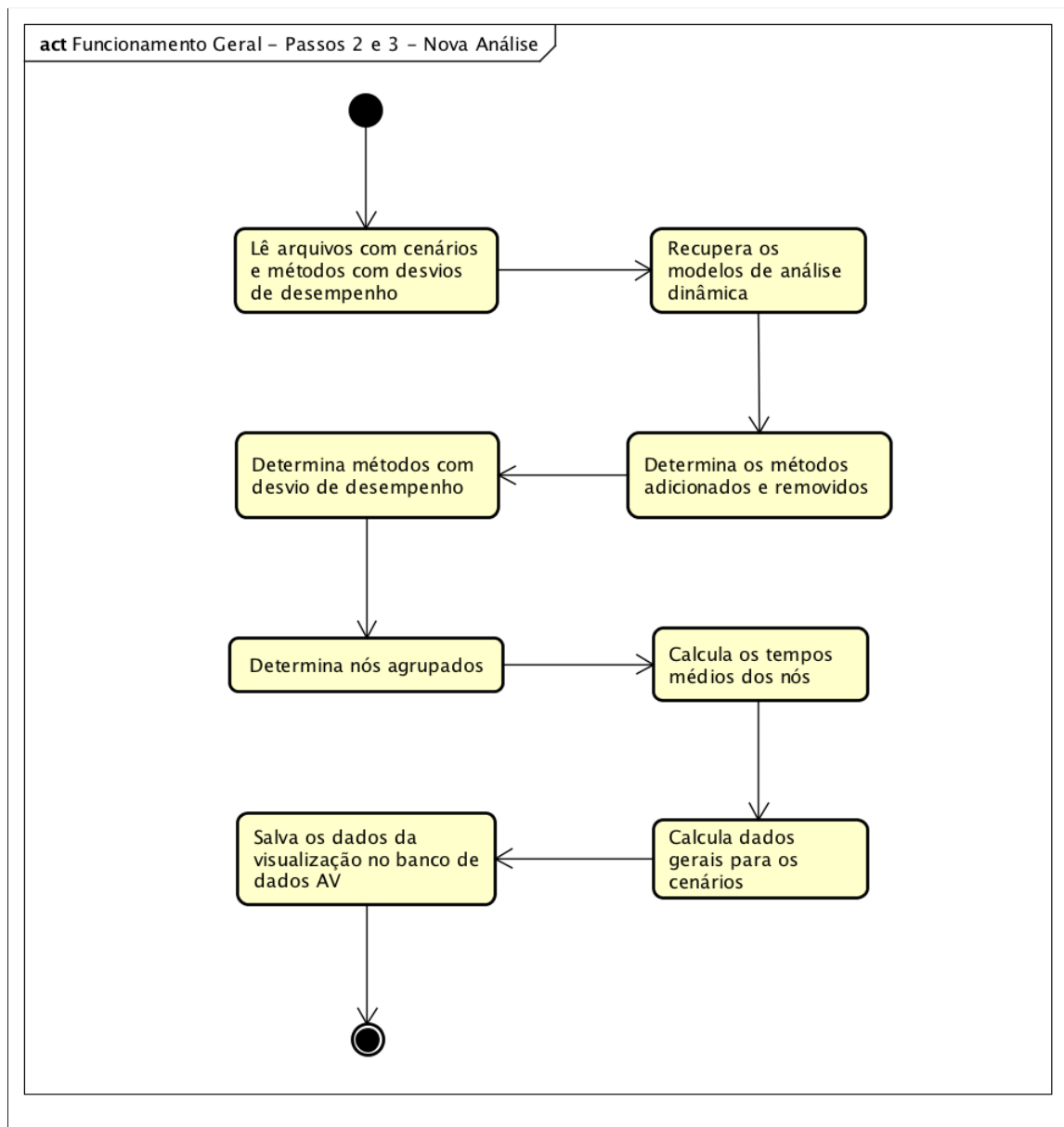


Figura 13: Passos 2 e 3 da realização de uma nova análise.

lados anteriormente, são salvos no banco de dados QAV no passo 3, marcando o fim do processo.

É importante frisar que o processamento em lote é executado apenas uma vez para duas versões de um sistema, sendo necessário para gerar os dados que possibilitam aos usuários um rápido tempo de resposta na requisição das visualizações.

3.2.1.2 Bancos de Dados

Os bancos de dados DAM e QAV mencionados anteriormente têm funções distintas. O DAM é utilizado pelo *PerfMiner* para armazenar os dados coletados na análise dinâmica.

Já o QAV é próprio da ferramenta *PerfMiner Visualizer* e armazena os dados que dão suporte às visualizações, funcionando também como um *cache*, minimizando o tempo de resposta do servidor para o usuário.

DAM - *Dynamic Analysis Model*

O resultado do processamento da fase 1 do *PerfMiner* gera um modelo de análise dinâmica que contém informações sobre os *traces* de execução do sistema, como mencionado na subseção 3.1.1. Esse modelo é persistido no banco de dados DAM, sendo usado, posteriormente, na extensão proposta por este trabalho para gerar as visualizações. A Figura 14 mostra o diagrama de classes parcial desse modelo.

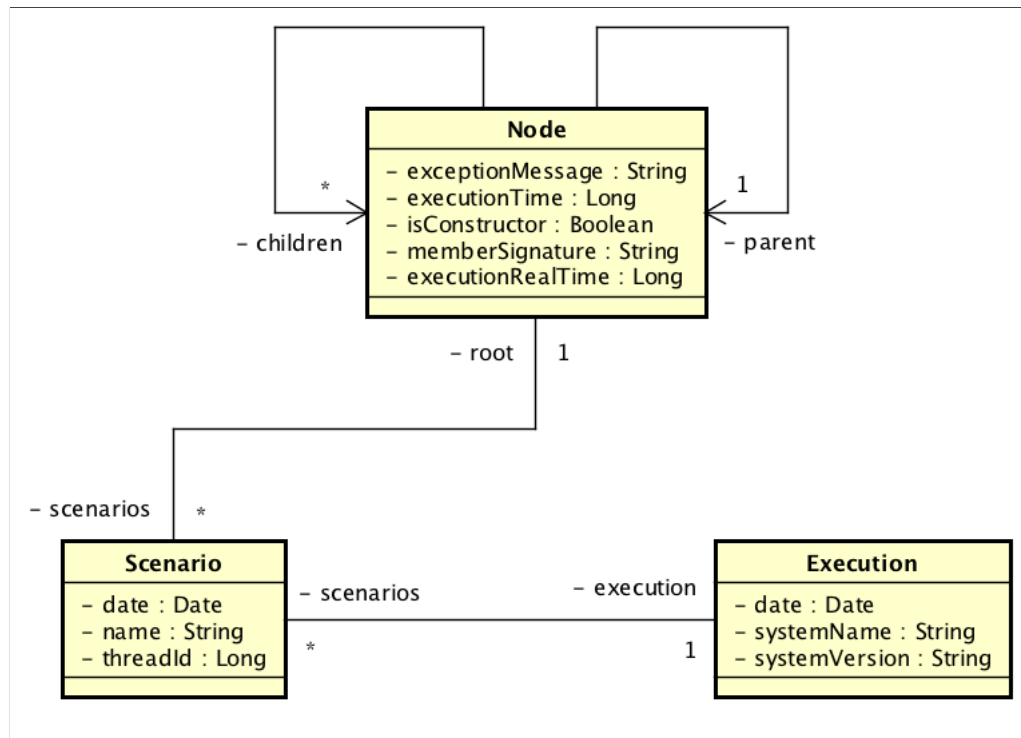


Figura 14: Diagrama de classes parcial do *PerfMiner*.

A classe **Execution** representa uma execução específica do *PerfMiner* para um sistema. Possui atributos que indicam o nome do sistema, a versão e a data que a análise foi iniciada. Já a classe **Scenario** é o modelo para cada cenário executado. Os cenários têm um nome, uma data que indica quando foi analisado, o *id* da *thread* que o executou, o contexto para representar requisições web (em caso de sistemas web) e o nó raiz.

A classe **Node** representa todo membro (método ou construtor) executado dentro de um cenário em particular. Essa classe possui atributos para indicar a assinatura do método ou construtor, uma mensagem se alguma exceção acontecer durante a execução, o tempo de execução total do membro, o tempo de execução do próprio membro, uma propriedade

booleana que indica se o nó representa ou não um construtor e dois autorrelacionamentos: um para indicar o nó pai e outro para indicar os nós filhos. Uma atenção especial merece ser dada aos dois tempos de execução: a propriedade `executionRealTime` indica o tempo de execução do próprio método ou construtor, sem considerar os tempos de execução dos nós filhos. Já a propriedade `executionTime` representa o tempo total de execução do membro, considerando os tempos dos membros filhos.

QAV - *Quality Attribute Visualization*

A ferramenta faz uso de outro banco de dados além do DAM: o QAV. Essa base de dados é responsável por armazenar os dados que darão suporte às visualizações. A Figura 15 a seguir mostra o diagrama de classes para esse modelo.

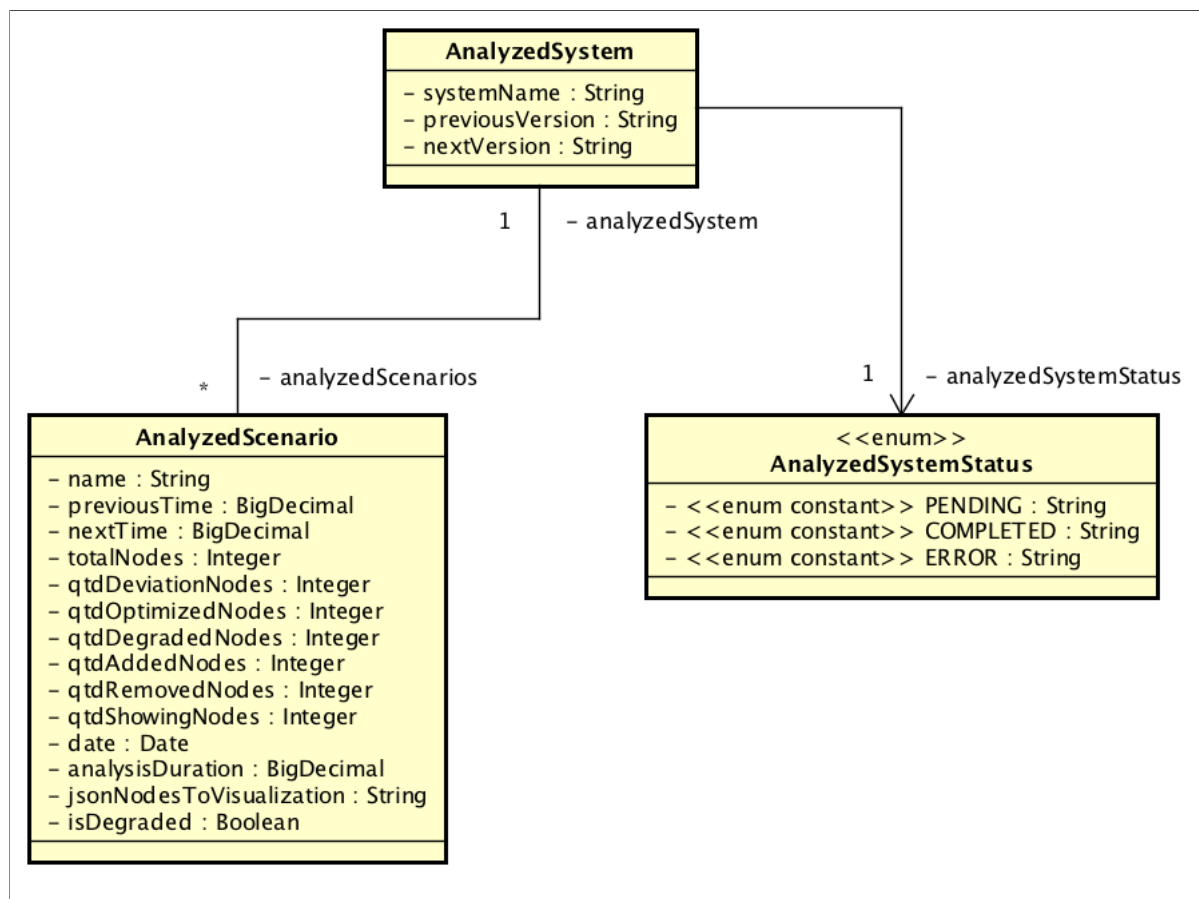


Figura 15: Diagrama de classes da extensão proposta.

A classe `AnalyzedSystem` representa uma análise para dado sistema, em determinadas versões. Possui propriedades para o nome do sistema, versão anterior e versão posterior. Cada análise possui um *status*, definido pela propriedade `analyzedSystemStatus` dessa classe, do tipo `AnalyzedSystemStatus`. Essa propriedade pode assumir um dos três valores a seguir: `PENDING` (análise em andamento), `COMPETED` (análise finalizada), `ERROR`

(análise com erro - necessário novo processamento).

A classe `AnalyzedScenario` reflete cada cenário analisado de determinadas versões. Possui propriedades para o nome do cenário, tempos de execução na versão anterior e posterior, total de nós do grafo de chamadas, quantidade de nós otimizados, degradados, adicionados, removidos e exibidos no grafo, data e hora do processamento da análise, um atributo booleano que indica se o cenário é de otimização ou degradação e um atributo que armazena o JSON que dá suporte à visualização do grafo de chamadas. Embora existam atributos que armazenem dados referentes a visualização do grafo de chamadas, essas classes foram modeladas de modo a dar suporte às duas visualizações propostas neste trabalho.

3.3 Visualização da Sumarização de Cenários

A Sumarização dos Cenários permite ao usuário obter uma visualização dos cenários com desvios de desempenho entre duas versões do sistema. Para essa visualização foi utilizada uma variação do gráfico de rosca, que por sua vez é uma derivação do de pizza. O gráfico de pizza é considerado um gráfico de informação simples cujo principal objetivo é mostrar a relação de uma parte com o todo [61]. No contexto da visualização apresentada, o todo se configura como sendo todos os cenários com desvios de desempenho para as versões analisadas, e as partes são cada cenário com suas características.

A altura, a largura e a cor de uma fatia do gráfico são as características visuais que possuem significado nessa visualização:

- *Largura*: a largura de uma fatia do gráfico indica a porcentagem de desvio de desempenho do cenário na versão atual relacionado com a versão anterior. Quanto mais larga a fatia, maior foi o desvio de desempenho. De maneira contrária, quanto mais fina a fatia, menor o desvio;
- *Altura*: a altura da fatia indica o tempo de execução do cenário na versão atual. Quanto mais alta a área preenchida, maior o tempo de execução. De maneira contrária, quanto mais baixa, menor o tempo de execução;
- *Cor*: cada fatia do gráfico possui uma cor que indica se houve degradação ou otimização no desempenho do cenário. A cor marrom claro indica que o cenário foi degradado, ao passo que a cor verde indica que ele foi otimizado em relação a versão anterior.

A Figura 16 exibe uma visão geral desta visualização. Na parte superior, é possível notar 5 blocos com as seguintes informações: nome do sistema, versões analisadas (anterior e atual), quantidade de cenários com desvios de desempenho, quantidade de cenários com degradação e quantidade de cenários com otimização. Na parte central, encontra-se o gráfico. Nele, existem 5 divisões, onde cada uma delas representa um cenário e possui largura, altura e cor. Na parte superior direita, há uma legenda explicativa sobre altura, a largura e as cores das fatias.



Figura 16: Visão geral da Sumarização de Cenários.

A partir do gráfico da Figura 16, podem ser vistos 4 cenários com degradação de desempenho e 1 cenário com otimização. Dos cenários com degradação, 3 deles possuem os maiores tempos de execução perante o restante (altura da fatia). A respeito do cenário com otimização, pode-se notar que possui baixo tempo de execução com relação aos demais (altura da fatia) e tem o menor desvio de desempenho com relação à versão anterior analisada (largura da fatia). O cenário indicado pela letra A foi o cenário com maior tempo de execução dentre os analisados, como evidencia a altura da fatia. Já o cenário indicado pela letra B foi o que possuiu a maior variação de desempenho, destacado pela largura da fatia.

3.3.1 Interação

O gráfico desta visualização é passível de ações do usuário, a fim de (i) obter maiores informações sobre determinado cenário ou (ii) avançar para a visualização do grafo de chamadas, descrita mais adiante neste trabalho. Para a primeira ação, o usuário deve posicionar o ponteiro do *mouse* sobre uma das fatias do gráfico e então um *tooltip* é apresentado, como pode ser visto na Figura 17.

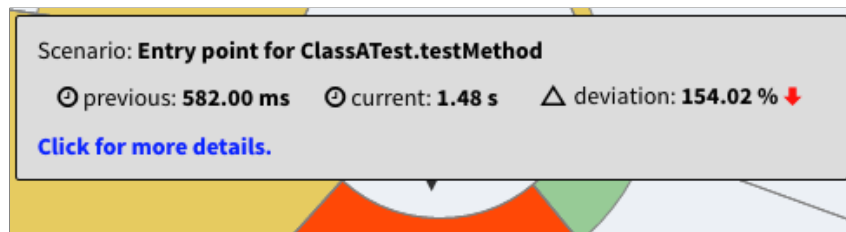


Figura 17: *Tooltip* com maiores informações sobre determinado cenário.

No *tooltip* mostrado, é possível verificar o nome do cenário: **Entry point for ClassA Test.testMethod**; o seu tempo de execução na versão anterior: 582,00 milissegundos; o tempo de execução na versão atual: 1,48 segundos; e a porcentagem de desvio do tempo de execução da versão atual em relação a versão anterior: uma degradação de 154,02%. É possível concluir que houve uma degradação através da seta vermelha para baixo. Em caso de seta verde para cima, significa que houve uma otimização de desempenho. Ao clicar em uma das fatias do gráfico, o usuário será levado para a visualização do grafo de chamadas.

3.3.2 Funcionamento

As visualizações apresentadas neste trabalho seguem o mesmo padrão, apresentado na Figura 18. Os usuários somente terão acesso às visualizações quando uma análise para a versão desejada já tiver sido processada.

A partir da listagem das análises realizadas na página inicial da aplicação, exibida na Figura 10, o usuário pode clicar no ícone desta visualização (🔍) para acessá-la. Ao clicar, é realizada uma requisição para a aplicação, solicitando acesso à visualização de sumarização de cenários (passo 1). O servidor (*PerfMiner Visualizer*) recebe essa requisição e, a partir dos dados referentes ao sistema e versões desejadas, recupera do banco de dados QAV (passo 2) os dados necessários para a visualização. Após isso, é retornado para o *browser* do usuário um arquivo em formato JSON (*JavaScript Object Notation*) contendo os dados recuperados (passo 3). O *browser*, por fim, recebe, interpreta esse ar-

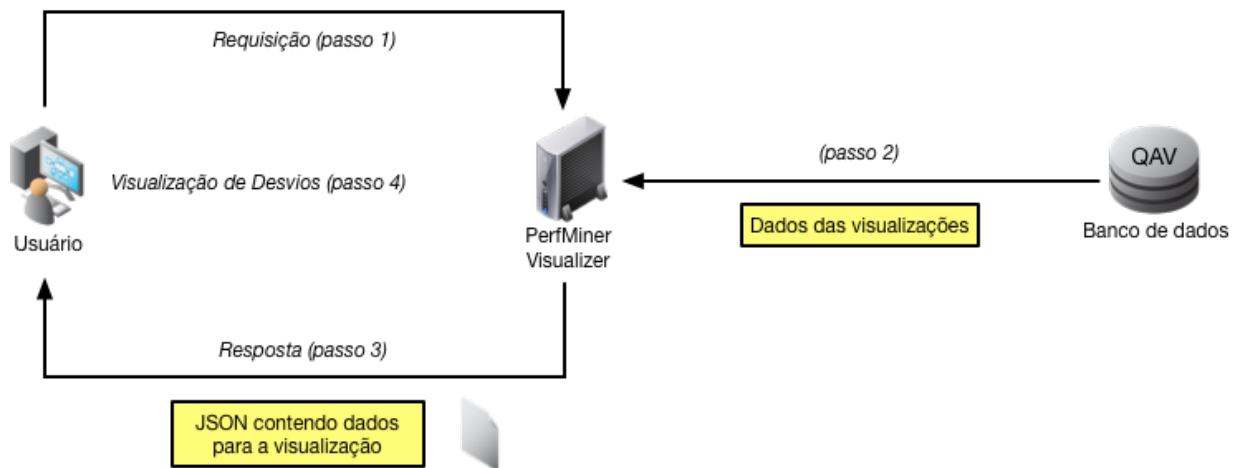


Figura 18: Funcionamento geral das visualizações.

quivo e monta a visualização com as devidas informações para que a renderização seja feita com sucesso (passo 4).

O processamento realizado quando o usuário recebe a resposta está exposto na Figura 19. Ao receber o arquivo JSON do servidor, o navegador define a área de desenho que abrigará o gráfico baseado na altura e largura do monitor do usuário. Após isso, é determinado, dentre os cenários indicados pela análise, qual o que possuiu o maior tempo de execução. Esse tempo é utilizado para estabelecer o limite das fatias do gráfico. Isto posto, pelo menos um cenário preencherá por completo uma das fatias do gráfico. Na sequência, a cor, largura e altura, além dos dados do *tooltip* de cada fatia são determinadas baseadas nos dados dos respectivos cenários. Por fim, o gráfico é criado e renderizado.

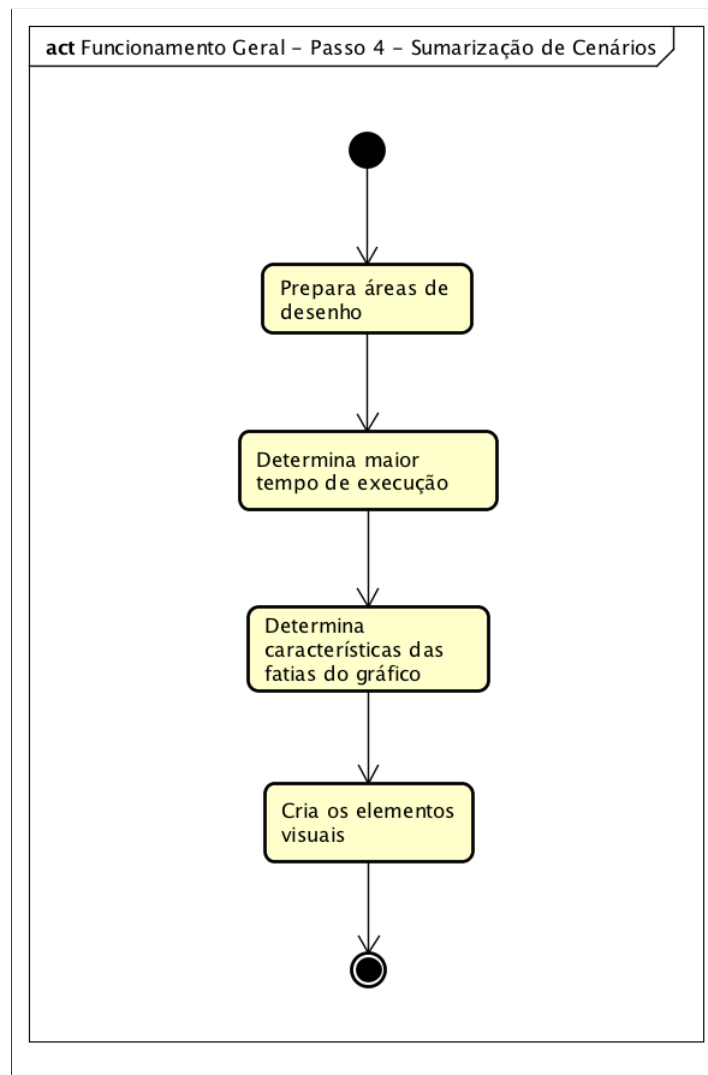


Figura 19: Detalhamento do passo 4 da Figura 18, na Sumarização de Cenários.

3.4 Visualização do Grafo de Chamadas

Esta visualização tem o objetivo de mostrar, para dadas duas versões de um sistema, os métodos que potencialmente causaram o desvio de desempenho para um determinado cenário. Esses métodos são exibidos em um grafo direcionado de chamadas com propriedades visuais que destacam quais dos métodos mostrados tiveram desvios de desempenho.

Os grafos podem ser utilizados quando os dados a serem representados são estruturados, ou seja, quando existe uma relação inerente entre os elementos de dados a serem visualizados [62]. Há uma vasta quantidade de áreas onde os grafos podem ser aplicados, por exemplo: hierarquia de arquivos em formato de árvore, mapas de sites, mapas moleculares e genéticos, diagramas de fluxos de dados, entre outros [62].

Nesse sentido, em uma chamada de métodos é evidente a relação entre eles, uma vez

que os objetos, em um sistema orientado a objetos, trocam mensagens entre si através da invocação dos métodos. O grafo é exibido utilizando o *layout* em árvore, onde os nós filhos estão dispostos abaixo dos nós ancestrais, e a direção dos nós é *top-down*. Um exemplo do grafo é exibido na Figura 20. Nesta seção serão detalhadas as suas propriedades visuais, o seu funcionamento básico, além de descrever a seção de Sumário e Histórico desta visualização.

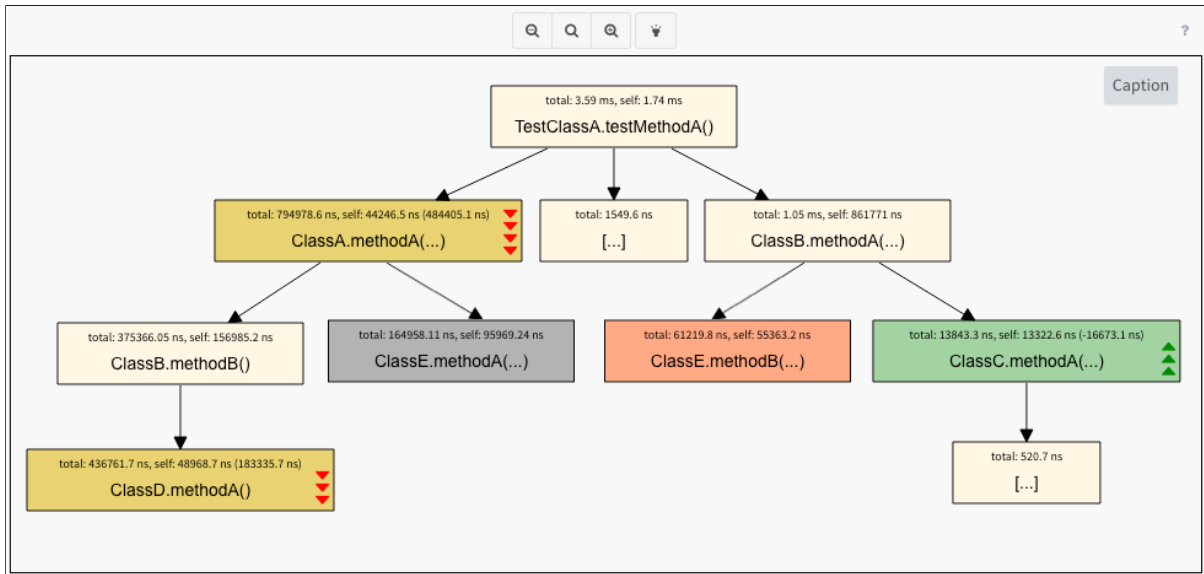


Figura 20: Exemplo do grafo de chamadas.

3.4.1 Sumário

A seção Sumário está disponível acima da seção do Grafo de Chamadas e o seu intuito é trazer outras informações à visualização visando auxiliar os usuários na contextualização do cenário exibido.

Na Figura 21, é mostrada uma visão geral sobre determinado cenário sendo exibido na visualização. Para a seção *Description*, as informações exibidas são:

- *Scenario*: exibe o nome do cenário analisado;
- *System*: nome do sistema analisado;
- *Versions*: versões do sistema que foram analisadas. É mostrado no formato *<versionA>* to *<versionB>*. Espera-se que a versão B, neste caso, seja posterior a versão A;
- *Time*: mostra o tempo de execução do cenário na versão mais recente. A diferença de tempo entre as duas versões vai guiar o que será exibido na barra de títulos da

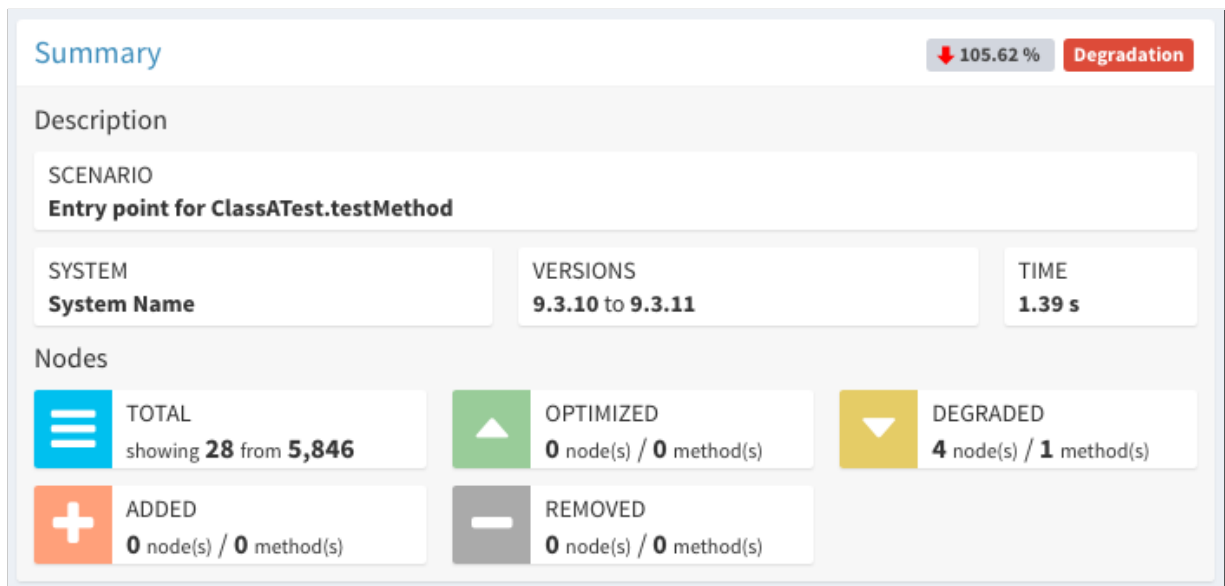


Figura 21: Seção de sumário do grafo de chamadas.

figura. No exemplo mostrado, o tempo na última versão degradou com relação ao tempo na versão anterior, sendo exibido, no canto superior direito, a porcentagem da degradação (105,62%) acompanhada de uma seta vermelha para baixo e um rótulo em vermelho marcando o cenário como degradado (*degradation*). Caso o tempo do cenário seja otimizado, é exibida uma seta verde para cima ao lado da porcentagem da variação, além de um rótulo verde marcando o cenário como otimizado (*optimization*).

Para a seção *Nodes*, as informações são as seguintes:

- *Total*: mostra o número de nós que estão sendo mostrados no grafo de chamadas e total de nós do cenário. Vale salientar que cada nó representa um método executado durante a hierarquia de chamadas do cenário. Pode acontecer de nós diferentes representarem o mesmo método. Isso pode acontecer porque o mesmo método pode ter hierarquias de chamadas diferentes;
- *Optimized*: exibe o número de nós e métodos que tiveram otimização de desempenho. Para que um determinado nó seja considerado com otimização de desempenho, ele (i) tem que estar presente nas duas versões analisadas e (ii) o tempo de execução na versão posterior tem que ter sido menor do que na versão anterior;
- *Degraded*: exibe o número de nós e métodos que tiveram degradação de desempenho. Para que um determinado nó seja considerado com degradação de desempenho, ele

(i) tem que estar presente nas duas versões analisadas e (ii) o tempo de execução na versão posterior tem que ter sido maior do que na versão anterior;

- *Added*: apresenta o número de nós e métodos que foram adicionados da versão anterior para a posterior. Ou seja, são os métodos que não existiam na execução do cenário para a versão anterior e passaram a existir na execução da versão posterior. O número de métodos pode ser menor do que o número de nós, uma vez que nós diferentes podem representar o mesmo método;
- *Removed*: o contrário do conceito anterior. São apresentados os nós e métodos que foram removidos da versão anterior para a posterior.

Para as informações sobre os nós otimizados, degradados, adicionados e removidos, o número de métodos pode ser menor do que o número de nós, uma vez que nós diferentes podem representar o mesmo método se eles estiverem em hierarquias de chamadas diferentes.

3.4.2 Histórico

A segunda parte da visualização do Grafo de Chamadas apresenta o histórico da evolução do desempenho, em termos de tempo de execução, de determinado cenário. A exibição do histórico é em formato de um gráfico de linha, onde no eixo X são apresentadas as dez últimas versões analisadas em que o cenário esteve presente e no eixo Y é mostrado o tempo de execução. A Figura 22 adiante mostra um exemplo desse gráfico.

Na figura, pode-se concluir que o cenário em questão esteve presente nas versões 9.3.10, 9.3.11, 9.3.12, 9.3.13, 9.3.16, 9.4.0, 9.4.1 e 9.4.2. Entre as versões 9.3.10 e 9.3.11, nota-se uma forte degradação, praticamente duplicando o seu tempo de execução. Por outro lado, da versão 9.3.11 para a 9.3.12, houve uma otimização importante, o que resultou em um tempo de execução próximo ao da versão 9.3.10. Da 9.3.12 a 9.4.2 pequenas degradações são notadas no desempenho do cenário.

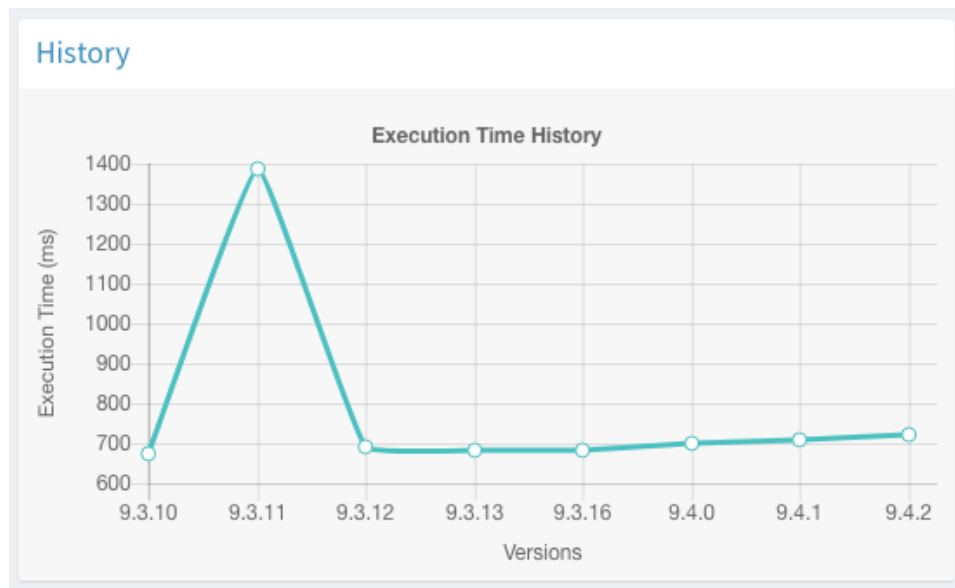


Figura 22: Seção de histórico do grafo de chamadas.

3.4.3 Grafo de Chamadas

A terceira e última parte da visualização é o grafo de chamadas propriamente dito. Esse grafo, como mencionado, é composto de nós e arestas direcionadas que expõem os caminhos das chamadas dos métodos para o cenário analisado. Os nós, representados por caixas, possuem algumas características visuais, apresentadas adiante.

3.4.3.1 Nó sem desvio

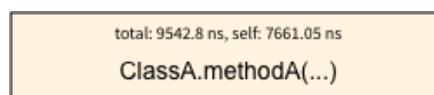


Figura 23: Nó que representa um método sem desvio de desempenho.

O primeiro e mais básico tipo de nó é o que representa um método que não teve desvios de desempenho para o cenário e versões analisadas, apresentado na Figura 23. As características desse nó são:

- *Cor*: esta é a principal característica que diferencia os nós uns dos outros. No caso deste nó, a cor é marrom claro;
- *Nome do nó*: posicionado ao centro, são mostrados o nome da classe e o nome do método executado, no formato `ClassName.methodName()`. Para otimizar e evitar a exibição de grande quantidade de texto, o nome do pacote e os parâmetros do

método foram ocultados, sendo estes, quando houver, representados por três pontos (...). Essa definição serve para todos os tipos de nós dessa visualização;

- *Tempos de execução*: localizados no canto superior do nó, são apresentados dois tempos de execução: à esquerda, o tempo total do nó, que representa a soma dos tempos de execução dos nós filhos com o tempo do próprio nó. No exemplo da figura, o tempo total foi de 9.542,8 nanossegundos; e à direita, o tempo do próprio nó, desconsiderando o tempo dos nós filhos. Na figura, o tempo foi de 7.661,05 nanossegundos.

3.4.3.2 Nó degradado

A visualização também é capaz de apontar se determinado nó teve o seu desempenho degradado com relação à versão anterior. A Figura 24 apresenta um nó degradado:

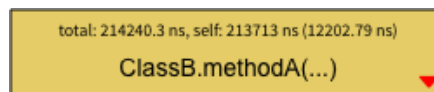


Figura 24: Nó que representa um método com degradação de desempenho.

As características visuais deste nó são bem semelhantes às apresentadas para os nós otimizados:

- *Cor*: os nós que degradaram o desempenho são mostrados em laranja amarelo;
- *Tempos de execução*: além dos tempos total e do próprio nó, um nó com desvio de desempenho, seja otimização ou degradação, apresenta a diferença entre os tempos de execução da versão anterior para a posterior entre parênteses, no canto superior direito. No exemplo da figura, o nó degradou o seu tempo em 12.202,79 nanossegundos;
- *Setas*: no canto inferior direito são mostradas setas indicativas de quão forte ou fraca foi a variação de desempenho. No caso de nós com degradação, são exibidas setas vermelhas apontadas para baixo, onde cada uma delas representa 25% de desvio do tempo com relação à versão anterior. Sendo assim, uma única seta representa um desvio de 0 a 25%, duas setas de 25% a 50%, três setas de 50% a 75% e quatro setas de 75% ou superior. No exemplo apresentado na Figura 24, a degradação se deu entre 0 e 25% do tempo de execução em relação ao desempenho anterior.

3.4.3.3 Nó otimizado

Quando um cenário possui nós que otimizaram o seu desempenho em relação a versão anterior, outros elementos visuais são acrescentados à visualização. A Figura 25 a seguir apresenta um nó otimizado:

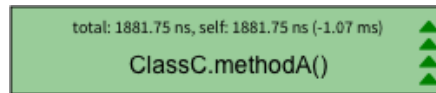


Figura 25: Nó que representa um método com otimização de desempenho.

As características visuais deste nó, além das comentadas anteriormente, são:

- *Cor*: os nós que otimizaram o atributo de qualidade desempenho são mostrados em um tom de verde;
- *Tempos de execução*: também são exibidos os tempos de execução total, do próprio nó e o desvio de desempenho. No exemplo, o nó otimizou o tempo em 1,07 milissegundos;
- *Setas*: as setas indicativas de variação de desempenho para nós de otimização são apontadas para cima, de cor verde. No exemplo, a otimização se deu em mais de 75% do desempenho em relação à versão anterior.

3.4.3.4 Nó adicionado

Um cenário pode apresentar, para diferentes versões, nós de chamadas de métodos que estão presentes na versão atual, mas que não existiam na versão anterior: são os nós adicionados. A visualização representa esses nós como mostra a Figura 26. A descrição dos atributos visuais é a que segue:

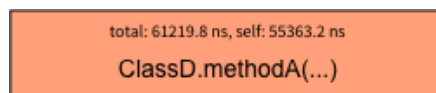


Figura 26: Nó que representa um método adicionado.

- *Cor*: os nós que foram adicionados na versão atual são mostrados na cor salmão claro;
- *Tempos de execução*: os tempos de execução total e do próprio nó são exibidos. No exemplo, o tempo total do nó é de 61.219,8 nanossegundos e o tempo do próprio nó é de 55.363,2 nanossegundos.

3.4.3.5 Nó removido

De maneira contrária ao que foi relatado sobre o nó adicionado, um cenário pode não possuir, na versão atual, chamadas de métodos que estavam presentes na versão anterior. Significa que essas chamadas, para a hierarquia de chamadas de métodos do cenário, foram removidas. No entanto, vale salientar que esse fato não representa que o método em si (código-fonte) foi removido. A Figura 27 mostra a representação de um nó removido no grafo. A descrição dos atributos visuais é a que segue:

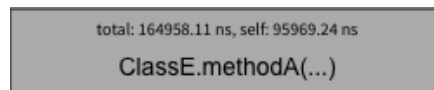


Figura 27: Nó que representa um método removido.

- *Cor*: os nós que foram removidos na versão atual possuem cor em um tom de cinza;
- *Tempos de execução*: os tempos de execução total e do próprio nó na versão anterior são exibidos. No exemplo, o tempo total do nó é de 164.958,11 nanossegundos e o tempo do próprio nó é de 95.969,24 nanossegundos.

3.4.3.6 Nó de agrupamento

Além dos que representam métodos com ou sem desvio de desempenho, e métodos adicionados e removidos, há um tipo de nó que representa um agrupamento de vários outros: o agrupado. Assim, os nós que não estão diretamente relacionados com nós de desvio, adicionados ou removidos, são omitidos e agrupados em um único nó, exceto os nós pai e, se houver, nós avôs, bisavôs, tataravôs e pentavôs. A Figura 28 a seguir ilustra esse tipo de nó, seguido da descrição dos seus atributos visuais.

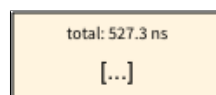


Figura 28: Nó que representa um agrupamento de outros nós.

- *Cor*: os nós de agrupamento são mostrados na cor marrom claro;
- *Nome do nó*: posicionado ao centro, é exibido o texto [...];
- *Tempos de execução*: é exibido o tempo de execução total que representa a soma de todos os tempos de execução dos nós contidos no agrupamento. No exemplo, o tempo total é de 527,3 nanossegundos.

Algoritmo de Supressão de Nós Exibidos

O nó agrupado é criado a partir da execução de um algoritmo de supressão de nós exibidos. Esse algoritmo calcula os nós que devem ser apresentados para o usuário. Dessa forma, em um grafo de chamadas para determinado cenário, são exibidos os nós que têm relevância para o entendimento e identificação dos métodos possivelmente responsáveis pelo desvio de desempenho do cenário.

O algoritmo parte da premissa de que todos os nós que representam os métodos com algum desvio de desempenho (degradação ou otimização), adicionados ou removidos devem ser exibidos para o usuário. A partir desses nós, é verificado se existem repetições deles, ou seja, se, para a mesma hierarquia de chamadas, um mesmo nó foi executado novamente, ele será marcado como repetido e terá uma borda espessa na visualização. Assim, evita-se que a hierarquia seja criada novamente na visualização para representar o mesmo nó. Casos como esse podem acontecer em loops, por exemplo. A Figura 29 adiante exemplifica um nó com uma borda espessa.

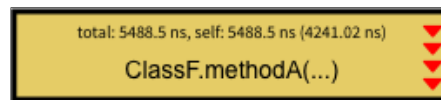


Figura 29: Borda espessa de um nó, representando múltiplas execuções.

Uma vez identificadas todas as repetições, o algoritmo verifica todos os nós que podem ser transformados em nós de agrupamento. Os casos em que esse tipo de nó ocorre são: (i) durante a hierarquia de chamadas, (ii) em nós filhos dos nós a serem exibidos, (iii) em hierarquias de chamadas adjacentes não diretamente relacionadas com os nós a serem exibidos.

Durante a hierarquia de chamadas a um nó com desvio de desempenho, adicionado ou removido, pode existir uma grande quantidade de nós. Para mostrar o referido nó e reduzir a quantidade de nós a serem exibidos no grafo, a maioria dos existentes nessa hierarquia de chamadas são agrupados em um único nó. A Figura 30 ilustra esse caso, onde o nó com desvio de desempenho a ser exibido é o `DefaultConverters.findConverterType(...)` e o indicado com a letra A é o nó agrupado que representa vários outros nós nessa hierarquia de chamadas.

Há caso em que existem nós filhos daqueles que possuíram algum tipo de desvio. Caso nenhum desses nós filhos também possua uma das características dos nós apresentáveis, eles serão agrupados para diminuir a quantidade de nós a serem exibidos. Esse caso é exemplificado na Figura 31, onde o nó degradado `Exclusions.shouldSkipField(...)`

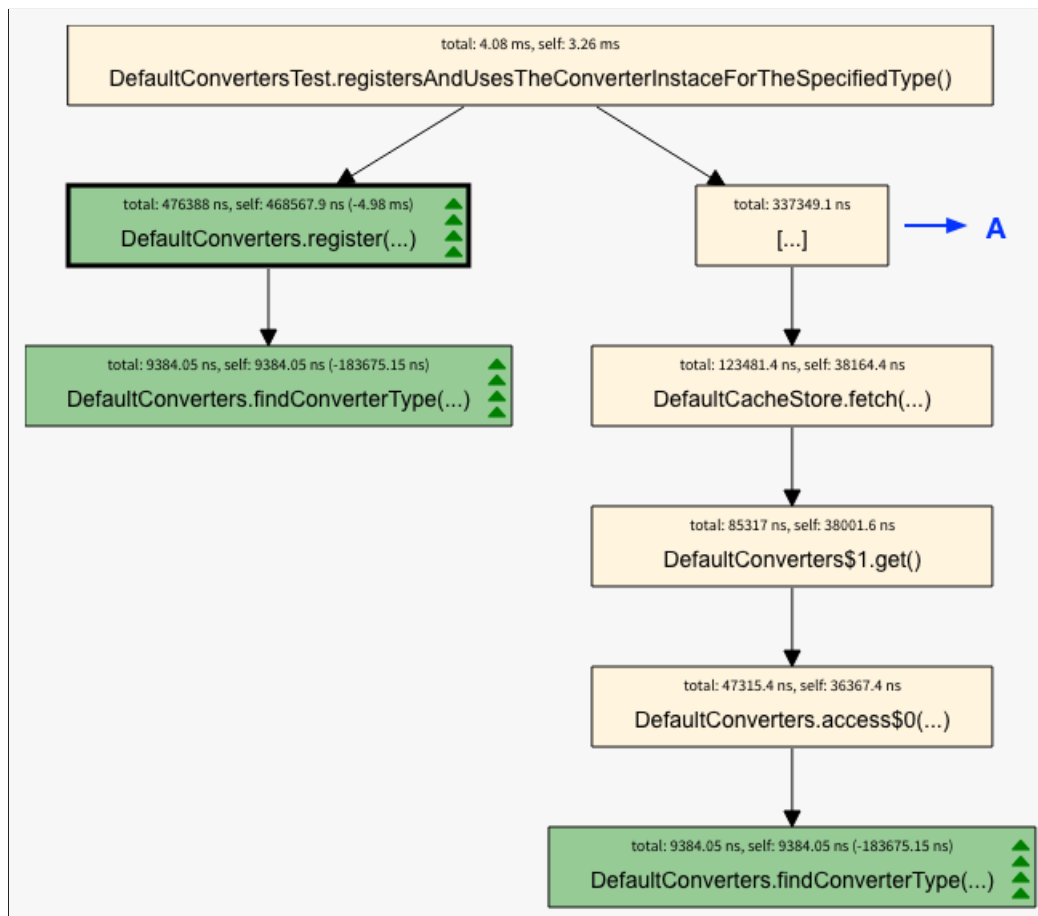


Figura 30: Exemplo de nós agrupados durante a hierarquia de chamadas.

possui filhos, no entanto, dentre eles não possui nenhum outro nó com prioridade a ser exibido (nós com desvios, adicionados ou removidos). Assim sendo, os nós filhos foram agrupados no nó indicado pela letra B.

As hierarquias de chamadas adjacentes que não estão diretamente relacionadas com os nós a serem exibidos são agrupadas em um único nó. Na Figura 31, o nó indicado com a letra A mostra que há uma ou mais hierarquias de chamadas agrupadas.

As propriedades visuais apresentadas para a visualização do grafo de chamadas incluem, em suma, três partes: (i) seção de sumário, contendo informações gerais sobre o cenário; (ii) seção de histórico, mostrando o histórico do desempenho do cenário dentre as dez últimas versões analisadas; e (iii) seção do grafo de chamadas, onde a execução dos métodos é representada através de nós e arestas. Esta seção é a mais importante desta visualização, contendo características visuais que diferem os nós de acordo com sua classificação. Um resumo dessas características é apresentado na Tabela 1 a seguir.

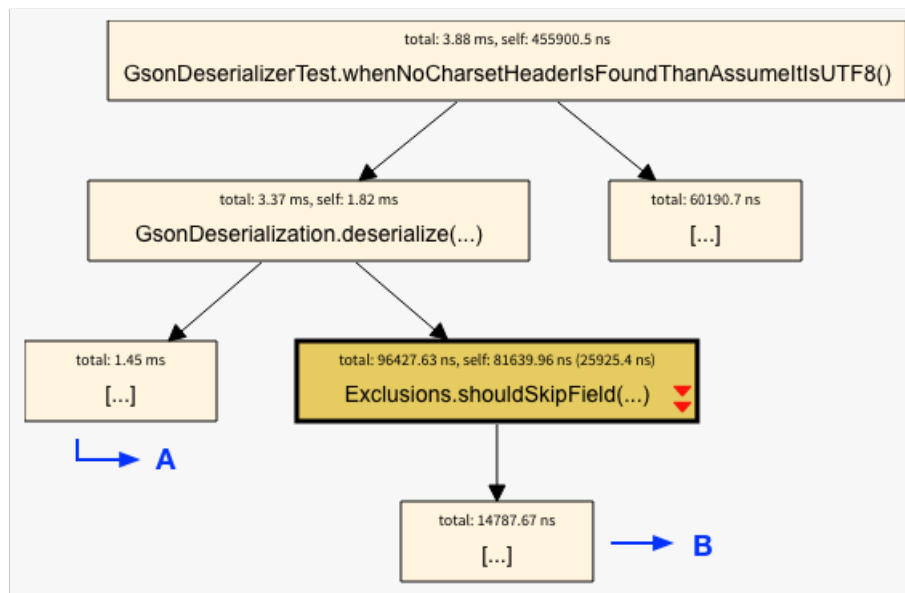


Figura 31: Exemplo de nós hierárquias em árvores adjacentes (A) e em nós filhos (B).

Tabela 1: Resumo das características visuais dos nós.

Tipo	Cor	Tempos de Execução	Setas
Sem desvio	Marrom claro	Total e próprio	Sem setas
Degradado	Vermelho claro	Total, próprio e desvio	Vermelhas, para baixo
Otimizado	Verde claro	Total, próprio e desvio	Verdes, para cima
Adicionado	Salmão claro	Total e próprio	Sem setas
Removido	Cinza	Total e próprio	Sem setas
Agrupado	Marrom claro	Total	Sem setas

3.4.4 Interação

Esta visualização exibe um conjunto de informações que a torna capaz de indicar os métodos que potencialmente foram responsáveis por desvios de desempenho de um determinado cenário. Contudo, o usuário pode interagir com o grafo efetuando o efeito de zoom, destacando o caminho de execução de determinado nó com desvio ou obtendo mais informações sobre um nó passando o cursor do *mouse* sobre ele.

O zoom pode ser acessado pela barra de ferramentas mostrada na Figura 32. Essa barra situa-se logo acima da área de desenho do grafo (como pode ser visto na Figura 20) e exibe os botões marcados de A a D. Os botões A, B e C podem ser clicados para realizar o efeito de zoom. O botão A realiza o efeito de distanciamento (*Zoom Out*), o botão B redefine o zoom para o estado inicial (100%) e o botão C efetua o efeito de aproximação (*Zoom In*). O zoom pode fazer com que o grafo seja comportado novamente na área de desenho.

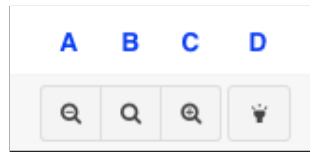


Figura 32: Barra de ferramentas do grafo de chamadas.

O destaque do caminho de execução de um nó com desvio esmaece todos os nós que não estão no caminho de execução do nó desejado. Existem duas formas de se obter esse efeito: a primeira delas é a partir do botão D da barra de ferramentas. Esse botão, ao ser clicado, destaca os caminhos para todos os nós com desvio mostrados no grafo, como exemplifica a Figura 33. Para desfazer o efeito, o botão deve ser clicado novamente; a segunda forma de se obter esse destaque é efetuando um duplo clique em algum dos nós com desvio exibido no grafo. Isso fará com que o caminho do nó raiz até o nó clicado seja destacado, esmaecendo todos os outros. Para reverter o efeito, nessa situação, o usuário pode realizar o duplo clique em qualquer nó sem desvio de desempenho exibido no grafo.

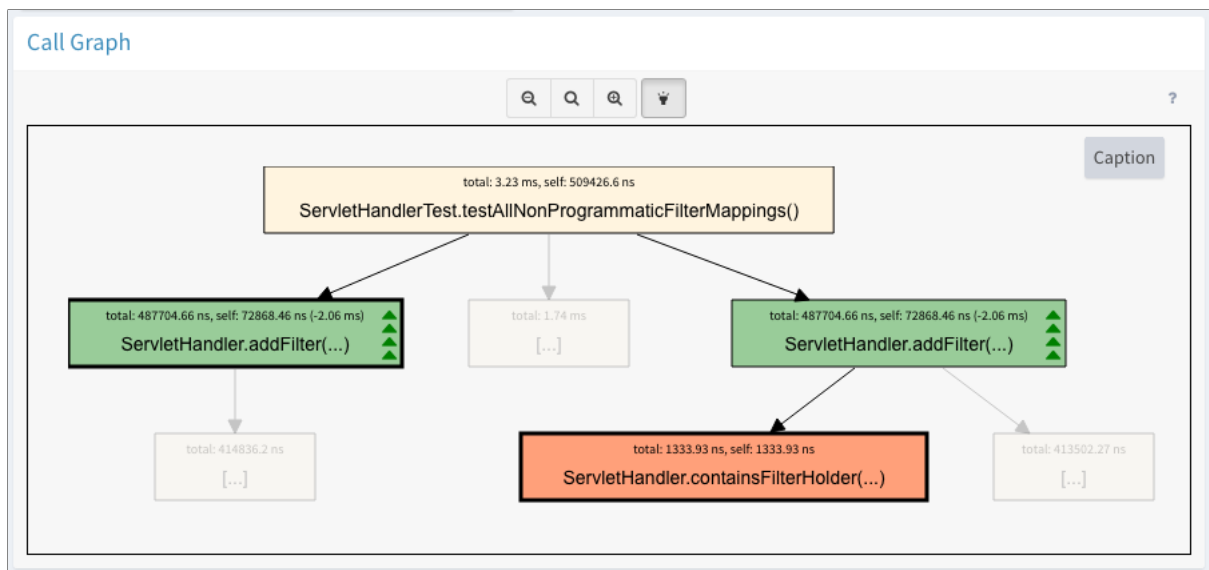


Figura 33: Efeito de destaque do caminho de execução de nós com desvio.

Uma legenda explicativa também foi implementada para ajudar na interpretação do grafo. O usuário pode acessá-la clicando no botão **Caption**, no canto superior direito do grafo. Ao ser clicada, a legenda é expandida, exibindo os elementos visuais e seus significados, conforme mostrado na Figura 34.

O usuário pode obter mais detalhes sobre cada nó exibido no grafo. Para isso, ele deve posicionar o ponteiro do mouse sobre o nó desejado e aguardar meio segundo. Após esse tempo, é exibido um *tooltip* com essas informações. A Figura 35 exemplifica os detalhes de um nó.

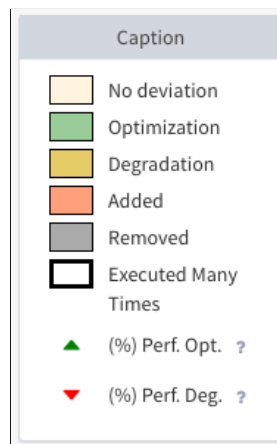


Figura 34: Legenda do grafo de chamadas.

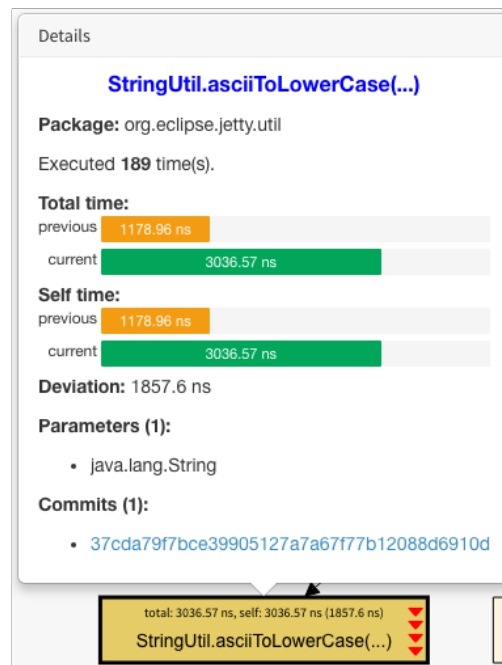


Figura 35: Detalhes de um nó no grafo de chamadas.

Na figura, podem ser encontradas informações sobre o pacote do método representado, a quantidade de vezes que o nó foi executado em caso de execução múltipla, o tempo total na versão anterior e atual, o tempo do próprio nó na versão anterior e atual, o tempo de desvio de desempenho, a listagem de parâmetros do método e a listagem de *commits* que possivelmente causaram o desvio de desempenho do nó. Dependendo do tipo de nó ao qual se quer obter os detalhes, podem existir mais ou menos as informações a serem apresentadas.

3.4.5 Funcionamento

Após a execução de uma nova análise, conforme apresentado na seção 3.2.1, o funcionamento desta visualização segue o funcionamento geral das visualizações mostrado na Figura 18, no entanto, no passo 4 dessa figura, existem processamentos específicos a fim de tratar os dados antes de serem exibidos para o usuário.

Seguindo o fluxo da Figura 18, a partir da visualização de sumarização de cenários, o usuário pode clicar em uma das fatias daquele gráfico e realizar uma requisição para a visualização do grafo de chamadas (passo 1). O servidor recebe essa requisição e, a partir dos parâmetros enviados, recupera do banco de dados QAV (passo 2) os dados necessários para a visualização. Após isso, é retornado para o navegador do usuário um arquivo em formato JSON contendo os dados recuperados (passo 3). O navegador, por fim, recebe, interpreta esse arquivo e monta a visualização com as devidas informações para que a renderização seja feita com sucesso (passo 4).

No passo 4 da Figura 18, há um procedimento executado no navegador do usuário que recebe e processa o arquivo JSON e, ao final, renderiza o grafo de chamadas. Esse processo está exposto em detalhes na Figura 36 adiante. O início ocorre ao receber o arquivo do servidor e a primeira atividade é criar a área de desenho que abrigará o grafo. Nessa atividade são considerados a altura e largura do monitor do usuário, de modo que a área útil de apresentação seja compatível com a área do monitor.

Após isso, as características dos nós contidos no arquivo JSON são determinadas. Para cada um deles, são determinados: (i) a altura e largura da sua caixa, que é diretamente proporcional ao tamanho do nome a ser exibido. Vale salientar que o pacote da classe é omitido para a exibição do nome; (ii) a espessura da borda, caso o nó tenha sido executado mais de uma vez; (iii) a cor, que depende do seu tipo; (iv) os tempos de execução total e, caso pertinente, o do próprio nó e o de desvio; (v) as setas, que dependem do seu tipo e da porcentagem de desvio do tempo de execução ocorrida de uma versão para outra; e (vi) os detalhes a serem exibidos no *tooltip*, que dependem do tipo do nó.

Depois, as arestas são determinadas de acordo com a relação de nós pais e filhos contidos no arquivo. Elas são direcionadas do nó pai para os filhos, indicando a hierarquia de chamadas.

Com a definição dos nós e suas características, e das arestas, a próxima atividade do processo é criar os elementos visuais na área de desenho, para, posteriormente, determinar o *layout* (conforme citado anteriormente, o *layout* utilizado é em árvore) e posicionamento

dos nós. Por fim, o grafo de chamadas é renderizado para o usuário.

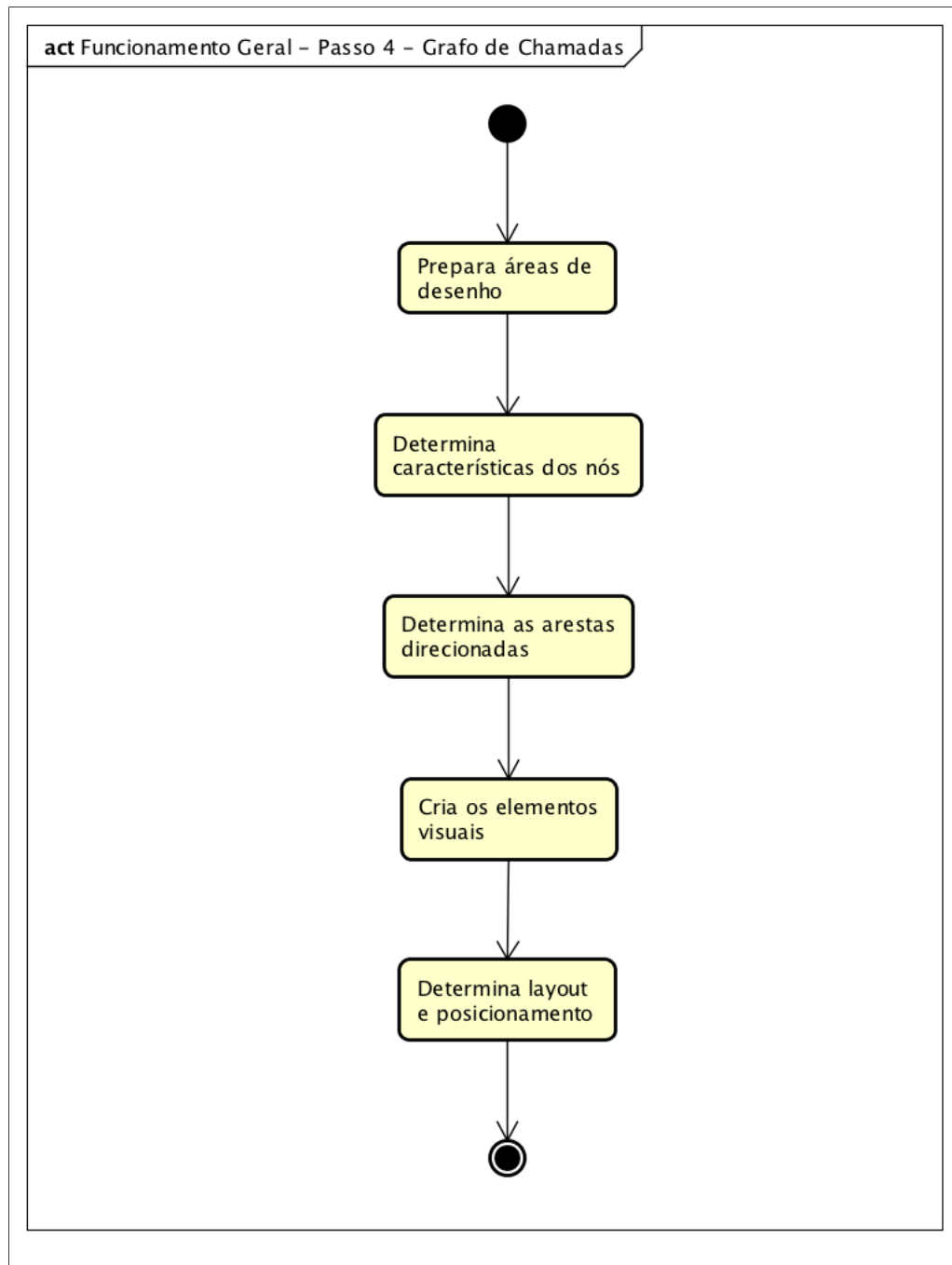


Figura 36: Detalhamento do passo 4 da Figura 18, no Grafo de Chamadas.

3.5 Considerações

Durante o processo de manutenção, as modificações podem ter consequências negativas, diminuindo a qualidade do software, aumentando a sua complexidade além de, também, poder afetar o desempenho dos sistemas ao longo do tempo [10][11][12]. Este

capítulo apresentou o *PerfMiner Visualizer* e suas visualizações, que visam tornar direta e clara a monitoração do atributo de qualidade de desempenho. Uma vez que as ferramentas existentes se mostram complexas ou ineficientes para essa finalidade, as visualizações foram propostas a serem implementadas como extensões da ferramenta *PerfMiner*. Em suma, são as seguintes:

- (i) *Sumarização dos Cenários*: a sumarização dos cenários com desvio de desempenho objetiva mostrar, de maneira sucinta, quais cenários analisados tiveram desvio de desempenho, seja degradação ou otimização;
- (ii) *Grafo de Chamadas*: a visualização do grafo de chamadas exhibe, para cada cenário, as chamadas dos métodos que tiveram desvio de desempenho. Dessa forma, o usuário pode localizar em qual trecho de código da execução do cenário houve o desvio.

A visualização da sumarização de cenários permite aos usuários ter uma visão geral de todos os cenários indicados com desvios de desempenho entre uma determinada versão e a anterior. A visualização mostra os cenários em um gráfico de rosca, onde cada cenário é representado por uma fatia dessa rosca, e todos os cenários compõem a integralidade do gráfico. Dessa forma, os usuários podem identificar se os cenários exibidos são de degradação ou otimização, identificar o cenário com maior tempo de execução dentre os exibidos e o que possuiu a maior porcentagem de desvio de desempenho.

Foi mostrado os detalhes da visualização do grafo de chamadas, que visa exhibir, para dadas duas versões de um sistema, os métodos que potencialmente causaram o desvio de desempenho para um determinado cenário. Nesta visualização, os métodos são apresentados em um grafo direcionado de chamadas com propriedades visuais que destacam quais dos métodos mostrados tiveram desvios de desempenho. Com essa visualização, os usuários podem identificar as possíveis causas de desvios de desempenho em determinado cenário através da listagem de *commits* presentes em cada nó com desvio de desempenho, adicionado ou removido.

4 Avaliação

Este capítulo descreve o estudo empírico realizado em dois sistemas *open source* de diferentes domínios cujos objetivos são avaliar se as visualizações são úteis para representar os desvios de desempenho encontrados durante a análise desses sistemas, avaliar se os usuários conseguem encontrar as informações sobre os desvios através das visualizações oferecidas e a verificar a possível aplicabilidade da ferramenta como parte integrante dos processos de desenvolvimento dos sistemas analisados.

O restante deste capítulo está organizado como segue: a seção 4.1 apresenta como o estudo foi projetado; a seção 4.2 exhibe os resultados do estudo e a seção 4.3 conclui o capítulo, apresentando as principais contribuições do estudo, as ameaças à validade e as limitações da avaliação aplicada.

4.1 Projeto do Estudo Empírico

Esta seção inclui a descrição do projeto do estudo, dividida da seguinte maneira: os objetivos e questões de pesquisa estão relatados na subseção 4.1.1; os sistemas analisados e os critérios para escolha, na subseção 4.1.2; os procedimentos do estudo estão mostrados na subseção 4.1.3 e a caracterização dos participantes do estudo na subseção 4.1.4.

4.1.1 Objetivos e Questões de Pesquisa

Os principais objetivos deste estudo são investigar se a ferramenta *PerfMiner Visualizer*, com suas visualizações e propriedades visuais oferecidas, é capaz de representar as evoluções de desempenho dos cenários analisados, se os usuários conseguem identificar esses desvios através das representações visuais oferecidas e se é útil para as equipes de desenvolvimento dos sistemas analisados. Ao utilizar a ferramenta, os usuários poderão identificar os cenários com desvios de desempenho e, a partir da análise do grafo de chamadas de cada um deles, tomar conhecimento sobre os *commits* dos métodos que

possivelmente foram os responsáveis pelo desvio. A partir de então, ações podem ser tomadas pela equipe de desenvolvimento para sanar possíveis problemas no desempenho das aplicações. O estudo foi guiado pelas seguintes questões de pesquisa:

- QP1. Os cenários identificados com desvios de desempenho são apresentados claramente nas visualizações implementadas pela ferramenta?** A expectativa é que esta questão de pesquisa seja respondida através da visualização da sumarização de cenários, uma vez que ela é responsável por dar uma visão geral dos cenários identificados com desvios de desempenho após a análise de duas releases de um sistema. Nessa visualização, os usuários podem identificar, prioritariamente, o cenário com maior/menor porcentagem de desvio, o cenário com maior/menor tempo de execução e o tipo de desvio (degradação ou otimização) dos cenários. Espera-se que os cenários com desvios apontados pela ferramenta, para cada versão, sejam apresentados pela visualização de maneira clara, sem prejuízos para a sua identificação.
- QP2. O algoritmo de redução de nós aplicado pela ferramenta é capaz de reduzir significativamente o número de nós no grafo de chamadas para sistemas reais?** A quantidade de nós de cada cenário das releases analisadas são suficientemente grandes para dificultar ou inviabilizar o tamanho da hierarquia de chamadas na visualização do grafo de chamadas. A expectativa desta questão de pesquisa é a de que o algoritmo de redução de nós exibidos (apresentado na subseção 3.4.3.6) implementado reduza consideravelmente a quantidade de nós a serem exibidos para o usuário, diminuindo a complexidade da visualização e permitindo que os métodos com desvios de desempenho sejam identificados, juntamente com as possíveis causas de tais desvios.
- QP3. Desenvolvedores e arquitetos conseguem identificar os cenários com variações de desempenho e suas potenciais causas, em termos de métodos e *commits*, através do auxílio visual da ferramenta?** Através da ferramenta, uma vez executada uma análise para um par de versões de um sistema, os usuários podem identificar os cenários que tiveram algum desvio de desempenho pela visualização da sumarização de cenários e conhecer suas possíveis causas (métodos e *commits*) através da visualização do grafo de chamadas. A expectativa é que os participantes do questionário online (contribuidores dos projetos Jetty e VRaptor) consigam identificar corretamente essas características através dos elementos visuais fornecidos pelas visualizações.

QP4. Há indícios de que as visualizações implementadas pela ferramenta são mais eficazes do que os dados tabulares para se encontrar informações sobre os cenários, métodos e *commits*? Os participantes do questionário online foram organizados em dois grupos, conforme será destacado adiante na subseção de Procedimentos (4.1.3). Cada grupo respondeu a uma visualização através da ferramenta e a outra visualização através de dados tabulares. Através da comparação dessas respostas, a expectativa é obter indícios se as visualizações implementadas são mais eficazes do que os dados tabulares para encontrar as informações solicitadas pelo questionário.

4.1.2 Sistemas

Neste estudo foram usados dois sistemas *open source* de diferentes domínios selecionados através dos seguintes critérios:

- Ser *open source*;
- Ser desenvolvido na linguagem de programação Java;
- Ter no mínimo dez releases (no momento do início do estudo);
- Possuir casos de testes automatizados utilizando a biblioteca JUnit;
- Utilizar o Maven¹ como ferramenta de gerenciamento de dependências e compilação;
- Possuir repositório no GitHub.

A partir desses critérios, foram escolhidos os sistemas Jetty [63] e o VRaptor [64]. O Jetty é um servidor web e um *servlet* contêiner Java capaz de fornecer conteúdo estático e dinâmico a partir de instâncias *standalone* ou embutidas. As dez últimas releases do Jetty foram 9.3.10, 9.3.11, 9.3.12, 9.3.13, 9.3.14, 9.3.15, 9.3.16, 9.4.0, 9.4.1 e 9.4.2.

O VRaptor é um *framework* MVC para desenvolvimento web em Java que visa trazer alta produtividade para desenvolvimento com CDI (*Contexts and Dependency Injection*). As dez últimas releases do VRaptor foram 4.0.0.Final, 4.1.0.Final, 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.2.0.RC1, 4.2.0.RC2, 4.2.0.RC3 e 4.2.0.RC4.

¹<http://maven.apache.org>

4.1.3 Procedimentos

Esta subseção descreve os procedimentos seguidos após a escolha dos sistemas, organizados sequencialmente nos passos a seguir.

4.1.3.1 Passo 1 - Coleta e Preparação dos Dados

Após a escolha dos sistemas e suas releases, os códigos-fonte de todos eles foram baixados do repositório com a finalidade de executar as três fases do *PerfMiner*. Após o download, cada uma das versões foi configurada para compilar e executar sem erros, uma vez que a fase 1 necessita exercitar a aplicação através dos testes automatizados. Todos os testes executados são considerados como pontos de entrada de cenários. Para o cálculo do desvio de desempenho, o *PerfMiner* não considera quaisquer desvios a partir de classes de teste, somente de classes do código-fonte da aplicação.

De posse dos códigos-fonte, cada release dos projetos foi configurada para dar suporte ao *AspectJ* e para incluir as bibliotecas do *PerfMiner*, no entanto, de maneira não intrusiva, sem qualquer alteração no código-fonte.

4.1.3.2 Passo 2 - Aplicação da Abordagem

Após isso, para cada versão de cada sistema, todos os casos de testes automatizados foram executados para que os dados da análise dinâmica (fase 1) fossem coletados pela ferramenta. A análise dinâmica para todas as releases foi executada no mesmo computador sob as mesmas condições e com serviços não essenciais desabilitados. A configuração do computador utilizado foi um AMD Phenom II com 8 GB de memória RAM, executando o sistema operacional Linux Ubuntu 16.04 LTS e o Java na versão 8. A suite de testes automatizados de cada release foi executada dez vezes nas mesmas condições a fim de obter médias de desempenho, em termos de tempo de execução, mais precisas.

Na sequência, as dez releases de cada sistema foram agrupadas em 9 pares de evoluções para executar as fases 2 e 3 do *PerfMiner*, além da extensão proposta pelo *PerfMiner Visualizer*. A Tabela 2 mostra como ficaram organizados esses pares para cada sistema.

Na fase 2, o resultado da análise dinâmica para cada par de releases mostrado anteriormente foi recuperado e comparado, entre os pares, com a finalidade de determinar métodos e construtores com degradação ou otimização de desempenho. Já na fase 3, os elementos com desvios de desempenho foram minerados nos seus sistemas de controle de

Tabela 2: Organização dos pares de releases para o Jetty e VRaptor.

Jetty		VRaptor	
9.3.10	9.3.11	4.0.0.Final	4.1.0.Final
9.3.11	9.3.12	4.1.0.Final	4.1.1
9.3.12	9.3.13	4.1.1	4.1.2
9.3.13	9.3.14	4.1.2	4.1.3
9.3.14	9.3.15	4.1.3	4.1.4
9.3.15	9.3.16	4.1.4	4.2.0.RC1
9.3.16	9.4.0	4.2.0.RC1	4.2.0.RC2
9.4.0	9.4.1	4.2.0.RC2	4.2.0.RC3
9.4.1	9.4.2	4.2.0.RC3	4.2.0.RC4

versão e sistemas de gerenciamento de tarefas. Ao final das 3 fases do *PerfMiner*, são gerados os artefatos de saída requeridos para o *PerfMiner Visualizer*. Vale salientar neste ponto que, embora o *PerfMiner* realize a mineração e obtenha o conjunto de tarefas que estão relacionadas com os *commits* que possivelmente foram os responsáveis por inserir determinado desvio de desempenho, a integração com essas ferramentas de gerenciamento de tarefas está, até este momento, fora do escopo do *PerfMiner Visualizer*.

Com os artefatos de saída, a fase 4, pertencente à extensão proposta, foi iniciada. Para cada sistema e um par de releases foi executada a análise descrita na subseção 3.2.1, cuja finalidade é, a partir desses artefatos, realizar um processamento com o intuito de gerar os dados que dão suporte às visualizações. Após a execução dessa fase, iniciou-se o processo de verificação das visualizações geradas para, posteriormente, publicar o resultado das análises em um ambiente na nuvem, o Heroku², cujo endereço é <http://apvis.herokuapp.com>.

4.1.3.3 Passo 3 - Elaboração e Aplicação dos Questionários

A elaboração do questionário se deu na sequência, após a publicação dos resultados das análises. O questionário foi dividido em 5 seções: (i) uma com questões demográficas, (ii) outra com questões sobre o atributo de qualidade de desempenho, (iii) uma terceira com questões sobre a visualização do grafo de chamadas, (iv) outra sobre a visualização da sumarização de cenários e, por fim, (v) uma seção com questões gerais que concluem o questionário. Todas as questões do questionário foram opcionais.

A primeira seção foi elaborada com o intuito de coletar informações sobre a experiência dos participantes no desenvolvimento de software na linguagem de programação Java

²<http://www.heroku.com>

e nos seus respectivos projetos. A segunda seção continha questões sobre a experiência de uso de ferramentas de *profiling* e de gerenciamento de desempenho de aplicações (APM), e sobre estratégias que os participantes utilizam para analisar o desempenho das funcionalidades dos respectivos sistemas. Na terceira seção, foram perguntadas questões sobre a visualização do grafo de chamadas. Foram apresentadas situações reais para que eles se baseassem ao responderem as questões. Além disso, foi perguntado sobre os aspectos que eles mais e menos gostaram com relação a visualização. A mesma estratégia foi utilizada para a quarta seção do questionário, mas relacionada à visualização da sumarização de cenários. Na quinta e última seção, os participantes responderam sobre os benefícios de se utilizar a ferramenta, bem como se eles a usariam nos processos de desenvolvimento de software dos seus respectivos sistemas. Os participantes tiveram acesso a todos os resultados gerados pela ferramenta no final do questionário.

Os participantes foram agrupados de acordo com o sistema em que contribui. Sendo assim, dois grupos foram formados: 66 participantes para grupo de contribuidores Jetty e 48 para o grupo de contribuidores do VRaptor. Depois, cada grupo foi dividido em dois, de modo que, para cada sistema, foram aplicados dois tipos de questionários. A Tabela 3 mostra a divisão dos grupos de participantes:

Tabela 3: Distribuição dos grupos de participantes.

Grupo	Sistema	Quantidade de Participantes	Tipo de Questionário	Idioma
A	Jetty	33	Tipo 1	Inglês
B	Jetty	33	Tipo 2	Inglês
C	VRaptor	24	Tipo 1	Português
D	VRaptor	24	Tipo 2	Português

Os tipos dos questionários têm as seguintes características:

- *Tipo 1*: neste tipo o participante respondeu às questões sobre a visualização do grafo de chamadas através da visualização exibida pela ferramenta. Por outro lado, as questões sobre a visualização da sumarização de cenários foram respondidas baseadas em resultados dispostos em uma tabela, ao invés da visualização fornecida pela ferramenta;
- *Tipo 2*: é o inverso do anterior, o participante respondeu às questões sobre a visualização do grafo de chamadas se baseando em resultados exibidos em uma tabela e respondeu às questões sobre a visualização da sumarização de cenários através da visualização mostrada pela ferramenta.

A divisão do questionário nesses dois tipos permite que as respostas sejam comparadas entre os que responderam baseados nas visualizações da ferramenta e os que responderam baseados em dados tabulares. Ambos os tipos tiveram questões iguais, no entanto, utilizando referências específicas aos cenários dos projetos analisados, quando aplicado nas questões. Através dessa divisão, pode ser comparado, por exemplo, o percentual de acerto às questões com e sem as visualizações. Além disso, as perguntas sobre as visualizações permitem coletar informações sobre a identificação correta de elementos, a facilidade de se identificar esses elementos e as impressões dos participantes sobre elas. Os dois tipos do questionário podem ser consultados no apêndice A e os dados tabulares usados pelos participantes pode ser encontrado no link <https://goo.gl/DJwtLT>.

Com o questionário elaborado, foram coletados os dados de e-mail e login dos contribuidores de cada projeto através da API do Github. No entanto, não foi possível coletar esses dados para todos os contribuidores pelo fato de existir uma opção nessa ferramenta em que os usuários podem desabilitar a publicação do seu endereço de e-mail nessa API. Para esses casos foram feitas visitas manuais aos perfis de cada contribuidor. Ao final desse processo, foram submetidos 66 questionários para os contribuidores do Jetty e 48 para os contribuidores do VRaptor, um total de 114.

4.1.4 Caracterização dos Participantes

Do total de 114 contribuidores que receberam o questionário, 16 deles responderam com algum feedback: 9 do Jetty e 7 do VRaptor, uma taxa de resposta de 14,04%. Das respostas, 3 foram consideradas inválidas, pois: (i) em uma delas, o usuário abriu a ferramenta em um celular, comprometendo as visualizações e, conseqüentemente, as suas respostas; (ii) outro usuário, ao invés de abrir a ferramenta para responder às questões com base nas visualizações, respondeu com base em apenas uma figura explicativa elaborada para elucidar informações sobre as visualizações; e (iii) o terceiro usuário respondeu apenas as questões demográficas, deixando todas as outras em branco. Portanto, 13 respostas foram consideradas válidas para a análise dos resultados.

A maioria dos participantes que responderam são desenvolvedores (31%), arquitetos (15%) e engenheiros (23%) de software, possuem mais de 5 anos de experiência na linguagem de programação Java (Figura 37a) e possuem entre 1 e 10 contribuições no seu projeto (Figura 37b). Apesar de alguns participantes não terem contribuído para os projetos nos últimos 12 meses, o fato de serem experientes no desenvolvimento de software em Java pode dar credibilidade às suas respostas.

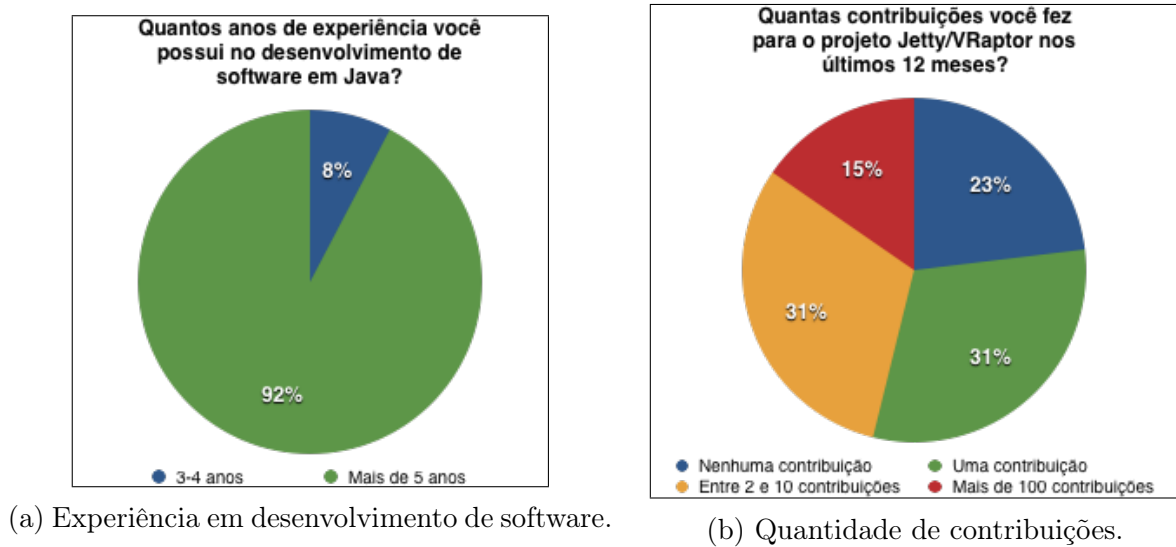


Figura 37: Dados demográficos dos participantes.

Sobre a experiência dos participantes com o uso de ferramentas de análise de desempenho, como ferramentas de *profiling* ou ferramentas APM, 54% declarou usar com frequência essas ferramentas e 15% declarou já ter usado pelo menos uma vez. As ferramentas mais citadas pelos participantes foram: JProfiler, New Relic e JVisualVM.

Em resposta à questão “*Suponha que uma funcionalidade do Jetty/VRaptor foi evoluída (por você ou outro membro da equipe). O que você faz para garantir que o tempo de execução/resposta de tal funcionalidade é aceitável em comparação com outras releases?*”, a maioria dos participantes respondeu que fazem a medição de desempenho, no entanto, foram várias as maneiras descritas por eles: o participante P4GA (participante 4 do grupo A) realiza a “*medição e comparação de tempos de execução dos testes*”, já o participante P11GA menciona que faz o “*teste de integração contínua de um teste de gerador de carga*”, o participante P13GC realiza “*análise da complexidade do algoritmo envolvido na evolução e caso não seja possível garantir dessa forma, faço com alguns testes automatizados, seja unitário ou usando um software como o JMeter*” e o participante P6GD faz “*comparação via NewRelic (response time - response time by endpoint - etc)*”.

4.2 Resultados

Esta seção discute os resultados obtidos após a execução da ferramenta para os sistemas mencionados e a aplicação do questionário online com os desenvolvedores dessas aplicações. A subseção 4.2.1 comenta o comportamento das visualizações para os dados obtidos dos sistemas e a subseção 4.2.2 apresenta os resultados obtidos após a aplicação

do questionário.

4.2.1 Análise do Comportamento das Visualizações

Após a execução dos passos 1 e 2 da subseção de Procedimentos (4.1.3) foram verificadas as visualizações geradas e constatou-se que, de maneira geral, a representação visual dos dados analisados foi feita de maneira correta e satisfatória. As subseções seguintes apresentam os resultados da visualização de Sumarização de Cenários, mostrando exemplos em que a visualização foi gerada conforme esperado e outros que expõem situações onde a identificação dos cenários foi prejudicada, e da visualização do Grafo de Chamadas, expondo o algoritmo de supressão de nós exibidos além de comentar casos especiais ocorridos.

4.2.1.1 Sumarização de Cenários

Ao final da análise das releases selecionadas, foram encontrados um total de 244 cenários com desvios de desempenho, sejam degradações ou otimizações. Eles estão mostrados nas tabelas 4 para o Jetty, e 5 para o VRaptor.

Tabela 4: Resumo para o Jetty.

Versão Inicial	Versão Final	Cenários Degradados	Cenários Otimizados	Total de Cenários
9.3.10	9.3.11	5 (3,18%)	1 (0,63%)	157
9.3.11	9.3.12	0 (0%)	36 (21,55%)	167
9.3.12	9.3.13	0 (0%)	7 (4,51%)	155
9.3.13	9.3.14	0 (0%)	0 (0%)	157
9.3.14	9.3.15	3 (1,84%)	2 (1,22%)	163
9.3.15	9.3.16	2 (1,21%)	1 (0,60%)	165
9.3.16	9.4.0	28 (14,97%)	6 (3,20%)	187
9.4.0	9.4.1	1 (0,53%)	0 (0%)	186
9.4.1	9.4.2	4 (2,15%)	0 (0%)	186
Total		43	53	1523

Nas tabelas, três pares de versões não possuíam nenhum cenário identificado como degradado ou otimizado: 9.3.13/9.3.14 para o Jetty, 4.1.0.Final/4.1.1 e 4.1.3/4.1.4 para o VRaptor. Isso significa que esses pares de versões tiveram os seus cenários exercitados e seus tempos de execução medidos, no entanto, o *PerfMiner* não encontrou diferenças estatisticamente relevantes nos seus tempos de execução entre a versão anterior e a posterior.

Tabela 5: Resumo para o VRaptor.

Versão Inicial	Versão Final	Cenários Degradados	Cenários Otimizados	Total de Cenários
4.0.0.Final	4.1.0.Final	36 (4,86%)	41 (5,54%)	740
4.1.0.Final	4.1.1	0 (0%)	0 (0%)	745
4.1.1	4.1.2	20 (2,66%)	10 (1,33%)	750
4.1.2	4.1.3	9 (1,19%)	2 (0,26%)	752
4.1.3	4.1.4	0 (0%)	0 (0%)	739
4.1.4	4.2.0.RC1	0 (0%)	1 (0,12%)	774
4.2.0.RC1	4.2.0.RC2	4 (0,51%)	1 (0,12%)	776
4.2.0.RC2	4.2.0.RC3	6 (0,77%)	0 (0%)	771
4.2.0.RC3	4.2.0.RC4	15 (1,94%)	3 (0,38%)	773
Total		90	58	6820

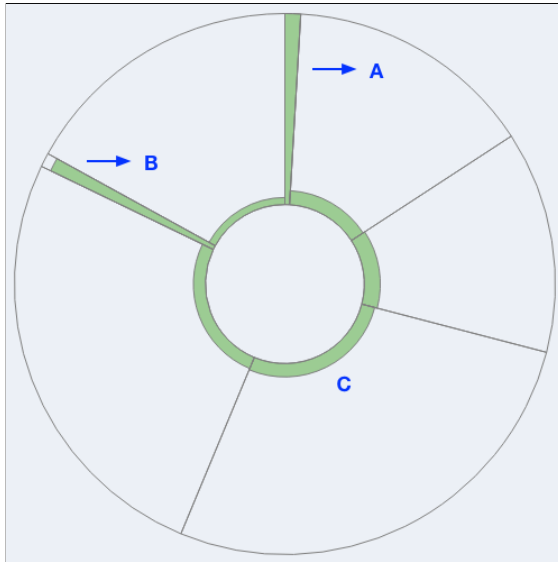
Visualização sem Prejuízos

A Figura 38 apresenta exemplos de visualizações de sumarização de cenários geradas sem prejuízos. Na Figura 38a são mostrados 7 cenários com otimizações de desempenho para o **Jetty**, versão 9.3.12 para 9.3.13. O cenário indicado pela letra A foi o de maior tempo de execução, conforme denuncia a altura da sua fatia preenchida. Percebe-se, através do gráfico, que os cenários A e B tiveram tempos de execução bem maiores do que o restante. Já o cenário C foi o que teve a maior otimização (47,46%), identificada através da largura da fatia. A Figura 38b exibe 6 cenários degradados para o **VRaptor**, versão 4.2.0.RC2 para 4.2.0.RC3. O cenário A foi o que possuiu o maior tempo de execução, sendo o seu tempo de execução muito maior do que o dos outros cenários. O cenário B foi o que teve o maior desvio de desempenho, 111,93%.

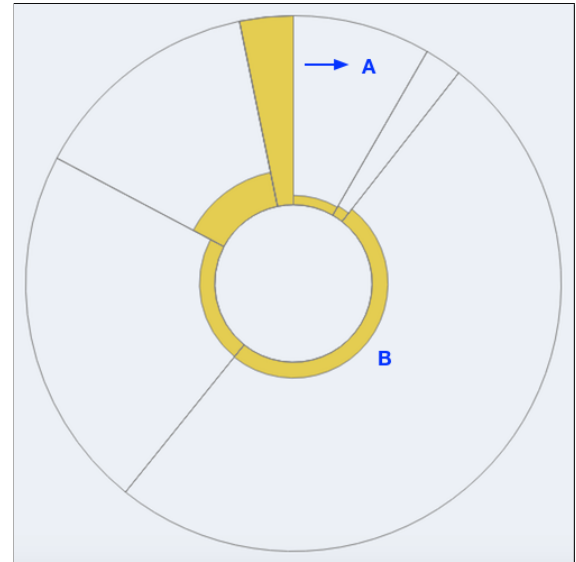
Os dois exemplos comentados anteriormente mostram cenários com apenas um tipo de desvio de desempenho entre as versões. Na Figura 38c são exibidos cenários de ambos os tipos, 3 degradações e 2 otimizações, para o **Jetty**, versões 9.3.14 para 9.3.15. O cenário indicado pela letra A foi o de maior tempo de execução, já o indicado pela letra B foi o que teve a maior porcentagem de desvio de desempenho, 92,99% de otimização. Já na Figura 38d são mostrados 5 cenários, 4 degradações e 1 otimização, para o **VRaptor**, versões 4.2.0.RC1 para 4.2.0.RC2. O único cenário otimizado, identificado pela letra A, foi o que possuiu o maior tempo de execução, já o maior desvio de desempenho foi identificado no cenário B, 142,87% de degradação.

É importante salientar que os cenários apresentados na Figura 38 não são todos os cenários existentes para as versões analisadas, mas sim apenas os cenários identificados com desvios de desempenho após a execução da ferramenta. De acordo com a Tabela 4,

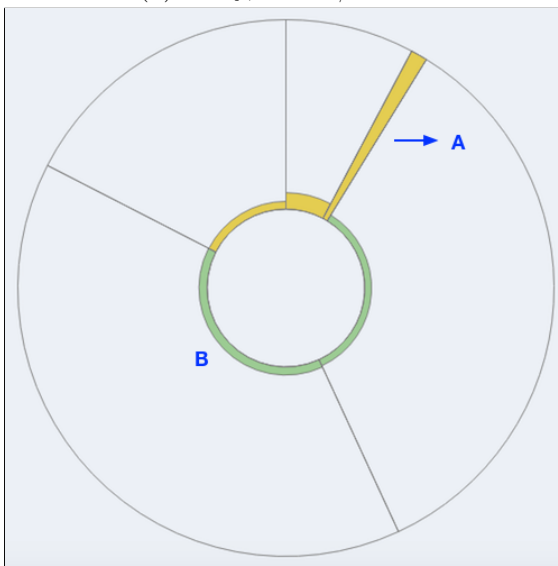
existem 155 cenários entre as versões 9.3.12 e 9.3.13, sendo 7 (4,51% do total) deles identificados com desvios de desempenho. Já entre as versões 9.3.14 e 9.3.15, são 163 cenários ao todo, mas apenas 5 (3,06%) deles com desvios de desempenho.



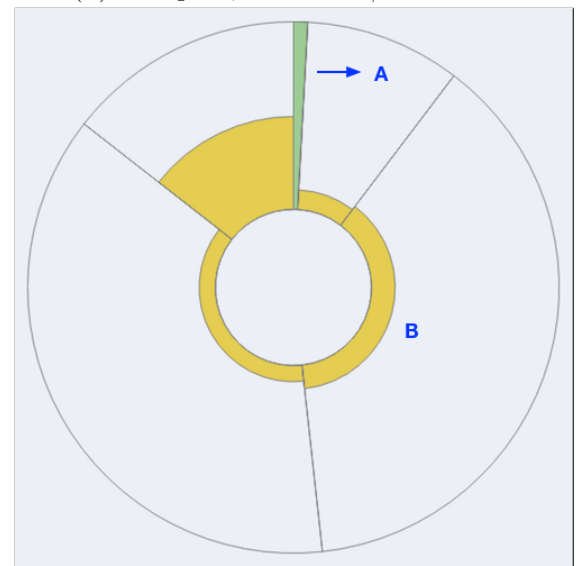
(a) Jetty, 9.3.12/9.3.13.



(b) VRaptor, 4.2.0.RC2/4.2.0.RC3.



(c) Jetty, 9.3.14/9.3.15.



(d) VRaptor, 4.2.0.RC1/4.2.0.RC2.

Figura 38: Exemplos da visualização de Sumarização de Cenários.

Visualização com Prejuízos

Houve casos em que a identificação dos cenários na visualização foi prejudicada, de maneira geral: quando há muitos cenários com desvios de desempenho para o par de versões analisados e cenários com baixas porcentagens de desvios de desempenho. Quando isso acontece, as fatias ficam pequenas, dificultando a visualização, a localização pelo mouse e o clique para a visualização do grafo de chamadas. Em alguns casos, parte dos

cenários apresentados podem ficar até mesmo ilegíveis. A Figura 39 apresenta esses casos.

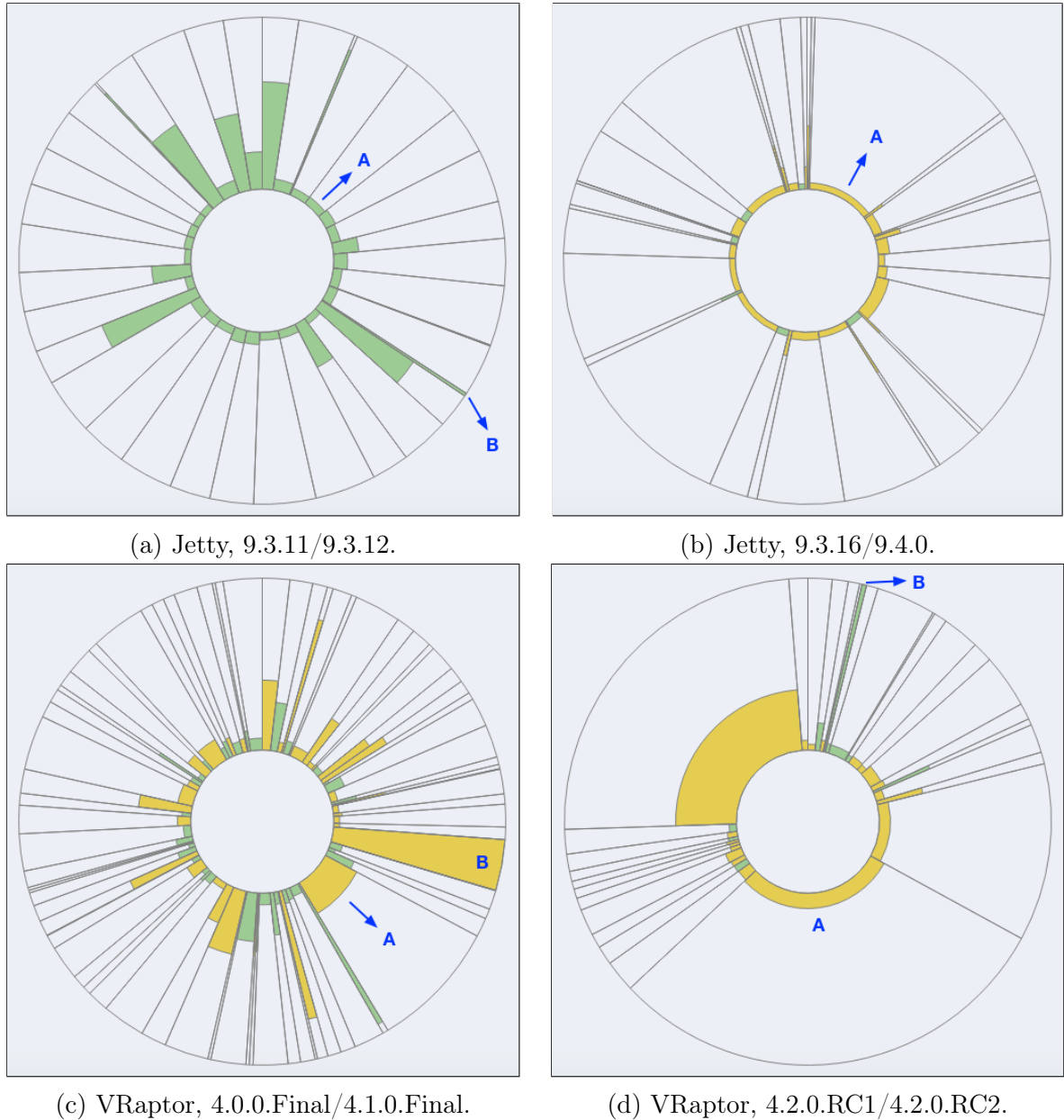


Figura 39: Exemplos da visualização de Sumarização de Cenários com excesso de cenários.

A Figura 39a mostra 36 cenários de otimização para o sistema Jetty, versões 9.3.11 para 9.3.12. Nesse caso, os cenários estão com larguras distribuídas uniformemente, exceto 4 deles que tiveram baixa porcentagem de desvio e, portanto, suas fatias estão finas com relação às demais. Por causa dessa distribuição, pode ser complexo localizar o cenário A, que foi o que possuiu o maior porcentagem de desvio de desempenho (87,64%). Nesse caso, como a distinção visual é difícil, é necessário passar o ponteiro do mouse sobre as fatias com largura semelhante para, somente após diferenciar os valores das porcentagens, constatar que o cenário A é o com maior desvio. O cenário B possuiu o maior tempo de

execução. Sua identificação é menos difícil do que o cenário A pelo fato de existirem poucas fatias do gráfico com a mesma largura. Entretanto, o baixo desvio torna a fatia que representa esse cenário bastante fina, prejudicando a sua identificação visual.

A Figura 39b apresenta 34 cenários para o sistema **Jetty**, versões 9.3.16 para 9.4.0. Nesse caso, embora a quantidade de cenários seja ligeiramente menor do que no caso anterior, a distribuição irregular ocasionada por diferentes porcentagens de desvios faz com que o cenário indicado pela letra A seja, através da distinção visual de sua largura perante os demais cenários, identificado como o cenário com maior desvio de desempenho (136,88% de degradação) dentre os apresentados. Por outro lado, essa distribuição irregular fez com que o cenário com menor desvio de desempenho ficasse imperceptível na visualização: foram apenas 0,07% de degradação. Coincidentemente, nesse caso, esse cenário é o que possui o maior tempo de execução dentre os exibidos e, através dessa visualização, não foi possível identificá-lo.

De todas as versões analisadas para os dois sistemas, as versões 4.0.0.Final para 4.1.0.Final, do sistema **VRaptor**, foi a que possuiu a maior quantidade de cenários identificados com desvios de desempenho: 77. A Figura 39c mostra a sumarização de cenários para esse caso. A distribuição irregular faz com que o cenário indicado com a letra A seja identificado como o que teve o maior desvio de desempenho (323,44% de degradação), também, como no caso anterior, através da diferenciação visual das larguras das fatias. O cenário da letra B foi o que possuiu o maior tempo de execução e, como também teve uma porcentagem de degradação alta (127,27%) comparado com a maioria dos cenários, sua identificação pode ser feita por distinção visual. Embora a maioria dos cenários esteja legível por terem porcentagens de desvios de desempenho razoáveis, os que possuem baixa porcentagem de desvio podem ser difíceis de identificar.

A Figura 39d apresenta 30 cenários para o **VRaptor**, versões 4.2.0.RC1 para 4.2.0.RC2. Através de diferenciação visual das larguras das fatias, como no caso anterior, o cenário da letra A pode ser identificado como o cenário que possuiu o maior desvio de desempenho: uma degradação de 730,93%. De identificação mais difícil e não tão imediata como o cenário A, o cenário da letra B se apresenta como o de maior tempo de execução, porém com baixa porcentagem de desvio. Assim como no caso anterior, os cenários com baixa porcentagem de desvio de desempenho podem ser difíceis de identificar. No entanto, como nos casos da Figura 39a e 39c, tanto o cenário com maior desvio de desempenho quanto o cenário com maior tempo de execução podem ser identificados.

Essa dificuldade em localizar os cenários, nesses casos, não necessariamente está li-

gada somente a quantidade de cenários apresentados pela visualização, mas também a distribuição das porcentagens de desvio. Quando há uma vasta quantidade de cenários a serem exibidos (acima de 15) e cenários com baixas porcentagens de desvio com relação aos demais, a identificação destes pode ser prejudicada ou inviabilizada. No entanto, vale ressaltar que foi possível identificar o cenário com maior porcentagem de desvio para todas as releases analisadas pelo estudo e em apenas um caso o cenário com maior tempo de execução não pôde ser localizado.

Para resolver ou minimizar essa situação, podem ser feitas adaptações na implementação dessa visualização: (i) os cenários poderiam ser agrupados por módulos ou funcionalidades do sistema, de acordo com critérios definidos pelos usuários. Dessa forma, várias visualizações de sumarização de cenários seriam geradas, uma para cada módulo, e a ferramenta as apresentaria, uma a uma, com mecanismos de paginação entre os módulos; (ii) outra forma seria fazer um ranqueamento das maiores porcentagens de desvios de desempenho e agrupá-los por esse critério. Assim, utilizando um mecanismo de paginação para apresentá-las, na primeira página estariam os n (por exemplo, 10) cenários com maior porcentagem de desvio de desempenho, e assim por diante nas páginas seguintes.

QP1. Os cenários identificados com desvios de desempenho são apresentados claramente nas visualizações implementadas pela ferramenta? Para 10 de 15 visualizações de sumarização de cenários geradas houve eficácia na apresentação dos casos analisados, de modo que o usuário pode distinguir de maneira clara, sem prejuízos, entre os cenários com maior/menor desvio de desempenho, além de maior/menor tempo de execução. Em 5 das 15 visualizações, quando há uma grande quantidade de cenários a serem exibidos (acima de 15) e cenários com baixas porcentagens de desvio de desempenho com relação aos demais, a identificação dos cenários foi prejudicada (em 4 das 5) ou inviabilizada (em 1 das 5).

4.2.1.2 Grafo de Chamadas

Cada um dos 244 cenários identificados com desvios de desempenho pela ferramenta possuem nós a serem dispostos na visualização do grafo de chamadas. Nesta subseção serão destacados o resultado do algoritmo de supressão dos nós exibidos, bem como serão comentados 8 casos especiais de grafos.

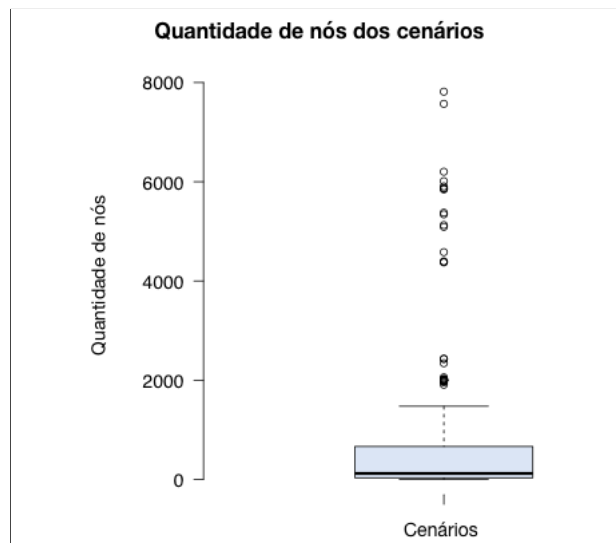
Algoritmo de Supressão de Nós Exibidos

Com a execução do algoritmo de supressão de nós exibidos (explicado na seção 3.4.3.6) para cada grafo de chamada gerado pela ferramenta, a redução da quantidade de nós a serem mostrados foi importante, consequentemente, diminuindo a complexidade de se encontrar informações. O gráfico exibido na Figura 40b indica que em 75% dos cenários analisados a redução de nós se deu entre 73,77% e 99,83% e a mediana de redução de nós considerando todos os cenários é de 90,26%. Os *outliers* desse gráfico indicam os casos em que a porcentagem de redução foi considerada baixa. Isso pode acontecer, por exemplo, em cenários com poucos nós no total, onde a redução de nós a serem exibidos é baixa ou até mesmo inexistente.

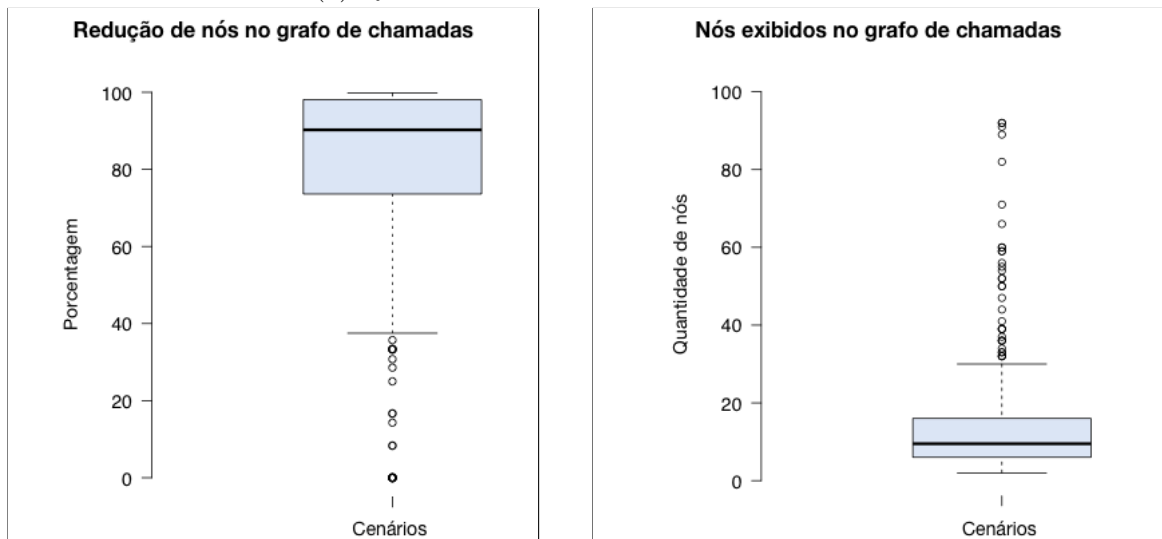
O gráfico da Figura 40c reforça o resultado do algoritmo, mostrando a distribuição da quantidade de nós exibidos no grafo de chamadas para os cenários analisados. Nesse gráfico, em 75% dos cenários a quantidade de nós exibidos é entre 2 e 16, e a mediana de exibição dos nós considerando todos os cenários é de 9,5, indicando a baixa complexidade do grafo, na maioria dos casos. Entretanto, embora na maioria dos cenários a quantidade de nós exibidos seja baixa, houve casos em que essa quantidade pode ser considerada alta, conforme indicado pelos *outliers* do gráfico. Por exemplo, o caso com maior quantidade de nós no grafo foi o cenário `Entry point for DispatcherForwardTest.testQueryRetainedByForwardWithoutQuery`, do Jetty, versões 9.3.16 para 9.4.0. Nesse cenário são exibidos 92 nós de um total de 5.376. Nesse caso, apesar de a quantidade de nós seja alta e seja considerado um *outlier* no gráfico da Figura 40c, a porcentagem de redução foi de 98,29%, indicando a eficiência do algoritmo.

O gráfico da Figura 40a exibe a distribuição da quantidade total de nós dos cenários analisados, para ambos os sistemas. Em 75% dos cenários a quantidade de nós é entre 2 e 670, e a mediana é de 119,5 nós. Um dos *outliers* mostrados no gráfico é o cenário `Entry point for DefaultServletTest.testListingContextBreakout` do sistema Jetty, que teve maior quantidade de nós: 7816. Após aplicação desse algoritmo, a quantidade de nós a serem exibidos para esse cenário caiu para 56, ou seja, uma redução de 99,28%. Esse caso e a distribuição apresentada no gráfico evidencia que, sem a eficácia do algoritmo de supressão de nós, a complexidade da visualização do grafo de chamadas seria alta ou até mesmo inviável de ser produzida e interpretada.

QP2. O algoritmo de redução de nós aplicado pela ferramenta é capaz de reduzir significativamente o número de nós no grafo de chamadas para



(a) Quantidade total de nós dos cenários.



(b) Porcentagem de nós reduzidos.

(c) Quantidade de nós exibidos.

Figura 40: Gráficos *boxplot* sobre o algoritmo de redução de nós.

sistemas reais? O algoritmo de supressão de nós exibidos conseguiu reduzir, para 75% dos cenários analisados, entre 73,77% e 99,83% a quantidade de nós a serem mostrados no grafo. Essa quantidade é de 2 a 16 nós exibidos, para 75% dos cenários analisados.

Casos Especiais

Dentre os cenários analisados para os sistemas Jetty e VRaptor, alguns deles podem ser destacados por serem casos especiais de situações envolvendo a quantidade de nós degradados e otimizados, a porcentagem de redução dos nós exibidos e a porcentagem de desvio de desempenho de cenários. Os cenários que exemplificam esses casos são os

seguintes:

- *Alta quantidade de nós degradados (10)*: Entry point for `AsyncContextListenersTest.testAsyncDispatchAsyncCompletePreservesListener` (C1)
- *Alta quantidade de nós otimizados (8)*: Entry point for `ServletHolderTest.testUnloadableClassName` (C2)
- *Menor porcentagem de redução de nós (0%)*: Entry point for `SafeResourceBundleTest.shouldReturnKeyBetweenQuestionMarksWhenKeyDoesntExist` (C3)
- *Maior porcentagem de redução de nós (99,83%)*: Entry point for `ServletContextHandlerTest.testInitOrder` (C4)
- *Maior degradação (1.646%)*: Entry point for `DefaultEnvironmentTest.shouldUseContextInitParameterWhenSystemPropertiesIsntPresent` (C5)
- *Menor degradação (0,07%)*: Entry point for `PostServletTest.testGoodPost` (C6)
- *Maior otimização (97,47%)*: Entry point for `XStreamXmlDeserializerTest.shouldBeAbleToDeserializeADogWhenMethodHasMoreThanOneArgumentAndTheXmlIsTheLastOne` (C7)
- *Menor otimização (0,21%)*: Entry point for `XStreamXMLSerializationTest.shouldSerializeCollection` (C8)

A Tabela 6 a seguir resume alguns dados desses cenários. O cenário C1 foi o cenário com a maior quantidade de nós degradados dentre os analisados: 10. Houve também nós otimizados (4), adicionados (10) e removidos (3). Apesar de ter acontecido otimizações causadas pelos nós otimizados e removidos, o ganho de desempenho não foi suficiente para que o tempo total do cenário fosse otimizado. As degradações impostas pelos nós degradados e adicionados foram maiores, portanto, causando a degradação de 2,9%. Dentre as modificações que potencialmente causaram a degradação desse cenário, podem ser citados melhoramentos de tratamento de erros, permissão de personalização de métodos de classes, implementação de um conector para *socket* e correção de cálculos para a verificação de número mínimos de *threads* no servidor.

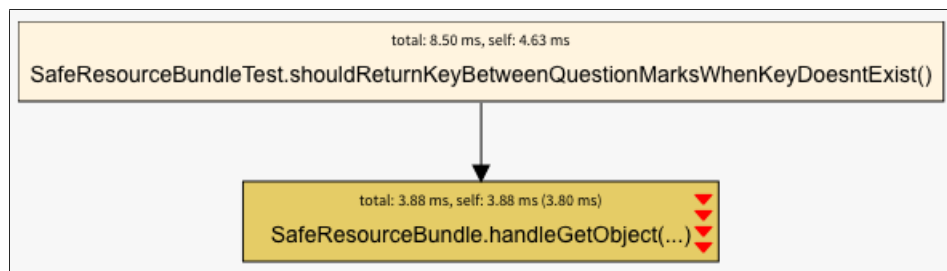
O cenário C2 foi o cenário com a maior quantidade de nós otimizados: 8. Assim como o cenário C1, também houve nós degradados (1), adicionados (6) e removidos (3). O resultado das otimizações e degradações impostas por esses nós foi um cenário degradado

Tabela 6: Características dos cenários exemplos de casos especiais.

Cenário	Sistema	Versão inicial/final	Tipo	% Desvio	Nós degradados/otimizados/adicionados/removidos	Nós exibidos/total	% Redução
C1	Jetty	9.3.16/9.4.0	Degradação	2,9	10/4/10/3	92/5.342	98,28
C2	Jetty	9.3.16/9.4.0	Degradação	4,07	1/8/6/3	71/718	90,11
C3	VRaptor	4.2.0.RC3/4.2.0.RC4	Degradação	124,49	1/0/0/0	2/2	0
C4	Jetty	9.3.11/9.3.12	Otimização	78,61	1/0/0/0	4/2.342	99,83
C5	VRaptor	4.2.0.RC3/4.2.0.RC4	Degradação	1.646	0/1/0/0	6/23	73,91
C6	Jetty	9.3.16/9.4.0	Degradação	0,07	0/0/5/0	28/98	71,43
C7	VRaptor	4.0.0.Final/4.1.0.Final	Otimização	97,47	2/0/0/0	8/42	80,95
C8	VRaptor	4.1.2/4.1.3	Otimização	0,21	1/0/0/0	8/171	95,32

em 4,07%, apesar da alta quantidade de nós otimizados. Esses nós pertencem a mesma classe do sistema: **Log**; e os métodos otimizados têm um baixo tempo de execução em nanosegundos. Diante disso, todas as 8 otimizações foram entre 0 e 25% do tempo, fazendo com que os nós permanecessem, então, com um baixo tempo de execução. Por outro lado, os nós adicionados, alguns deles executando em loop, e o removido adicionaram tempos de execução suficientes para que o cenário se degradasse.

A menor porcentagem de redução de nós exibidos foi no cenário **C3**. A quantidade de nós exibidos é exatamente o total de nós que o cenário possui: 2, sendo eles o nó raiz, que é o método da classe de teste automatizado que dá nome ao cenário, e o nó `SafeResourceBundle.handleGetObject(java.lang.String)`. O *commit* que modificou este método indica uma mudança no mecanismo de *log* que potencialmente é a causa da degradação de 124,49% do cenário. A Figura 41 a seguir ilustra o grafo de chamadas do cenário **C3**.

Figura 41: Grafo de Chamadas do cenário **C3**.

A maior porcentagem de redução de nós exibidos se deu no cenário **C4**: 99,83%. De um total de 2.342 nós, são exibidos apenas 4. Destes, um é o nó raiz que dá nome ao cenário, dois são nós agrupados, indicando a existência dos outros 2.338 nós, e o último é o nó que de fato houve desvio de desempenho, nesse caso, uma otimização. A Figura 42 adiante exemplifica esse grafo.

O cenário **C5** foi o que mais degradou o seu tempo de execução em relação à versão anterior: 1.646%. O grafo da Figura 43 mostra que o único nó com desvio de desempenho

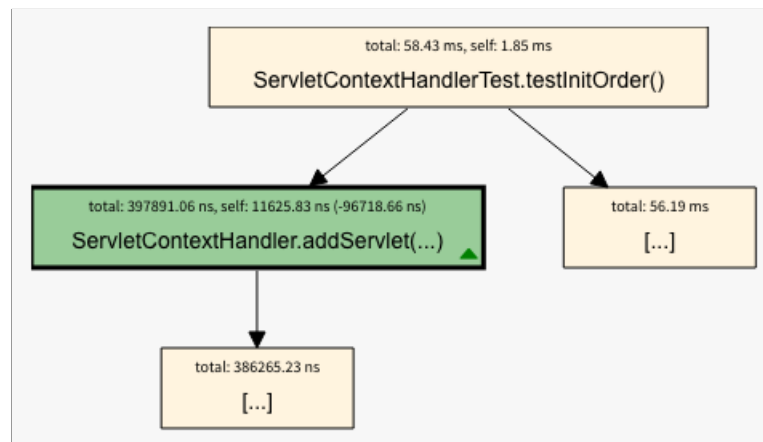


Figura 42: Grafo de Chamadas do cenário C4.

desse cenário apresentou uma degradação de mais de 75%. O método representado por esse nó, `DefaultEnvironment.setup()`, é o responsável por prover uma implementação padrão que carrega o arquivo de ambiente com base na propriedades do sistema.

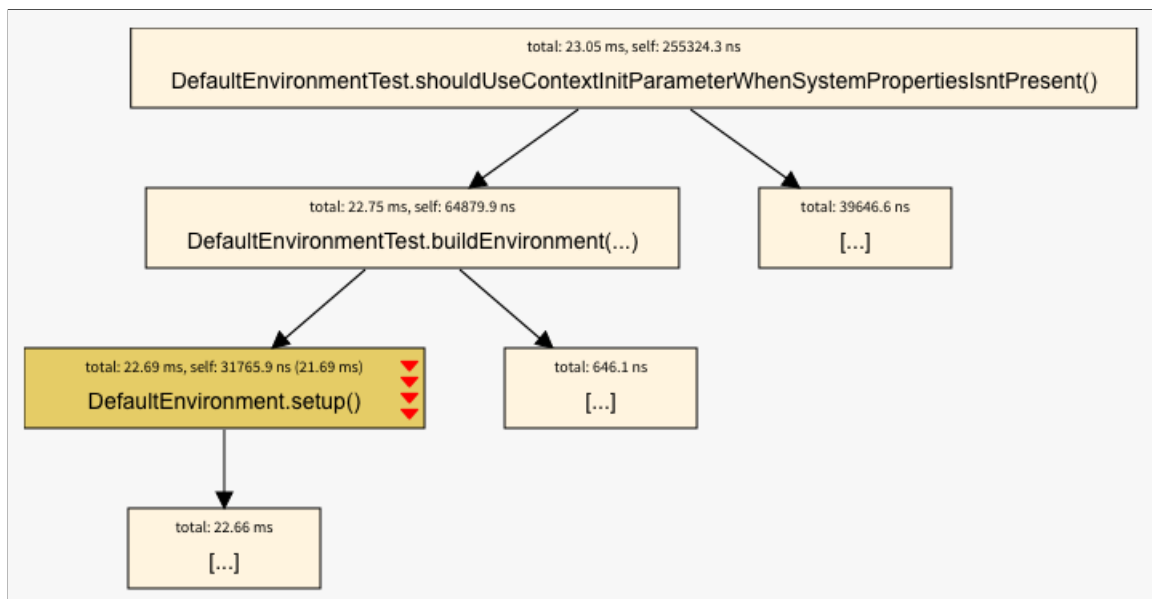


Figura 43: Grafo de Chamadas do cenário C5.

O cenário que menos degradou dentre os analisados foi o C6: 0,07%. Cinco nós adicionados foram os responsáveis por essa degradação. No entanto, os seus tempos são pequenos (em nanosegundos) em comparação ao tempo do cenário (em segundos) e, por isso, a degradação causada por eles foi baixa. Em suma, as modificações impostas por esses nós adicionados são simples verificações com um `if`.

Outro caso especial que merece ser destacado é o cenário C7, que possuiu a com maior otimização dentre os analisados: 97,47%. Dois nós com otimizações de mais de 75% foram os responsáveis por esse desvio, adicionando uma estratégia recursiva para a configuração

de um objeto da classe `com.thoughtworks.xstream.XStream`. A Figura 44 exemplifica esse grafo.

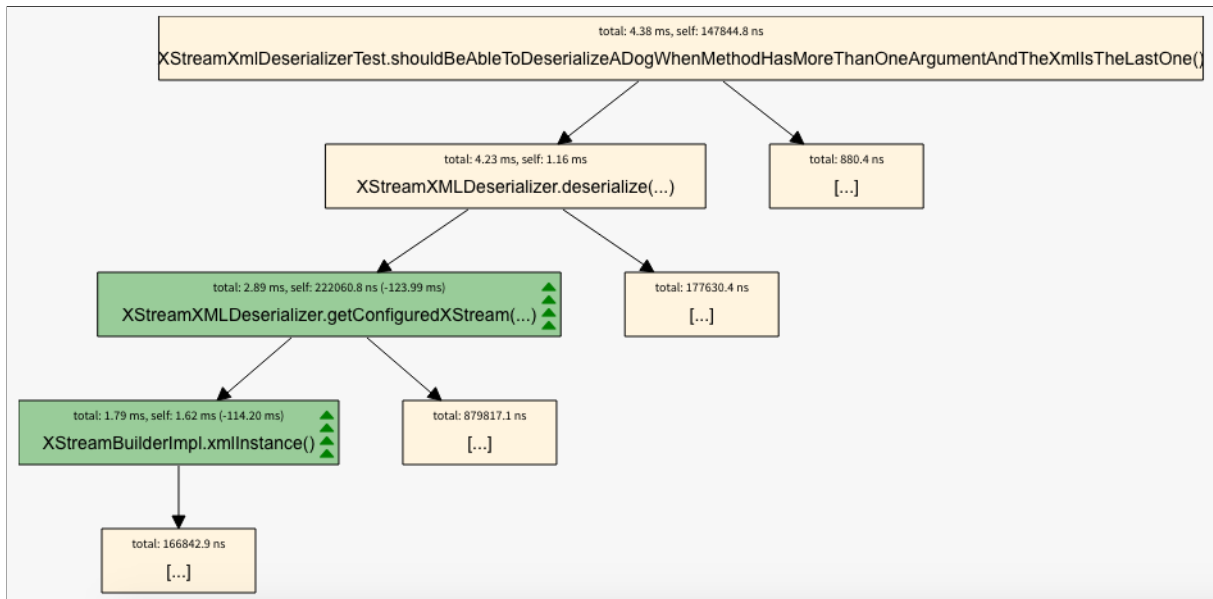


Figura 44: Grafo de Chamadas do cenário C7.

O último cenário destacado é o C8, que possuiu a menor otimização. Como exibe o grafo da Figura 45, um único nó foi responsável por esse desvio, `XStreamBuilderImpl.xmlInstance()`. Esse nó teve uma otimização no seu desempenho de 0 a 25% causada por uma mudança na chamada de um construtor. Isso fez com que o cenário otimizasse em apenas 0,21%.

A visualização do grafo de chamadas se mostrou eficaz na disposição dos nós dos cenários analisados. Através do grafo, é possível identificar os nós degradados, otimizados, adicionados e removidos, além de, através dos *commits*, tomar conhecimento sobre as mudanças que potencialmente causaram esses desvios de desempenho.

4.2.2 Questionário Online

O questionário online foi enviado para 114 contribuidores de ambos os sistemas. Desse, 16 responderam com algum feedback. Foram consideradas inválidas 3 dessas respostas, pelos motivos expostos na subseção 4.1.4, restando, portanto, 13 respostas válidas. As respostas comentadas e expostas nesta subseção consideram apenas as respostas válidas.

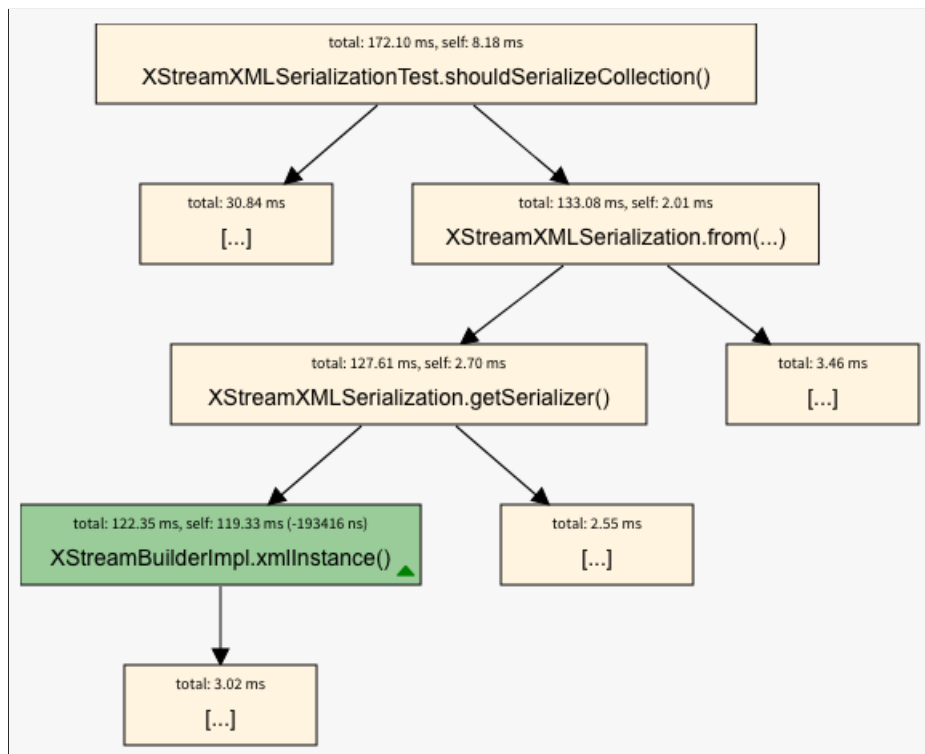


Figura 45: Grafo de Chamadas do cenário C8.

4.2.2.1 Grafo de Chamadas

A seção do questionário sobre o grafo de chamadas foi respondida pelos 4 grupos: dois deles responderam o questionário do tipo 1 e os outros dois grupos responderam ao questionário de tipo 2. Nessa seção, foi perguntado se os usuários conseguiam identificar os possíveis métodos responsáveis pelo desvio de desempenho de determinado cenário, além de solicitar que os métodos fossem listados em dois grupos: otimizados e degradados. Dos 9 participantes que responderam a essa pergunta baseados na visualização (questionário tipo 1), 6 (67%) deles responderam corretamente e um (11%) respondeu parcialmente correto. Nesse caso, a resposta foi considerada parcialmente correta pois o participante não mencionou um dos métodos degradados. Por outro lado, dos 4 participantes que responderam à questão baseados em dados tabulares (questionário tipo 2), apenas um (25%) respondeu corretamente.

Em seguida, os participantes responderam à questão “*Quão fácil foi responder à pergunta anterior?*”, para ambos os questionários. Considerando apenas os que responderam de maneira correta ou parcialmente correta, para o questionário em que os participantes responderam baseados na visualização, 4 (58%) dos 7 acharam muito fácil ou fácil responder à questão comentada no parágrafo anterior. Por outro lado, o único participante que acertou a mesma questão baseado em dados tabulares achou fácil encontrar as informações

solicitadas.

Na sequência, uma questão, de ambos dos questionários, solicitou aos participantes que identificassem o *hash* do *commit* responsável pelo principal desvio de desempenho do sistema. Dos 9 participantes que responderam a essa questão baseados na visualização do grafo de chamadas, 6 (67%) deles acertaram a resposta. Por outro lado, dos 4 que responderam baseados nos dados tabulares, 2 (50%) acertaram.

A próxima questão indagava “*Quão fácil foi responder à pergunta anterior?*”, para ambos os questionários. Considerando apenas os 6 participantes que responderam corretamente à questão anterior no questionário de tipo 1, 5 (84%) deles considerou muito fácil ou fácil. Por outro lado, para o questionário de tipo 2, dos 2 participantes que responderam à questão corretamente, apenas 1 (50%) deles achou fácil.

QP3. Desenvolvedores e arquitetos conseguem identificar os cenários com variações de desempenho e suas potenciais causas, em termos de métodos e *commits*, através do auxílio visual da ferramenta? Para o grafo de chamadas, a maioria (6 de 9 – 67%) dos participantes conseguiu identificar corretamente os métodos com variações de desempenho. A maioria (6 de 9 – 67%) dos participantes identificou corretamente os *hashes* dos *commits* responsáveis pelos desvios de desempenho dos métodos.

QP4. Há indícios de que as visualizações implementadas pela ferramenta são mais eficazes do que os dados tabulares para se encontrar informações sobre os cenários, métodos e *commits*? Diante do exposto para as questões anteriores, pode-se obter indícios de que a visualização do grafo de chamadas se mostrou mais eficaz do que os dados tabulares (isto é, desconsiderando a existência dessa visualização) para se encontrar informações sobre os métodos responsáveis pelo desvio de desempenho de determinado cenário e os *hashes* dos *commits* potencialmente responsáveis por esses desvios. Entretanto, vale salientar que a quantidade de respostas do grupo que respondeu ao questionário de tipo 1 foi maior do que a do grupo que respondeu ao questionário de tipo 2: 9 a 4.

Através da visualização do grafo de chamadas, os participantes acessaram um cenário real a partir da execução da ferramenta em seus respectivos sistemas e tomaram conhecimento, sobre o desvio de desempenho deparado. Em resposta à pergunta “*Este*

desvio parece plausível de acordo com o seu conhecimento do sistema?”, 7 (54%) dos 13 participantes com respostas válidas consideraram ser plausível o desvio de desempenho mostrado na visualização. Apenas 2 (15%) responderam que não era plausível, enquanto que o restante não informou (3 - 28%) ou não soube responder (1 - 8%).

Na seção dos questionários de tipo 1 e 2 que tratavam das visualizações havia uma pergunta para os participantes relatarem as suas impressões: *“Você poderia mencionar aspectos dessa visualização que você gostou? E quais outros aspectos você não gostou? Você tem alguma sugestão de melhoria ou comentário para essa visualização?”*. Para a visualização do Grafo de Chamadas, no geral, os participantes disseram gostar das características visuais implementadas, mas também fizeram críticas à visualização. O participante P5GA disse que *“o gráfico é bastante simples e fácil de entender. Não há muita informação, apenas o necessário. Eu gostei do recurso de destaque e a seta verde/vermelha indicando o nível de melhoria/degradação. Como sugestão, pode ser interessante adicionar informações sobre o contexto de execução (ex: argumentos JVM)”*, já o participante P22GA menciona que *“o popup Detalhes é difícil de copiar / colar. Eu preferiria que ele permanecesse aberto até eu clicar em outro lugar. Os links para o commit exato são agradáveis. Toda a visualização parece muito boa, mas gostaria que o grafo de chamadas tivesse um pouco mais de espaço em tela. Seria bom ter (temporariamente?) preenchendo a tela inteira. Por algum motivo, eu esperava poder clicar e arrastar o grafo de chamadas (como o Google Maps) em vez de usar a rolagem”*. O participante P13GC, no entanto, não percebeu que a funcionalidade de zoom foi implementada e relatou sentir falta: *“gostei dos gráficos e dos detalhes das informações. Achei o espaço reservado para o grafo de chamadas muito reduzido, o espaço vertical pra ele é pequeno. Outra coisa que senti falta foi uma opção de Zoom Out no grafo de chamadas”*. O gráfico da Figura 46 quantifica as opiniões dos participantes com relação ao que gostaram, não gostaram ou as sugestões feitas por eles para essa visualização.

Dos aspectos que os participantes gostaram, é importante destacar que houve 5 opiniões de que o grafo está fácil de entender. Outras menções foram feitas para funcionalidades como a de destaque dos nós com desvio, as setas indicativas de nós degradados ou otimizados e os *links* diretamente para os *commits* que possivelmente foram os causadores do desvio de desempenho do referido nó.

Sobre os aspectos que os participantes não gostaram, o participante P22GA comentou sobre os *tooltips*, conforme relatado no parágrafo anterior, e o participante P26GA mencionou que *“as diferenças de unidades de tempo (ms, ns) podem atrapalhar as pessoas”*.

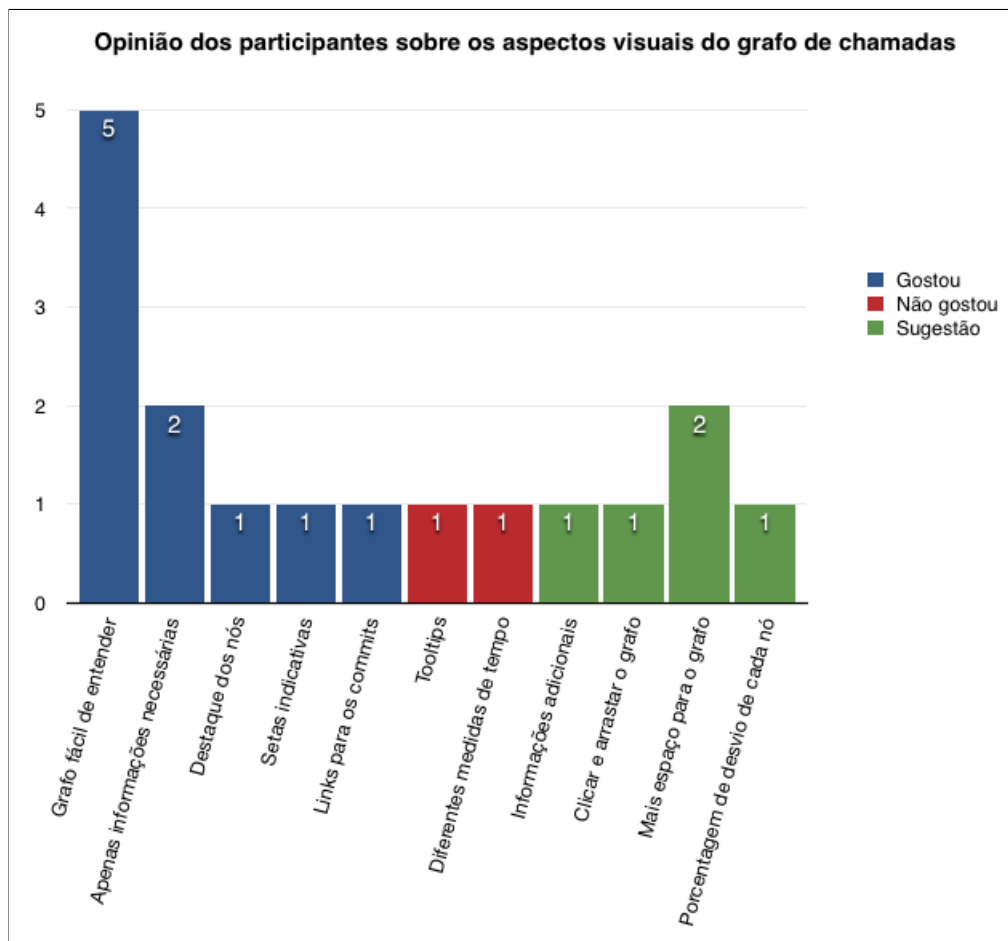


Figura 46: Opinião dos participantes sobre os aspectos da visualização do grafo de chamadas.

Essa diferença das unidades de tempo foi implementada para que a quantidade de números exibidos nos nós seja a menor possível. Dessa forma, sempre que possível o tempo é convertido para a unidade superior. Por exemplo: um tempo de execução de 1.200.000 nanosegundos será automaticamente convertido para 1,2 milissegundos, já um tempo de 1.500 milissegundos será convertido para 1,5 segundos.

Como sugestões, o participante P5GA acha interessante visualizar informações sobre o contexto de execução do teste automatizado, como, por exemplo, os argumentos da JVM. Já o participante P22GA, em sua resposta, comentou que esperava poder clicar e arrastar o grafo para navegação. Diante dessa frustração, a implementação dessa funcionalidade foi colocada como sugestão. Dois participantes sugeriram mais espaço para o grafo de chamadas. Se possível, mesmo que temporariamente, exibi-lo em tela cheia. Por fim, outro participante acha interessante que haja a porcentagem de desvio de desempenho indicada nos nós.

4.2.2.2 Sumarização de Cenários

A seção do questionário sobre a sumarização de cenários também foi respondida pelos 4 grupos: dois deles responderam o questionário do tipo 1 e os outros dois grupos responderam o questionário de tipo 2. Foi perguntado se os participantes conseguiam identificar qual dos cenários, dadas duas versões de seus respectivos sistemas, possuiu o maior desvio de desempenho dentre os exibidos. Além disso, foi solicitado que eles indicassem se o cenário tinha sido otimizado ou degradado. Dos 4 participantes responderam a essa questão baseados na visualização da sumarização de cenários, 2 (50%) a responderam corretamente e um (25%) respondeu parcialmente correto. Nesse caso, a resposta foi considerada parcialmente correta pois o participante apenas apontou o nome do cenário, mas não se ele degradou ou otimizou. Por outro lado, dos que responderam baseados em dados tabulares, 5 (56%) acertaram a resposta e uma (11%) foi considerada parcialmente correta. A resposta foi considerada parcialmente correta, nesse caso, pois o participante não apontou o nome do cenário e se este degradou ou otimizou, ele apenas indicou em qual linha da tabela estava o cenário.

Em seguida, os participantes responderam à questão *“Quão fácil foi responder à pergunta anterior?”* relacionada a questão do parágrafo anterior. Considerando apenas os participantes que responderam de maneira correta ou parcialmente correta a questão anterior, 2 (66%) dos 3 que responderam baseados na visualização apontaram ser muito fácil ou fácil respondê-la. Já os participantes que responderam baseados em dados tabulares, 5 (88%) dos 6 achou fácil ou muito fácil respondê-la.

Outra questão em que os participantes tinham que identificar um cenário a partir de suas características visuais foi para indicar o cenário com o maior tempo de execução dentre os exibidos, além de informar se ele foi otimizado ou degradado. Dois (50%) participantes que responderam à questão baseados na visualização acertaram, além de um (25%) que acertou parcialmente, de um total de 4. O participante que acertou parcialmente apenas informou o nome do cenário, mas não se este foi degradado ou otimizado. Por outro lado, dos que responderam baseados em dados tabulares, 5 (56%) responderam corretamente e um (11%) parcialmente correto. O participante que acertou parcialmente informou apenas a situação do cenário, porém não informou o seu nome.

Novamente, após a questão anterior, os participantes responderam à questão *“Quão fácil foi responder à pergunta anterior?”*. Considerando apenas os participantes que responderam de maneira correta ou parcialmente correta a questão anterior, 2 (66%) dos 3 que responderam baseados na visualização apontaram ser muito fácil ou fácil respondê-la.

Os 6 participantes que responderam baseados nos dados tabulares acharam muito fácil ou fácil respondê-la.

QP3. Desenvolvedores e arquitetos conseguem identificar os cenários com variações de desempenho e suas potenciais causas, em termos de métodos e *commits*, através do auxílio visual da ferramenta? Para a sumarização de cenários, não é possível afirmar que a visualização trouxe ganhos para a identificação dos cenários, uma vez que apenas metade (50%) dos participantes que a avaliaram conseguiram indicar corretamente os cenários com maior desvio de desempenho ou com maior tempo de execução.

QP4. Há indícios de que as visualizações implementadas pela ferramenta são mais eficazes do que os dados tabulares para se encontrar informações sobre os cenários, métodos e *commits*? Diante do exposto para as questões anteriores, não há indícios suficientes para concluir se a visualização da sumarização de cenários é mais eficaz na identificação dos cenários com maior desvio de desempenho ou com maior tempo de execução dentre os exibidos, uma vez que o resultado foi semelhante com o uso de dados tabulares.

De maneira análoga a seção dos questionários referentes a visualização do grafo de chamadas, os participantes responderam a questão *“Você poderia mencionar aspectos dessa visualização que você gostou? E quais outros aspectos você não gostou? Você tem alguma sugestão de melhoria ou comentário para essa visualização?”*. O sentimento foi dividido. O participante P1GD não se sentiu confortável com o gráfico de rosca utilizado para a visualização: *“gráficos em torta ou semelhantes normalmente querem representar 100% de algo. Usar a largura versus altura não foi intuitivo pra mim. Talvez separar mesmo em dois gráficos e dar a opção de ordenar ou por um ou por outro*. Por outro lado, o participante P6GD gostou e comentou que *“utilizar a largura, cor e altura da fatia me pareceu uma boa definição para identificação dos aspectos analisados”*.

4.2.2.3 Questões Finais

Após responderem sobre as visualizações, a seção final do questionário convidou os participantes a responderem a questões gerais sobre os benefícios, a utilidade da ferramenta além de comentários gerais. Na primeira das questões dessa seção, os participantes

responderam a questão “*Você vê benefícios de usar a ferramenta de visualização de desvios de desempenho apresentada? Se sim, quais?*”. Do total de participantes com respostas válidas, 9 (69%) deles disseram que veem benefícios no uso da ferramenta, conforme exibe a Figura 47a. Dentre os benefícios comentados pelos participantes, o participante P8GC disse que “*aparentemente pode trazer análises mais assertivas do atual desempenho do sistema*”, já o participante P13GC menciona que “*sim, é bem interessante para verificar se os novos releases contém alterações com prejuízos sérios à performance, podendo algumas vezes até detectar algum problema de lógica ou regra de negócio*”, ao passo que o participante P6GD informa que “*sim. É interessante saber de maneira precisa e quantitativa a quantidade e nível de melhorias e degradações entre versões*”.

Os 2 (15%) que talvez vejam benefícios na ferramenta também merecem destaque. O participante P19GD respondeu que “*Acho a ideia muito interessante, mas acredito que o projeto ainda precise evoluir bastante para se tornar usável*”. Já o participante P11GA disse que se sente “*duvidoso com relação a falsos positivos/negativos de testes automatizados que não foram projetados para serem comparados*”.

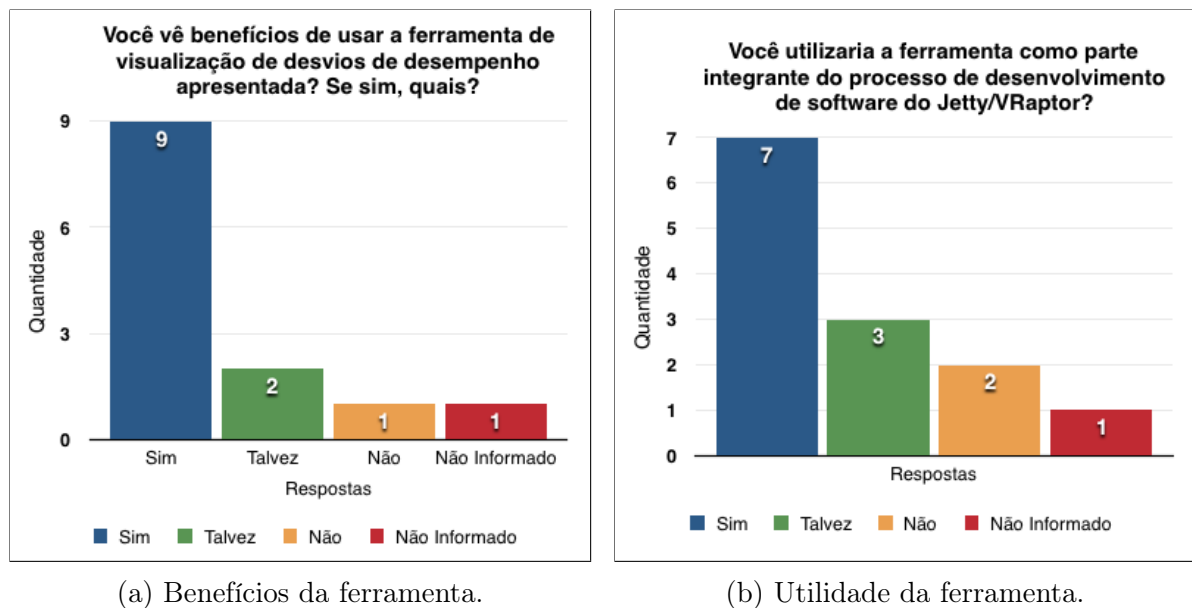


Figura 47: Questões sobre os benefícios e utilidade da ferramenta.

Com relação à questão “*Você utilizaria a ferramenta como parte integrante do processo de desenvolvimento de software do Jetty/VRaptor? Se sim, como você vislumbra que ela seria utilizada?*”, 7 (54%) dos participantes respondeu que usaria a ferramenta como parte integrante do processo de desenvolvimento. O participante P22GA disse que “*eu imagino usar esta ferramenta de duas maneiras: monitoramento regular para ver como as mudanças afetam o desempenho e investigação focada em uma questão específica*”, já o

participante P8GC reforça que *“talvez ela pudesse fazer parte do conjunto de análises antes da release”*, o participante P13GC afirma que *“Sim. Acho que poderia ser utilizada para avaliar releases antes de serem lançadas, para garantir que não há consequências graves à performance após as evoluções implementadas”* e o participante P18GD disse que *“sim. Integrada ao ciclo de entrega continua (acredito que usam travis.ci) para geração de reports (configurado no maven)”*. A Figura 47b sumariza as respostas à essa questão e expõe que a maioria dos usuários usariam a ferramenta em seus processos de desenvolvimento.

Vale destacar que 3 (23%) responderam talvez à questão do parágrafo anterior. Dentre as justificativas apresentadas por eles, o participante P4GA disse que *“não sei dizer. Não tenho certeza sobre outros testes mais complexos e de integração”*, já o participante P19GD comentou que *“não neste momento. Posteriormente poderia fazer parte de uma análise de status do projeto antes de um release”*. Essas respostas indicam que, caso tivessem mais conhecimento e experiência sobre a ferramenta, eles são potenciais usuários a integrarem-na aos seus processos de desenvolvimento.

Dos 13 participantes com respostas válidas, 9 (69%) veem benefícios em se utilizar a ferramenta e 7 (54%) usariam a ferramenta em seus processos de desenvolvimento de software.

A última questão dos questionários convidou os participantes a deixar comentários gerais sobre a ferramenta. A maioria deles deixou comentários positivos, no entanto, houve críticas. O participante P1GD comentou: *“achei algo bem profissional o que fizeram. Ainda está muito com cara de produto para hacker/coder, algo que já não sou mais, e fico perdido.”*. Já o participante P19GD comentou: *“gostei da ideia do projeto, mas acho que precisa de alguns ajustes, principalmente na exibição das informações. Por exemplo, na planilha tem colunas em ‘ns’, outras em ‘ms’, mas que todas poderiam ter sido normalizadas para facilitar as comparações.”*. O participante P11GA disse *“A análise é boa, mas o alvo ser testes unitários é pobre. Eu não me importo com o desempenho de 90% dos casos de teste... Eles estão muitas vezes testando erros e mau comportamento.”*. Outro participante (P26GA) disse que *“parece ótimo”*.

4.3 Considerações

O estudo analisou um total de 20 releases dos projetos Jetty [63] e VRaptor [64], sendo 10 releases para cada sistema, considerando 193 cenários distintos ao total, gerando

as visualizações de sumarização dos cenários e grafo de chamadas para todos os casos. Sobre as visualizações, foram coletados feedback de usuários desses sistemas através de um questionário online.

As principais contribuições são: (i) a identificação visual dos cenários que mais tiveram desvios de desempenho para os sistemas Jetty e VRaptor, a partir da análise de múltiplas releases; (ii) a identificação visual dos cenários com maior tempo de execução para os sistemas analisados; (iii) para os mesmos sistemas, a percepção visual do tipo de desvio dos cenários através da análise de múltiplas releases; (iv) a distinção visual dos nós que tiveram desvio de desempenho (degradação ou otimização), foram adicionados ou removidos para cada cenário dos sistemas analisados; e (v) para cada um desses nós, a indicação das potenciais causas dos desvios de desempenho, adição ou remoção ocorridos.

A expectativa, através dos resultados desse estudo, é ajudar os desenvolvedores e arquitetos a analisar o desempenho dos sistemas, facilitando a correção de gargalos de desempenho, através da identificação das suas causas, inseridos durante o desenvolvimento de determinada release. Dessa forma, integrada ao processo de desenvolvimento de software, a ferramenta pode ser usada de maneira preventiva, evitando a liberação de releases com problemas de desempenho desconhecidos. Embora nem toda degradação de desempenho seja passível de correção (por exemplo, a inserção de verificações de segurança de maneira planejada), a ferramenta provê a possibilidade de acompanhá-las.

4.3.1 Ameaças à Validade

Para estruturar as ameaças à validade do estudo, foi utilizada a apresentada por Wohlin[65].

4.3.1.1 Validades de Construção

Projeto do Questionário

Os dois tipos de questionários submetidos aos participantes continham imagens explicativas das visualizações da ferramenta. Um dos participantes respondeu às questões sobre as visualizações com base nessas imagens, fazendo que sua resposta fosse considerada inválida, consequentemente diminuindo a taxa de resposta. Nesse caso, o questionário poderia deixar ainda mais explícito que a figura era somente explicativa e que, para responder às questões, o participante teria que necessariamente abrir a ferramenta.

Houve, também, um participante que abriu o questionário e a ferramenta de seu

celular. A ferramenta não foi implementada visando celulares e tablets, de modo que as visualizações podem não ser exibidas corretamente. Esse fato acabou prejudicando as respostas desse participante e, portanto, sua resposta foi considerada inválida. Para evitar esse caso, o questionário poderia incluir, explicitamente, orientações sobre a não utilização da ferramenta em dispositivos móveis.

4.3.1.2 Validades Internas

Medição do Desempenho

O *PerfMiner* se baseia em múltiplas execuções dos cenários para aumentar a confiança das medidas dos tempos de execução. Nesse estudo, todos os cenários foram executados 10 vezes para a análise do desvio de desempenho, mesmo assim alguns métodos em particular podem ter sido executados mais vezes. Por exemplo, os métodos podem ter várias hierarquias de chamadas, podem ser chamados em loops, etc. Para diminuir os riscos de viés na medição dos tempos de execução, foram tomadas precauções como desabilitar os serviços não essenciais do sistema operacional, cada teste foi executado em sua própria JVM e cada teste foi executado em ordem randômica [22].

Casos de Testes Automatizados

Um dos participantes (P11GA) que respondeu ao questionário online aplicado mostrou preocupação sobre a utilização de testes automatizados na análise. Segundo o participante, o atributo de qualidade de desempenho não é importante para 90% dos testes automatizados do sistema dele, pois eles apenas testam erros e mau comportamento. O *PerfMiner* utiliza esses casos de testes para a análise dos cenários, no entanto, quaisquer estratégias utilizadas para exercitar os cenários são válidas, pois a ferramenta continuará apontando os cenários com desvios de desempenho para os sistemas analisados. Como trabalho futuro, há a expectativa de que a ferramenta forneça a possibilidade de configurações dos cenários desejados para que os usuários escolham, de maneira não intrusiva, os cenários em que o desempenho é importante de acordo com as regras de negócio dos seus sistemas.

Dados Tabulares

Os questionários elaborados para o estudo foram de dois tipos, conforme destacado na subseção 4.1.3.3: tipo 1 e tipo 2. Em ambos, os participantes responderam a questões relacionadas a uma visualização através da ferramenta proposta e a outra visualização através dos dados tabulares, isto é, sem ter acesso à visualização. Essa estratégia foi adotada para que se pudesse obter evidências da eficácia das visualizações, analisando

as respostas dos dois tipos de questionários. Os dados tabulares utilizados são fruto do processamento feito pelo *PerfMiner*, portanto, são dados já pré-processados. Dessa forma, mesmo sem o acesso à visualização, os participantes conseguiram responder às questões de maneira razoável.

Experiência dos Participantes

O questionário online foi submetido para todos os contribuidores dos sistemas no GitHub e, por isso, (i) a experiência dos desenvolvedores com relação à linguagem de programação e ao sistema analisado eram incógnitas, além do (ii) não conhecimento das visualizações recém implementadas. Para diminuir a primeira ameaça, foram adicionadas, no questionário, questões sobre a experiência dos participantes na linguagem e no sistema analisado. Já para diminuir a segunda ameaça, foram apresentadas imagens explicativas nas seções do questionário que continham questões relacionadas às visualizações.

Baixa Taxa de Respostas

Apenas 16 participantes responderam ao questionário, de um total de 114. Das respostas, apenas 13 foram consideradas válidas. Essa baixa taxa de respostas (14,04%) pode ter afetado o resultado do estudo, uma vez que não houve dados o suficiente para extrair conclusões estatísticas.

Respostas Desequilibradas entre os Grupos

Além da baixa taxa de respostas, elas foram desequilibradas entre os grupos divididos no estudo. A quantidade de respostas do questionário de tipo 1 foi maior do que o de tipo 2, fornecendo maior feedback para a visualização do grafo de chamadas do que para a sumarização de cenários. Assim, não foi possível determinar diferenças estatisticamente relevantes para as respostas das questões do questionário do tipo 1 e 2.

Interação entre os Participantes

Uma vez que o questionário foi submetido aos participantes em uma ferramenta online e em um ambiente não controlado, não é possível afirmar que os participantes não interagiram entre si com o intuito de compartilhar informações sobre as suas respostas, oferecendo uma ameaça à autenticidade das respostas obtidas.

4.3.1.3 Validades Externas

Generalização e Limitação dos Resultados

Os resultados encontrados neste estudo não podem ser generalizados. Com relação ao

domínio e quantidade de sistemas analisados, a ferramenta foi aplicada apenas a dois sistemas de domínios diferentes. Para outros domínios e sistemas com outras características, não há evidências de que as visualizações seriam geradas corretamente, tampouco que seriam úteis às equipes de desenvolvimento desses sistemas. Já com relação a quantidade de respostas obtidas, não é possível generalizar os resultados para todos os participantes dos sistemas analisados, uma vez que a amostra foi de baixa representatividade.

4.3.1.4 Validades de Conclusão

Baixo Poder Estatístico

A baixa taxa de resposta obtida dos participantes levou a uma quantidade inadequada de amostras para que o estudo pudesse fornecer conclusões estatísticas.

5 Trabalhos Relacionados

Este capítulo confronta este trabalho com outras pesquisas que analisam a evolução do desempenho de sistemas. Qualquer trabalho ou ferramenta que mensure a evolução do atributo de qualidade de desempenho e possua visualizações para exibi-la é considerado relacionado a este. Na seção 5.1 são comentadas as ferramentas de *profiling*. Na seção 5.2 são mostradas as ferramentas APM. Por fim, na seção 5.3 são discutidas as abordagens de medição da degradação de desempenho.

Muitas abordagens relacionadas a visualização de software têm sido propostas para comparar versões de software de um ponto de vista geral da arquitetura [47][48][49][66][67]. Outras abordagens focam em visualizar as métricas do software em diferentes versões [50][52][53][68]. No entanto, essas abordagens diferem deste trabalho uma vez que o objetivo é comparar um aspecto dinâmico do software, o desempenho, ao invés de aspectos estáticos ou estruturais.

5.1 Ferramentas de Profiling

Há ferramentas de *profiling* que podem realizar a medição do atributo de qualidade de desempenho, no entanto, com diferentes características. O *JProfiler* [15] e o *YourKit Java Profiler* [16] são ferramentas comerciais para realizar profiling de aplicações na linguagem Java. Sandoval Alcocer et al.[13] comentam algumas limitações dessas duas ferramentas:

- *Variações de desempenho têm que ser manualmente rastreadas*: para cada execução, o *profiler* tem que ser manualmente configurado para executar uma versão em particular. Depois, os dados do *profiling* podem ser salvos no sistema de arquivos. Após realizar esse procedimento por duas vezes, ambas as execuções podem ser comparadas. Entretanto, cada execução requer muito trabalho manual;
- *Faltam métricas relevantes*: ambas as ferramentas não consideram se o código-fonte foi alterado ou não. Como consequência, variações de desempenho em métodos não

modificados podem distrair o programador de identificar alterações de código que realmente introduziram as variações;

- *Representações visuais ineficientes*: O *JProfiler* e o *YourKit Java Profiler* usam uma tabela textual incrementada com alguns ícones para indicar variações. Dessa forma, entender qual variação de desempenho decorre de mudanças de software requer um esforço significativo do programador.

Os autores comentam que essas ferramentas, apesar de serem úteis para acompanhar o desempenho geral, são ineficientes para saber a diferença dos tempos dos métodos e, muitas vezes, insuficientes para compreender as razões para a variação de desempenho. Há ainda uma ferramenta que acompanha a JVM, chamada *VisualVM*, que possui as mesmas limitações comentadas anteriormente. Além disso, o *VisualVM* não oferece a comparação entre execuções, tornando difícil a visualização da evolução do desempenho entre as versões do software, uma vez que teria que ser feita manualmente para cada método desejado.

5.2 Ferramentas APM

O trabalho de [Ahmed et al.\[17\]](#) realizou um estudo para verificar se as ferramentas de gerenciamento de desempenho de aplicações - APM - são eficazes na identificação de regressões de desempenho. Os autores definem regressão de desempenho quando as atualizações em um software provocam uma degradação no seu desempenho. As ferramentas utilizadas no estudo foram *New Relic* [18], *AppDynamics* [19], *Dynatrace* [20] e *Pinpoint* [21]. Como resultado, eles mostram que a maioria das regressões inseridas no código-fonte foram detectadas pelas ferramentas. Contudo, o processo de identificação do método exato cujo código foi inserido foi mais complicado, sendo necessário bastante trabalho manual: os autores inspecionavam as transações (requisições) marcadas como lentas e, manualmente, comparavam os respectivos *stacktraces* para verificar se a ferramenta indicava corretamente a regressão de desempenho.

A ferramenta e a extensão proposta neste trabalho é diferente das ferramentas apresentadas no trabalho de [Ahmed et al.\[17\]](#) por realizar a análise de duas versões do software alvo do estudo, por automatizar o processo de identificação da causa do desvio de desempenho, por prover visualizações adequadas à identificação dos desvios de desempenho, bem como exibindo dados adicionais dos nós do grafo, por mostrar a evolução global e por cenário do desempenho e por proporcionar aos desenvolvedores a identificação diretamente

do código-fonte das causas do desvio de desempenho.

5.3 Abordagens de Degradação de Desempenho

O trabalho de [Sandoval Alcocer et al.\[13\]](#) propõe o *Performance Evolution Blueprint*, uma abordagem visual para entender a causa de degradações de desempenho, comparando o desempenho de duas versões do sistema. Trata-se de uma visualização onde formas e cores dos elementos visuais indicam valores de métricas e propriedades do software analisado. O trabalho utiliza a ferramenta *Rizel* para medição de propriedades dos métodos, tais quais: suas métricas, quais métodos foram adicionados, removidos ou modificados, e o seu tempo e número de execução. A abordagem foi desenvolvida na linguagem de programação Pharo. A Figura 48 apresenta essa abordagem. A ferramenta foi avaliada ao ser aplicada em um software chamado Roassal¹.

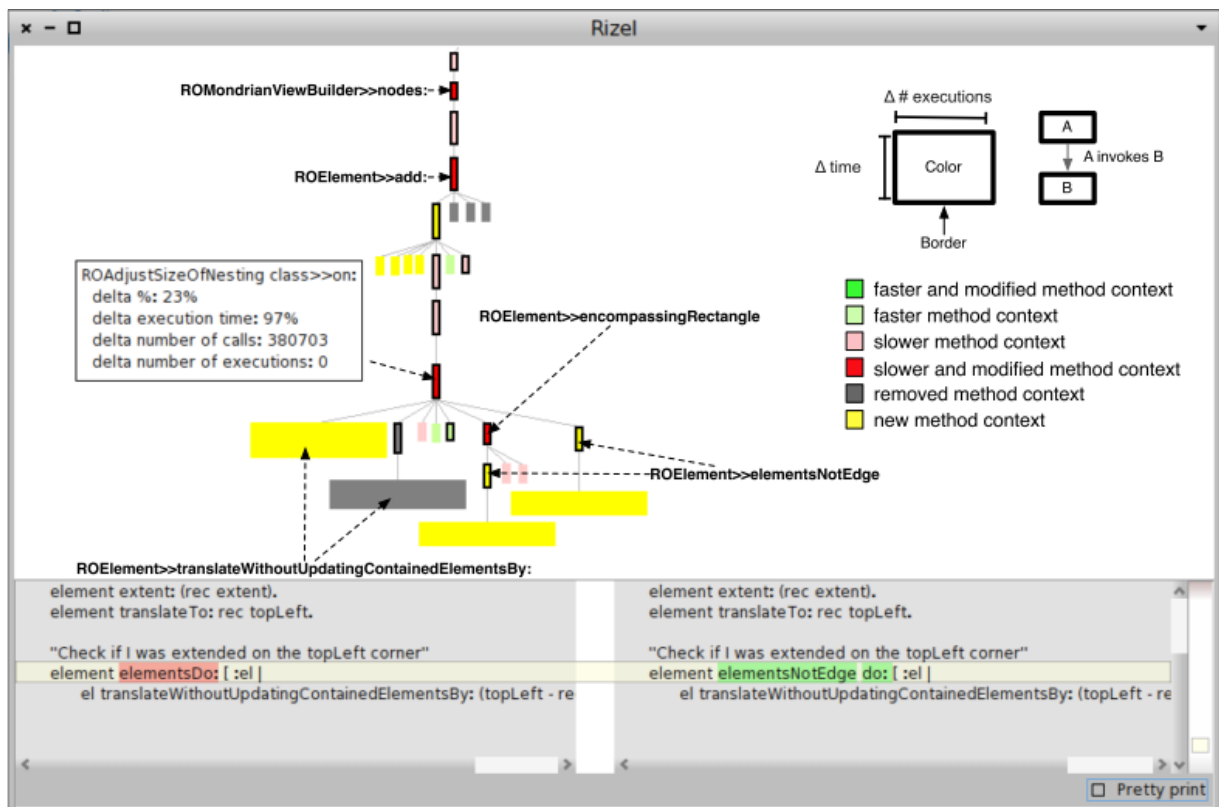


Figura 48: Exemplo do *Performance Evolution Blueprint* [13].

A ferramenta *PerfMiner Visualizer* e o *Performance Evolution Blueprint* utilizam a metáfora de grafo. No entanto, a primeira oferece as seguintes vantagens com relação à segunda:

¹<http://agilevisualization.com>

- *Informações da Hierarquia de Chamadas*: no grafo de chamadas do *PerfMiner Visualizer* são exibidas informações como o tempo de execução total, próprio e de desvio para cada nó, o tempo de execução do cenário (com a porcentagem de variação) e os nomes de cada nó da hierarquia de chamadas (próximos ao nó desviado/adicionado/removido). Essas informações apresentadas pela ferramenta se mostram necessárias, como mencionado pelo participante P5GA: “o gráfico é bastante simples e fácil de entender. Não há muita informação, apenas o necessário.”. Soma-se a isso, as informações contidas na seção de Sumário e Histórico da visualização. Com essas informações, torna-se factível identificar não só as possíveis causas dos desvios de desempenho, mas também a sua gravidade e localização na hierarquia de chamadas;
- *Diferentes Elementos Visuais*: a visualização do grafo de chamadas fornece elementos visuais diferentes para exibir as informações. O nó de agrupamento mostra que existem mais nós que foram omitidos e não estão diretamente ligados aos indicados com desvios/adicionados/removidos. Na visualização proposta por [Sandoval Alcocer et al.\[13\]](#), esses nós são, por padrão, omitidos, mas podem ser exibidos através da interação do usuário. Outro elemento visual são as setas indicativas da porcentagem de degradação ou otimização dos nós, propiciando a identificação da gravidade do desvio de cada nó. Esse elemento visual foi mencionado no estudo, pelo participante P5GA: “Eu gostei .. da seta verde/vermelha indicando o nível de melhoria/degradação.”. No *Performance Evolution Blueprint*, o usuário pode ter acesso às porcentagens de desvio de um nó apenas ao passar o ponteiro do mouse sobre ele. Dessa forma, não há uma visão geral da gravidade do desvio de todos os nós indicados com variações de desempenho;
- *Diferentes Funcionalidades*: o grafo de chamadas do *PerfMiner Visualizer* possui funcionalidades de zoom e destaque da hierarquia de chamada dos nós com desvio/adicionados/removidos. O participante P5GA disse, sobre a funcionalidade de destaque: “Eu gostei do recurso de destaque.”;
- *Maior Potencial de Aplicabilidade*: por ser compatível com a linguagem de programação Java, o potencial de aplicabilidade da ferramenta proposta por esta dissertação é maior. Para efeitos comparativos, no momento da escrita deste trabalho, existem no GitHub mais de 3 milhões e 300 mil projetos *open source* implementados na linguagem Java, ao passo que na linguagem SmallTalk existem pouco mais de 2 mil;

- *Maior Potencial de Utilização*: juntamente com a compatibilidade com a linguagem de programação Java, há maior potencial de utilização da ferramenta por parte dos desenvolvedores e arquitetos dos sistemas. A ferramenta pode ser oferecida como um serviço, pode ser integrada a ferramentas de integração contínua através de *plugins* ou ser oferecida de maneira *standalone*;
- *Avaliação com Participantes*: a avaliação com participantes realizada nesta dissertação é um diferencial com relação ao trabalho de [Sandoval Alcocer et al.\[13\]](#). A partir dessa avaliação foi possível obter valiosas respostas dos usuários com relação a utilidade das visualizações para identificação das possíveis causas de variações de desempenho e a sua aplicabilidade nos processos de desenvolvimento de software das equipes.

[Bergel, Robbes e Binder\[69\]](#) propuseram uma abordagem cujo objetivo é comparar duas versões de um software para a identificação de gargalos de execução. As visualizações propostas foram *Structural Distribution Blueprint* e *Behavioral Distribution Blueprint*. Ambas exibem informações de execução como um *call graph*, onde os nós são os métodos e as arestas são as invocações. Cada nó é renderizado como uma caixa e uma invocação é uma linha que une dois nós. A primeira visualização exibe métrica que indicam a distribuição do tempo de execução ao longo da estrutura estática de um programa. Já a segunda mostra as informações de tempo de execução junto com as chamadas de métodos. Essa abordagem considera apenas se um método gastou ou não mais tempo de execução do que na versão anterior. A Figura 49 exemplifica a visualização proposta pelos autores. A avaliação foi realizada ao aplicar a ferramenta no *framework* Mondrian².

As principais vantagens do trabalho proposto por esta dissertação com relação ao de [Bergel, Robbes e Binder\[69\]](#) são:

- *Mais Métricas*: por adicionar mais métricas no grafo de chamadas, como a porcentagem de desvio de desempenho e se houve métodos adicionados ou removidos, o *PerfMiner Visualizer* fornece a possibilidade de o usuário identificar as possíveis causas e a gravidade dos desvios de desempenho. Essas métricas adicionadas fortalecem a análise da evolução do sistema ao longo das versões;
- *Agrupamento de Nós*: no *PerfMiner Visualizer*, o nó de agrupamento mostra que existem mais nós na hierarquia de chamadas que foram omitidos e não estão diretamente ligados aos indicados com desvios/adicionados/removidos. A proposta de

²<http://scg.unibe.ch/archive/papers/Meye06aMondrian.pdf>

Bergel, Robbes e Binder[69] não menciona sobre escalabilidade em nenhuma das suas visualizações, nem como essa situação é trata;

- *Identificação das Causas dos Desvios*: o grafo de chamadas do *PerfMiner Visualizer* indica os *hashes* dos *commits* que possivelmente foram os responsáveis pelos desvios de desempenho dos nós e, conseqüentemente, do cenário. As visualizações *Structural Distribution Blueprint* e *Behavioral Distribution Blueprint* não fornecem essa possibilidade;
- *Verificação das Mudanças*: outra vantagem é que, uma vez que os *hashes* dos *commits* são exibidos para os nós com desvios/adicionados/removidos, eles podem ser clicados e, assim, o usuário tem acesso às mudanças implementadas no código-fonte que podem ter causado tal desvio de desempenho;
- *Visão Geral dos Cenários*: Bergel, Robbes e Binder[69] não mencionam se há uma forma de se obter uma visão geral dos gargalos de desempenho, para duas versões de um software, encontrados pela abordagem. Para essa finalidade, o *PerfMiner Visualizer* fornece a visualização de sumarização de cenários.

Mostafa e Krintz[70] propõem uma técnica que compara duas árvores de contexto de chamadas (CCT, do inglês *Context Call Tree*), cada uma obtida de uma versão diferente de um software. Os autores apresentam o *PARCS*, uma ferramenta de análise que identifica automaticamente diferenças entre o comportamento da execução de duas revisões de uma aplicação. A abordagem usa como base o algoritmo de correspondência de árvores comuns para comparar duas CCTs. No entanto, o suporte visual usado pelo *PARCS* não representa adequadamente a variação de uma estrutura dinâmica e múltiplas métricas [71]. Outra limitação dessa abordagem é que ela não detecta nós adicionados. Quando comparadas, duas CCTs que diferem apenas em um nó adicionado são consideradas completamente diferentes pelo *PARCS*. A Figura 50 exibe um exemplo dessa abordagem. A avaliação do *PARCS* foi realizada aplicando-o na ferramenta chamada FindBugs³.

O trabalho proposto nesta dissertação oferece as seguintes vantagens ao *PARCS*, dentre outras:

- *Nós Adicionados e Removidos*: o grafo de chamadas do *PerfMiner Visualizer* destaca os nós adicionados e removidos na hierarquia de chamadas de cada cenário. Esses nós são fundamentais para o entendimento da real causa dos desvios de desempenho de determinado cenário ao longo das versões;

³<http://findbugs.sourceforge.net>

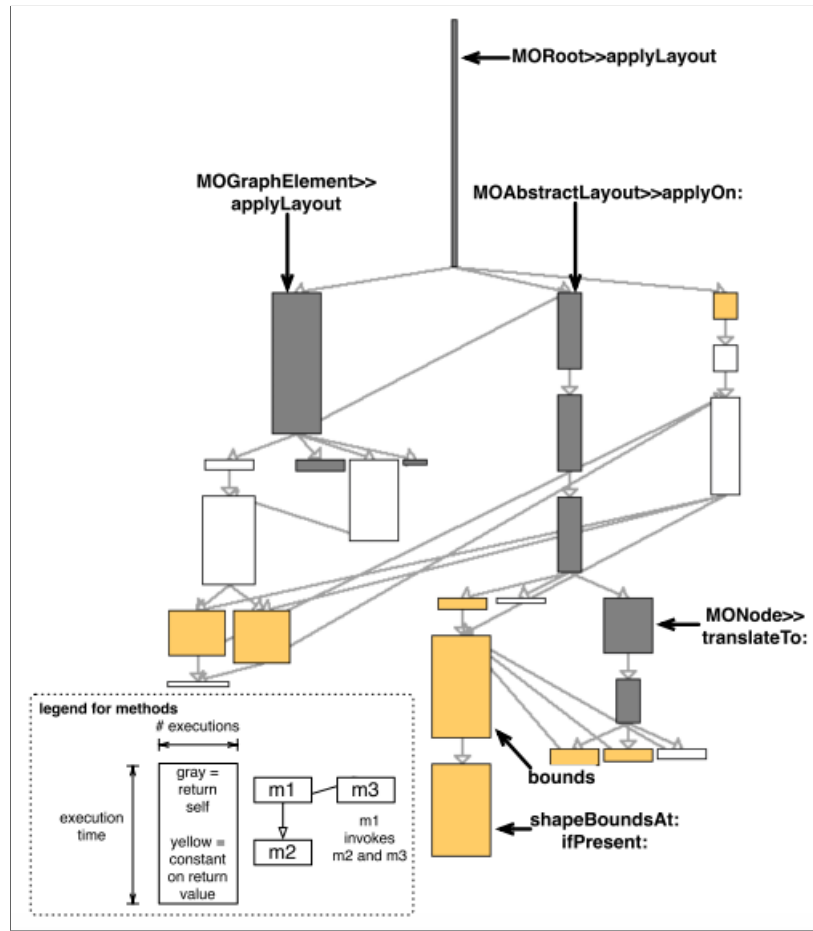


Figura 49: Exemplo da visualização comportamental proposta por Bergel, Robbes e Binder[69].

- *Mais Métricas*: de maneira semelhante ao trabalho relacionado comentado anteriormente, por adicionar mais métricas no grafo de chamadas, como a porcentagem de desvio de desempenho, os tempos de execução total, próprio e de desvio para cada nó, o *PerfMiner Visualizer* fornece a possibilidade de o usuário identificar as possíveis causas e a gravidade dos desvios de desempenho;
- *Identificação das Causas dos Desvios*: diferentemente do trabalho de Mostafa e Krintz[70], o grafo de chamadas do *PerfMiner Visualizer* indica os *hashes* dos *commits* que possivelmente foram os responsáveis pelos desvios de desempenho dos nós e, conseqüentemente, do cenário;
- *Verificação das Mudanças*: por indicar os *hashes* dos *commits*, ao clicar nesses *hashes*, o usuário pode ter acesso às mudanças implementadas no código-fonte que podem ter causado o desvio de desempenho do cenário;
- *Visão Geral dos Cenários*: os autores não mencionam se há uma forma de se obter uma visão geral das comparações entre as revisões da aplicação. A visualização da

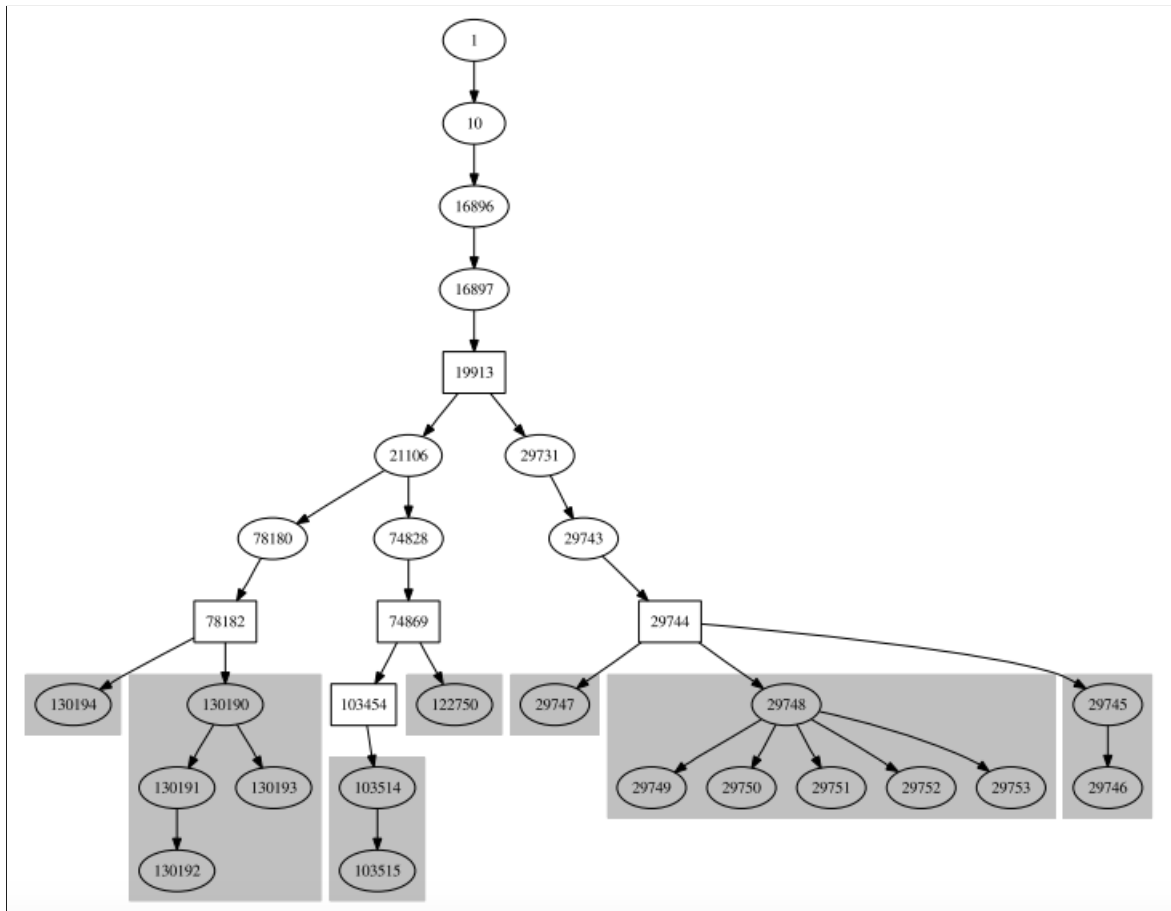


Figura 50: Exemplo do PARCS [70].

sumarização de cenários fornecida pelo *PerfMiner Visualizer* possibilita ao usuário obter uma visão geral de todos os cenários para a análise de um par de versões de um software.

Bezemer, Pouwelse e Gregg[72] realizam a comparação do desempenho de duas versões de um software através de gráficos de chama diferenciais (do inglês, *differential flame graphs* – DFG). Dadas duas versões v1 (anterior) e v2 (atual), a ferramenta realiza a comparação de três maneiras: (i) com v1 como base, (ii) com v2 como base e (iii) apenas as diferenças do item (ii). Utiliza cores para destacar as comparações: o branco indica que não houve mudanças, o azul que houve melhora no desempenho e o vermelho indica que este piorou. Entretanto, a abordagem não indica as causas das variações de desempenho e os autores indicam que o principal desafio é a coleta dos dados, pois são necessárias ferramentas de *profiling* de terceiros para a análise das versões e eles destacam que pode ser difícil coletar os dados para algumas linguagens, como o Java e Python. A Figura 51 mostra o DFG. A avaliação foi feita aplicando a abordagem no software utilitário Rsync⁴.

⁴<https://rsync.samba.org>

hashes, o usuário pode ter acesso às mudanças implementadas no código-fonte que podem ter causado o desvio de desempenho do cenário;

- *Visão Geral dos Cenários*: Os autores não mencionam se, através da abordagem, é possível obter uma visão geral das comparações realizadas. A visualização da sumarização de cenários fornecida pelo *PerfMiner Visualizer* possibilita uma visão geral de todos os cenários para a análise de um par de versões de um software.

6 Conclusão

Este trabalho apresentou uma ferramenta web, chamada *PerfMiner Visualizer*, cujo objetivo é aplicar técnicas de visualização de software para ajudar desenvolvedores e arquitetos a analisar a evolução do atributo de qualidade de desempenho, em termos de tempo de execução, ao longo das versões de um software. Essa ferramenta foi implementada como extensão a outra já existente, o *PerfMiner*. Foram implementadas duas visualizações:

- *Sumarização de Cenários*: visualização que apresenta uma visão geral dos cenários com desvios de desempenho, onde o usuário é capaz de distinguir se um cenário foi degradado ou otimizado, qual o cenário com maior tempo de execução e qual possui o maior desvio de desempenho dentre os apresentados;
- *Grafo de Chamadas*: objetiva mostrar, para determinado cenário, os métodos que potencialmente causaram o desvio de desempenho. Tais métodos são dispostos em um grafo de chamadas com propriedades visuais que destacam quais deles tiveram desvios de desempenho, foram adicionados ou removidos, além de exibir os *commits* que possivelmente foram os causadores desses desvios.

Foi conduzido um estudo onde a ferramenta foi aplicada a dois sistemas reais *open source* de diferentes domínios: Jetty e VRaptor. Para tanto, foram escolhidas dez releases de cada sistema. As visualizações geradas após a análise dessas releases foram, de maneira geral, corretas e satisfatórias. Os cenários, métodos e *commits* foram devidamente representados nas visualizações através das propriedades visuais definidas. Após isso, foi aplicado um questionário online onde os contribuidores do GitHub dos sistemas analisados puderam verificar e dar feedback sobre a ferramenta e suas visualizações.

Através dos estudos conduzidos, espera-se que os desenvolvedores e arquitetos dos sistemas sejam beneficiados ao dispor de uma ferramenta para gerenciar a evolução do atributo de qualidade de desempenho, identificando as causas dos desvios de desempenho das aplicações, tomar ações para sanar problemas inesperados, além de acompanhar a evolução, planejada ou não, desse atributo de qualidade.

O restante deste capítulo está organizado como segue: a seção 6.1 revisita as questões de pesquisa; a seção 6.2 mostra as limitações do trabalho e a seção 6.3 apresenta perspectivas para trabalhos futuros e melhorias da ferramenta.

6.1 Revisão das Questões de Pesquisa

Baseado nos resultados obtidos do estudo realizado e apesar das suas limitações, esta seção revisita as questões de pesquisa deste trabalho. São elas:

- QP1. Os cenários identificados com desvios de desempenho são apresentados claramente nas visualizações implementadas pela ferramenta?** Após aplicação da ferramenta nos sistemas escolhidos, os cenários identificados com desvios de desempenho foram representados nas visualizações da sumarização de cenários, uma para cada par de releases, de cada sistema. A maioria das visualizações (10 de 15) geradas apresentou os cenários de maneira clara, onde o usuário pode distinguir, sem prejuízos, qual o cenário com maior/menor porcentagem de degradação, qual o cenário com maior/menor tempo de execução e qual o tipo de desvio de desempenho (otimização ou degradação) de cada cenário. Das 5 visualizações restantes, em 4 delas a identificação das características comentadas anteriormente foi prejudicada pela quantidade de cenários apresentados na visualização e pela baixa porcentagem de desvio de parte desses cenários em relação aos demais. Para a visualização restante, foi impossível identificar o cenário com maior tempo de execução, pois a sua baixa porcentagem de desvio em relação aos demais tornou impossível distingui-lo visualmente no gráfico de rosca.
- QP2. O algoritmo de redução de nós aplicado pela ferramenta é capaz de reduzir significativamente o número de nós no grafo de chamadas para sistemas reais?** A visualização do grafo de chamadas para cada cenário foi gerada a partir dos dados da hierarquia de chamadas provenientes das análises realizadas nos sistemas. Para 75% dos cenários, o algoritmo de supressão de nós conseguiu reduzir entre 73,77% e 99,83% a quantidade de nós a serem mostrados no grafo. Esse número, em quantidade de nós, é de 2 a 16 nós exibidos no grafo de chamadas.
- QP3. Desenvolvedores e arquitetos conseguem identificar os cenários com variações de desempenho e suas potenciais causas, em termos de métodos e *commits*, através do auxílio visual da ferramenta?** Com o feedback obtido dos

participantes do questionário online, verificou-se que a maioria deles (67%) indicou corretamente os métodos, suas características e *commits* utilizando a visualização do grafo de chamadas. No entanto, para a visualização da sumarização de cenários, não é possível afirmar que esta trouxe ganhos para a identificação dos cenários, uma vez que apenas metade (50%) dos participantes que a avaliaram conseguiram identificar corretamente os cenários e suas propriedades.

QP4. Há indícios de que as visualizações implementadas pela ferramenta são mais eficazes do que os dados tabulares para se encontrar informações sobre os cenários, métodos e *commits*? Também após o feedback obtido dos participantes do questionário online, houve indícios - 67% de acerto nas questões utilizando a visualização - de que a visualização do grafo de chamadas se mostrou mais eficaz do que os dados tabulares (isto é, desconsiderando a existência dessa visualização) para se encontrar informações sobre os métodos responsáveis pelos desvios de desempenho de determinado cenário e os *hashes* dos *commits* potencialmente responsáveis por esse desvio. No entanto, para a visualização da sumarização de cenários, não houve indícios suficientes - 50% de acerto nas questões utilizando a visualização - que permitisse concluir se a visualização se mostrou mais eficaz do que os dados tabulares para identificar os cenários e suas propriedades.

6.2 Limitações

Este trabalho possui limitações no tocante à ferramenta e sua implementação, conforme descritas a seguir nesta seção.

6.2.1 Escalabilidade

Conforme mostrado na subseção 4.2.1.1, existem limitações com relação a escalabilidade da visualização da sumarização de cenários. Para a análise de um par de releases, quando a quantidade de cenários indicados com desvios de desempenho é grande (acima de 20) e há cenários com baixas porcentagens de desvio de desempenho em relação aos demais, a identificação destes pode ser prejudicada ou até mesmo inviabilizada.

O grafo de chamadas também pode possuir limitações de escalabilidade, embora não tenha acontecido em nenhum caso no estudo realizado. A quantidade dos nós com desvios de desempenho de uma versão para outra de um software pode ser grande de modo a comprometer o desempenho da própria visualização, comprometer a sua usabilidade e

visibilidade ou, até mesmo, inviabilizá-la. Embora medidas tenham sido tomadas para minimizar esse impacto - como, por exemplo, o agrupamento de nós não relacionados com nós de desvio, adicionados ou removidos - há possibilidades de acontecer.

No caso de serem analisadas duas versões muito distantes entre si, podem existir muitas modificações, fazendo com que o grafo gerado seja grande e complexo, resultando no problema do tamanho do grafo apresentado no parágrafo anterior. O ideal ao utilizar a ferramenta é analisar versões próximas para beneficiar-se da baixa granularidade apresentada pelo grafo de chamadas.

6.2.2 Automação

A análise do desempenho dos cenários contidos nas versões dos sistemas é feita de forma automatizada pelo *PerfMiner*, entretanto, é reconhecido que ainda há passos manuais a serem feitos, conforme destacado na subseção 4.1.3: (i) os códigos-fonte das versões são baixados e configurados manualmente para executar sem erros; (ii) a configuração do suporte ao *AspectJ* e a inclusão das bibliotecas do próprio *PerfMiner* são realizadas manualmente; (iii) o início da execução dos testes é dado de forma manual, mesmo que se utilize ferramentas que automatizam a execução de cada teste, como o Maven ou o Gradle¹; e (iv) os artefatos de saída da execução do *PerfMiner* são importados manualmente na ferramenta *PerfMiner Visualizer*, através da funcionalidade de Nova Análise (3.2.1).

6.2.3 Dependência de Linguagem de Programação ou Plataforma

O fato de a ferramenta ser capaz de analisar apenas sistemas desenvolvidos na linguagem de programação Java se mostra uma limitação, por ficar dependente de plataforma e linguagem. Por outro lado, a propicia uma alta aplicabilidade uma vez que, como mencionado na seção 5.3, existem no GitHub mais de 3 milhões e 300 mil projetos *open source* implementados nessa linguagem (até o momento da escrita desta dissertação). Entretanto, é possível que a ferramenta seja utilizada para analisar e visualizar os desvios de desempenho de softwares implementados em outras linguagens e plataformas, desde que o *PerfMiner* seja portado e adaptado para tal.

¹<https://gradle.org>

6.3 Trabalhos Futuros

Os seguintes trabalhos futuros foram vislumbrados a partir da condução desta dissertação.

6.3.1 Atributos de Qualidade Adicionais

A ferramenta apresenta a evolução do desempenho dos cenários das versões de um sistema em termos de tempo de execução. Para determinados sistemas, outros atributos de qualidade podem ser tão ou mais importantes do que o desempenho. Como trabalho futuro, a ferramenta pode ser expandida para medir o desempenho a partir de outras propriedades, como consumo de memória, atividades de disco e uso de CPU, mas também pode ser adicionado o suporte para novos atributos de qualidade, como segurança e confiabilidade. Um dos desafios encontrados ao se expandir a ferramenta para outros atributos de qualidade é como medi-los. É provável que para representar os novos atributos de qualidade seja necessário implementar novas visualizações com novas metáforas visuais de acordo com as características do atributo de qualidade desejado.

6.3.2 Aplicação em um Ambiente Real de Desenvolvimento

A maioria dos participantes indicou que utilizaria a ferramenta como parte integrante dos seus processos de desenvolvimento de software. A ferramenta foi aplicada em sistemas *open source* de diferentes domínios, como o Jetty e o VRaptor. Após isso, os desenvolvedores desses sistemas acessaram as visualizações implementadas para responder ao questionário online. Planeja-se implantar a ferramenta em um ambiente real de desenvolvimento para que os desenvolvedores e arquitetos experimentem, na prática, a sua utilização ao longo das releases do sistema. Uma empresa candidata é a Superintendência de Informática (SINFO) da UFRN pelo fato de ser uma empresa sediada na universidade, de fácil acesso e possuir sistemas de larga escala desenvolvidos na linguagem Java.

6.3.3 Novas Avaliações

As avaliações realizadas neste trabalho incluíram a aplicação da ferramenta em dois sistemas reais de diferentes domínios e a aplicação do questionário junto aos desenvolvedores desses sistemas. Há planos para que sejam feitas mais avaliações a fim de obter mais indícios sobre a utilidade da ferramenta e suas visualizações. Tais estudos seriam:

- (i) *Aplicar a abordagem em sistemas open source*: outros sistemas *open source* seriam escolhidos e então a abordagem seria aplicada. Após isso, o questionário seria submetido aos desenvolvedores e arquitetos desses sistemas, como foi feito no estudo relatado neste trabalho;
- (ii) *Experimento controlado I*: a partir dos mesmos resultados já obtidos, realizar um experimento controlado com desenvolvedores e arquitetos de empresas da cidade de Natal/RN. Dessa forma, o questionário poderia ser modificado para incluir outros aspectos não medidos neste trabalho, como a usabilidade ou o tempo de resposta para questões relacionadas a desvios de desempenho;
- (iii) *Experimento controlado II*: contactar os representantes de empresas de desenvolvimento da cidade de Natal/RN com a finalidade de aplicar a ferramenta em seus sistemas. Após isso, convidar os desenvolvedores desses sistemas para um experimento controlado, aplicando um questionário para coletar feedback sobre as visualizações. Assim como no ponto anterior, o questionário poderia ser modificado para incluir aspectos ainda não avaliados.

6.3.4 Novas Estratégias para Exercitar os Cenários

Como discutido na subseção 4.3.1.2, a ferramenta utiliza atualmente casos de testes automatizados para exercitar os cenários e realizar a análise, estratégia essa que foi criticada por um dos participantes do estudo (P11GA). No entanto, outras estratégias também são válidas, pois a ferramenta continuará apontando os cenários com desvios de desempenho. Nesse sentido, há planos futuros para que a ferramenta permita a configuração dos cenários desejados a serem considerados na análise, de maneira não intrusiva. Dessa forma, os usuários poderão escolher os cenários que devem ter o desempenho medido pela ferramenta, de acordo com as regras de negócio dos seus sistemas.

6.3.5 Automação

A abordagem realiza a comparação automática de duas releases de um sistema, indicando os cenários com desvios de desempenho. Embora essa comparação seja automática, há ainda passos manuais a serem executados, conforme destacado na subseção 4.1.3 e comentado novamente na subseção 6.2.2. Como trabalho futuro, planeja-se automatizar os passos executados manualmente, como a obtenção dos códigos-fonte das releases, configuração do suporte ao *AspectJ* e ao *PerfMiner*, execução da estratégia para exercitar os

cenários (através do Maven, Gradle ou outra ferramenta que automatize esse processo), execução da ferramenta e a importação dos artefatos de saída para o *PerfMiner Visualizer* para que as visualizações sejam geradas.

6.3.6 Customização

A ferramenta foi implementada até então para processar os artefatos de saída do *PerfMiner* e gerar as visualizações, sem opções de configuração no ato do processamento nem nas visualizações. Como proposta de trabalho futuro, a ferramenta pode oferecer opções de configuração para o processamento, como quais cenários o usuário deseja gerar as visualizações, quantos nós na hierarquia de chamadas devem ser mostrados além do nó com desvio, etc. Na visualização da sumarização de cenários, é possível implementar filtros para o usuário escolher as informações que deseja ver, como a ordenação dos cenários com maior porcentagem de desvio de desempenho, com o maior tempo de execução e exibir somente degradações/otimizações. Já para a visualização do grafo de chamadas, também podem ser implementados filtros para os nós, como exibir apenas os nós de um tipo (degradado, otimizado, adicionado ou removido).

6.3.7 Disponibilização para a Comunidade

O questionário online mostrou que a maioria dos participantes observam benefícios de utilização da ferramenta também a utilizariam como parte integrante de seus processos de desenvolvimento. A partir desse feedback, há planos futuros de disponibilizar a ferramenta para a comunidade para que desenvolvedores façam uso e tirem proveito de suas funcionalidades. Dentre as maneiras de se fazer isso, incluem o desenvolvimento e disponibilização de *plugins* para ferramentas de integração contínua como o Hudson CI, Jenkins CI ou Travis CI, e a disponibilização de uma versão que possa ser utilizada de maneira *standalone*, sem estar necessariamente integrada a ferramentas de integração contínua.

A fim de disponibilizar uma ferramenta fácil de usar, robusta e configurável, é de fundamental importância a implementação das modificações comentadas nas subseções 6.3.4, 6.3.5 e 6.3.6, além de outras que venham a ser vislumbradas durante o aprimoramento da ferramenta.

Referências

- [1] CASERTA, P.; ZENDRA, O. Visualization of the static aspects of software: A survey. *IEEE Transactions on Visualization and Computer Graphics*, v. 17, n. 7, p. 913–933, 2011. ISSN 10772626.
- [2] ABREU, F.; GOULÃO, M. Toward the design quality evaluation of object-oriented software systems. *Proc. 5th Int'l Conf. Software Quality*, n. October, p. 44–57, 1995. Disponível em: <<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.6468&rep=rep1&type=pdf>>.
- [3] CORBI, T. A. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, v. 28, n. 2, p. 294–306, 1989. ISSN 0018-8670.
- [4] NOVAIS, R. L. et al. Software evolution visualization: A systematic mapping study. *Information and Software Technology*, Elsevier B.V., v. 55, n. 11, p. 1860–1883, 2013. ISSN 09505849. Disponível em: <<http://dx.doi.org/10.1016/j.infsof.2013.05.008>>.
- [5] PETRE, M.; QUINCEY, E. D. A gentle overview of software visualisation. *PPIG News Letter*, n. September, p. 1 – 10, 2006. Disponível em: <<http://www.labri.fr/perso/fleury/courses/PdP/SoftwareVisualization/1-overview-swviz.pdf>>.
- [6] GHANAM, Y.; CARPENDALE, S. A survey paper on software architecture visualization. *University of Calgary, Tech. Rep*, 2008. Disponível em: <http://dSPACE.ucalgary.ca/bitstream/1880/46648/1/2008-906-19.pdf?origin=publication_detail>.
- [7] KHAN, T. et al. Visualization and evolution of software architectures. *OpenAccess Series in Informatics*, v. 27, p. 25–42, 2012. ISSN 21906807. Disponível em: <<http://dx.doi.org/10.4230/OASICS.VLUDS.2011.25>>.
- [8] WILLIAMS, L.; SMITH, C. Performance evaluation of software architectures. ... *workshop on Software and performance*, n. c, p. 164–177, 1998. Disponível em: <<http://dl.acm.org/citation.cfm?id=287353>>.
- [9] PABLO, J. et al. Learning from Source Code History to Identify Performance Failures Project under Study. p. 37–48, 2016.
- [10] BELADY, L. A.; LEHMAN, M. M. A model of large program development. *IBM Systems Journal*, v. 15, n. 3, p. 225–252, 1976. ISSN 0018-8670.
- [11] Lehman, Ramil, Wernick, Perry, T. Metrics and laws of software evolution—the nineties view. In: *IEEE, Piscataway*. [S.l.]: IEEE, 1997. p. 20–32.

- [12] MOLYNEAUX, I. *The Art of Application Performance Testing*. [S.l.]: O'Reilly Media, Inc., 2009. 159 p. ISBN 9780596551056.
- [13] Sandoval Alcocer, J. P. et al. Performance evolution blueprint: Understanding the impact of software evolution on performance. *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*, 2013.
- [14] VISUALVM. *VisualVM*. 2016. Disponível em: <<https://visualvm.github.io>>.
- [15] JPROFILER. JProfiler. 2016. Disponível em: <<https://www.ej-technologies.com/products/jprofiler/overview.html>>.
- [16] PROFILER, Y. J. *YourKit Java Profiler*. 2016. Disponível em: <<https://www.yourkit.com/java/profiler/features/>>.
- [17] AHMED, T. M. et al. Studying the effectiveness of application performance management (APM) tools for detecting performance regressions for web applications. *Proceedings of the 13th International Conference on Mining Software Repositories - MSR '16*, p. 1–12, 2016. Disponível em: <<http://dl.acm.org/citation.cfm?doid=2901739.2901774>>.
- [18] New Relic, I. *New Relic*. 2015. Disponível em: <<http://newrelic.com/>>.
- [19] APPDYNAMICS. *Appdynamics*. 2016. Disponível em: <<https://www.appdynamics.com/>>.
- [20] DYNATRACE. *Dynatrace*. 2016. Disponível em: <<http://www.dynatrace.com/>>.
- [21] PINPOINT. *Pinpoint*. 2016. Disponível em: <<https://github.com/naver/pinpoint>>.
- [22] PINTO, F. A. P. *An Automated Approach for Performance Deviation Analysis of Evolving Software Systems*. 154 p. Tese (Doutorado) — UFRN - Universidade Federal do Rio Grande do Norte, 2015.
- [23] MALIK, H.; HEMMATI, H.; HASSAN, A. E. Automatic detection of performance deviations in the load testing of Large Scale Systems. In: *Proceedings - International Conference on Software Engineering*. [s.n.], 2013. p. 1012–1021. ISBN 9781467330763. ISSN 02705257. Disponível em: <<https://doi.org/10.1109/ICSE.2013.6606651>>.
- [24] TAYLOR, R. N.; MEDVIDOVIC, N.; DASHOFY, E. M. *Software architecture: foundations, theory, and practice*. [S.l.]: Wiley Publishing, 2009. ISSN 0270-5257. ISBN 978-1-60558-719-6.
- [25] BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. [S.l.]: Pearson Education India, 2012. 1–426 p. ISSN 03008495. ISBN 0321154959.
- [26] CLEMENTS, P.; KAZMAN, R.; KLEIN, M. Evaluating Software Architectures: Methods and Case Studies. In: *Addison Wesley Longman SEI Series In Software Engineering*. Addison-Wesley, 2001. p. 368. ISBN 020170482X. Disponível em: <<http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/020170482X>>.

- [27] SVAHNBERG, M. et al. A Method for Understanding Quality Attributes in Software Architecture Structures. *SEKE 2002 - 14th international conference on Software engineering and knowledge engineering*, p. 819–826, 2002. Disponível em: <<http://portal.acm.org/citation.cfm?doid=568760.568900>>.
- [28] BABAR, M.; GORTON, I. Comparison of scenario-based software architecture evaluation methods. *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, p. 600–607, 2004. ISSN 1530-1362. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1371976>.
- [29] DIEHL, S. *Software visualization: Visualizing the structure, behaviour, and evolution of software*. [S.l.: s.n.], 2007. 1–187 p. ISSN 0018-9162. ISBN 9783540465041.
- [30] GOMEZ-HENRIQUEZ, L. M. Software Visualization: An Overview. *Informatik*, v. 2, n. 2, p. 4–7, 2001. Disponível em: <<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.23.8935&rep=rep1&type=pdf>>.
- [31] MALETIC, J. I.; MARCUS, A.; COLLARD, M. L. A Task Oriented View of Software Visualization. *Proceedings of the First International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'02)*, 2002.
- [32] DENFORD, M.; O'NEILL, T.; LEANEY, J. Architecture-based visualisation of computer based systems. In: *Engineering of Computer-Based Systems*. IEEE, 2002. p. 139–146. Disponível em: <<http://ieeexplore.ieee.org/document/999832/>>.
- [33] GALLAGHER, K.; HATCH, A.; MUNRO, M. A framework for software architecture visualization assessment. 2005. Disponível em: <<http://ieeexplore.ieee.org/document/1684309/>>.
- [34] GALLAGHER, K. et al. Software Architecture Visualization: An Evaluation Framework and Its Application. *Software Engineering, IEEE Transactions on*, v. 34, n. 2, p. 260–270, 2008. ISSN 0098-5589.
- [35] AMBROS, M. D.; LANZA, M. BugCrawler: Visualizing Evolving Software Systems. *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, p. 4–5, 2007. Disponível em: <<http://ieeexplore.ieee.org/document/4145055/>>.
- [36] JOHNSON, B.; SHNEIDERMAN, B. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. *Proceedings., IEEE Conference on*, p. 284–291, 1991.
- [37] WANG, W. et al. Visualization of Large Hierarchical Data by Circle Packing. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, 2006. (CHI '06), p. 517–520. ISBN 1-59593-372-7. Disponível em: <<http://doi.acm.org/10.1145/1124772.1124851>>.
- [38] BARLOW, T.; NEVILLE, P. A Comparison of 2-D Visualizations of Hierarchies. *Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS'01)*, v. 2001, 2001.
- [39] HOLTEN, D. Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. *IEEE Transactions on Visualization and Computer Graphics*, v. 12, n. 5, p. 741–748, 2006.

- [40] PINZGER, M. et al. A tool for visual understanding of source code dependencies. *IEEE International Conference on Program Comprehension*, p. 254–259, 2008. ISSN 1063-6897.
- [41] ALAM, S.; DUGERDIL, P. EvoSpaces: 3D visualization of software architecture. In: *19th International Conference on Software Engineering and Knowledge Engineering, SEKE 2007*. [S.l.: s.n.], 2007. p. 500–505. ISBN 9781627486613.
- [42] BALZER, M.; DEUSSEN, O. Level-of-detail visualization of clustered graph layouts. *Asia-Pacific Symposium on Visualisation 2007, APVIS 2007, Proceedings*, p. 133–140, 2007.
- [43] TERMEER, M. et al. Visual exploration of combined architectural and metric information. *Proceedings - VISSOFT 2005: 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, p. 21–26, 2005.
- [44] BYELAS, H.; TELEA, A. Visualizing metrics on areas of interest in software architecture diagrams. *IEEE Pacific Visualization Symposium, PacificVis 2009 - Proceedings*, p. 33–40, 2009. ISSN 1045-926X.
- [45] LANZA, M. et al. CodeCrawler - an information visualization tool for program comprehension. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, n. June 2016, p. 2–3, 2005. Disponível em: <<http://www.old.inf.usi.ch/faculty/lanza/Downloads/Lanz05a.pdf>>.
- [46] MAGNAVITA, R.; NOVAIS, R.; MENDONÇA, M. Using EVOWAVE to analyze software evolution. *ICEIS 2015 - 17th International Conference on Enterprise Information Systems, Proceedings*, v. 2, p. 126–136, 2015. Disponível em: <<http://www.scopus.com/inward/record.url?eid=2-s2.0-84939532296&partnerID=tZ0tx3y1>>.
- [47] STEINBRÜCKNER, F.; LEWERENTZ, C. Representing development history in software cities. *Proc. 5th Int. Symp. Softw. Vis.*, p. 193–202, 2010. ISSN 15437221. Disponível em: <<http://dl.acm.org/citation.cfm?id=1879211.1879239>>.
- [48] TELEA, A.; AUBER, D. Code flows: Visualizing structural evolution of source code. *Computer Graphics Forum*, v. 27, n. 3, p. 831–838, 2008. ISSN 01677055.
- [49] COLLBERG, C. et al. A system for graph-based visualization of the evolution of software. *Proceedings of the 2003 ACM symposium on Software visualization - SoftVis '03*, p. 77, 2003. Disponível em: <<http://portal.acm.org/citation.cfm?doid=774833.774844>>.
- [50] LANGELIER, G.; SAHRAOUI, H.; POULIN, P. Exploring the evolution of software quality with animated visualization. *Proceedings - 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2008*, p. 13–20, 2008. ISSN 1943-6092.
- [51] WETTEL, R.; LANZA, M. Visualizing software systems as cities. *VISSOFT 2007 - Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, p. 92–99, 2007.

- [52] LANZA, M. The evolution matrix: Recovering software evolution using software visualization techniques. *Proceedings of the 4th international workshop on principles of software evolution*, p. 37–42, 2001. Disponível em: <<http://portal.acm.org/citation.cfm?id=602461.602467>>.
- [53] PINZGER, M. et al. Visualizing multiple evolution metrics. *Proceedings of the 2nd ACM symposium on Software visualization*, v. 1, n. 212, p. 67, 2005. Disponível em: <<http://dl.acm.org/citation.cfm?id=1056018.1056027>>.
- [54] MCCONATHY, D. A. Evaluation methods in visualization: Combating the emperor's new clothes phenomenon. *ACM SIGBIO Newsletter*, 1993. Disponível em: <<http://portal.acm.org/citation.cfm?id=163425>>.
- [55] KOMLODI, A.; SEARS, A.; STANZIOLA, E. *Information Visualization Evaluation Review*. [S.l.], 2004.
- [56] SERIAI, A. et al. Validation of Software Visualization Tools: A Systematic Mapping Study. *2014 Second IEEE Working Conference on Software Visualization*, p. 60–69, 2014.
- [57] DMITRIEV, M. Profiling Java applications using code hotswapping and dynamic call graph revelation. *ACM SIGSOFT Software Engineering Notes*, v. 29, n. 1, p. 139, 2004. ISSN 01635948.
- [58] BATEMAN, S. et al. Interactive usability instrumentation. *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, p. 45–54, 2009. Disponível em: <<http://doi.acm.org/10.1145/1570433.1570443>>.
- [59] Apache JMeter. *Apache JMeter*. 2016. 1 p. Disponível em: <<http://jmeter.apache.org/>>.
- [60] NEUHÄUSER, M. Wilcoxon–Mann–Whitney Test. In: LOVRIC, M. (Ed.). *International Encyclopedia of Statistical Science*. Springer Berlin Heidelberg, 2011. p. 1656–1658. ISBN 978-3-642-04897-5. Disponível em: <<http://www.amazon.com/International-Encyclopedia-Statistical-Science-Miodrag/dp/3642048978>>.
- [61] SPENCE, I. No Humble Pie: The Origins and Usage of a Statistical Chart. *Journal of Educational and Behavioral Statistics*, v. 30, n. 4, p. 353–368, 2005. ISSN 1076-9986.
- [62] HERMAN, I.; MELANÇON, G.; MARSHALL, M. S. Graph visualization and navigation in information visualization: a survey. *IEEE Transactions on Visualization and Computer Graphics*, v. 6, n. 1, p. 24–43, 2000. ISSN 10772626.
- [63] JETTY. *Jetty*. 2016. Disponível em: <<http://www.eclipse.org/jetty/>>.
- [64] VRAPTOR. *VRaptor*. 2017. Disponível em: <<http://www.vraptor.org>>.
- [65] WOHLIN, C. et al. *Experimentation in software engineering*. [S.l.: s.n.], 2012. ISSN 1098-6596. ISBN 9788578110796.
- [66] EICK, S. G.; STEFFEN, J. L.; SUMNER, E. E. Seesoft—A Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, v. 18, n. 11, p. 957–968, 1992. ISSN 00985589.

- [67] HOLTEN, D.; Van Wijk, J. J. Visual comparison of hierarchically organized data. *Computer Graphics Forum*, v. 27, n. 3, p. 759–766, 2008. ISSN 01677055.
- [68] WETTEL, R.; LANZA, M. Visual exploration of large-scale system evolution. *Proceedings - Working Conference on Reverse Engineering, WCRE*, p. 219–228, 2008. ISSN 10951350.
- [69] BERGEL, A.; ROBBES, R.; BINDER, W. Visualizing dynamic metrics with profiling blueprints. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 6141 LNCS, p. 291–309, 2010. ISSN 03029743.
- [70] MOSTAFA, N.; KRINTZ, C. Tracking performance across software revisions. *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java - PPPJ '09*, p. 162–171, 2009. Disponível em: <<http://portal.acm.org/citation.cfm?doid=1596655.1596682>>.
- [71] PABLO, J.; ALCOCER, S.; BERGEL, A. Tracking performance failures with rizeL. *International Workshop on Principles of Software Evolution (IWPSE)*, n. Dcc, p. 38–42, 2013.
- [72] BEZEMER, C. P.; POUWELSE, J.; GREGG, B. Understanding software performance regressions using differential flame graphs. In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*. [S.l.: s.n.], 2015. p. 535–539. ISBN 9781479984695.

APÊNDICE A – Questionário

Este apêndice apresenta a versão em português dos questionários de tipo 1 (A.1) e tipo 2 (A.2) aplicados com os participantes do estudo apresentado no capítulo 4, intitulado “Avaliação”. Através desses questionários foram coletados os feedbacks de 16 contribuidores dos projetos Jetty e VRaptor. Todas as questões foram opcionais.

A.1 Tipo 1

PerfMiner Visualizer

Este questionário é parte de um projeto de pesquisa que propõe uma ferramenta de visualização de software que ajuda os desenvolvedores a gerenciar o desempenho dos sistemas de software, em termos de tempo de execução/resposta. Estamos atualmente conduzindo um estudo sobre o framework Jetty/VRaptor e ficaríamos gratos se você pudesse responder as perguntas abaixo.

Um conceito importante para o estudo é o cenário. Ele é definido pela interação entre os stakeholders e o sistema. Neste estudo, um cenário foi definido como um caso de teste automatizado do sistema.

Página 1/5 - Demográfico

Questão 1. *Qual sua ocupação atual no seu trabalho? (caixa de texto)*

Questão 2. *Quantos anos de experiência você possui no desenvolvimento de software em Java? (múltipla escolha)*

Questão 3. *Quantas contribuições você fez para o projeto Jetty/VRaptor nos últimos 12 meses? (múltipla escolha)*

Página 2/5 - Atributo de Qualidade de Desempenho

Questão 4. Você costuma usar ou já usou alguma ferramenta de análise de desempenho (ferramentas de profiling, APM) no Jetty/VRaptor? (múltipla escolha)

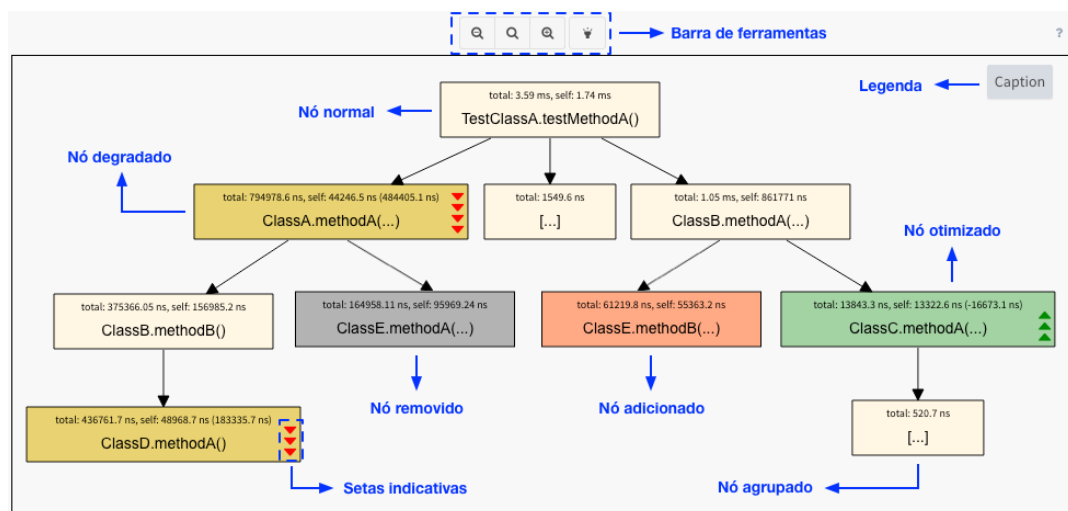
Questão 5. Se você respondeu “Uso essas ferramentas frequentemente” ou “Já usei essas ferramentas” na pergunta anterior, você poderia especificar qual(is) ferramenta(s) você usa ou já usou? (caixa de texto)

Questão 6. Suponha que uma funcionalidade do Jetty/VRaptor foi evoluída (por você ou outro membro da equipe). O que você faz para garantir que o tempo de execução/resposta de tal funcionalidade é aceitável em comparação com outras releases? (caixa de texto)

Página 3/5 - Grafo de Chamadas

A figura abaixo mostra uma breve explicação da visualização interativa do Grafo de Chamadas, que mostra os métodos que potencialmente causaram um desvio de desempenho para um determinado cenário (executado através de um caso de teste automatizado) do Jetty/VRaptor.

As próximas perguntas se referirão a uma visualização concreta da evolução de uma release do projeto Jetty/VRaptor. Use versões recentes do Google Chrome ou Mozilla Firefox para abrir os links das visualizações da ferramenta.



Questão 7. Considerando a visualização "Grafo de Chamadas" para o caso de teste <nome_do_caso_de_teste> acessado através do link <link_para_a_ferramenta>,

you can identify the possible methods responsible for the deviation of performance of the scenario? List these methods in two groups: optimized and degraded. (text box)

Questão 8. *Quão fácil foi responder à pergunta anterior? (múltipla escolha - escala Likert de 5 itens)*

Questão 9. *Considerando a visualização "Grafo de Chamadas" da ferramenta acessada através do link <link_para_a_ferramenta>, identifique o hash do commit responsável pelo principal desvio de desempenho do sistema. (caixa de texto)*

Questão 10. *Quão fácil foi responder à pergunta anterior? (múltipla escolha - escala Likert de 5 itens)*

Questão 11. *Este desvio parece plausível de acordo com o seu conhecimento do sistema? (caixa de texto)*

Questão 12. *Você poderia mencionar aspectos dessa visualização que você gostou? E quais outros aspectos você não gostou? Você tem alguma sugestão de melhoria ou comentário para essa visualização? (caixa de texto)*

Página 4/5 - Sumarização de Cenários

The link below provides the raw data for the scenarios (executed through a case of automated testing) with performance deviations for two versions of Jetty/VRaptor: <link_para_os_dados_tabulares>.

Questão 13. *Considerando os dados acessados através do link acima, você consegue identificar qual cenário (isto é, caso de teste) possui maior desvio de desempenho (degradação ou melhoria do tempo de execução/tempo de resposta) dentre os exibidos? O cenário foi otimizado ou degradado? (caixa de texto)*

Questão 14. *Quão fácil foi responder à pergunta anterior? (múltipla escolha - escala Likert de 5 itens)*

Questão 15. *Considerando os dados acessados através do link acima, você consegue identificar qual cenário possui maior tempo de execução/resposta dentre os exibidos? O cenário foi otimizado ou degradado? (caixa de texto)*

Questão 16. *Quão fácil foi responder à pergunta anterior? (múltipla escolha - escala*

Likert de 5 itens)

Página 5/5 - Geral

Questão 17. *Você vê benefícios de usar a ferramenta de visualização de desvios de desempenho apresentada? Se sim, quais? (caixa de texto)*

Questão 18. *Você utilizaria a ferramenta como parte integrante do processo de desenvolvimento de software do Jetty/VRaptor? Se sim, como você vislumbra que ela seria utilizada? (caixa de texto)*

Questão 19. *Utilize o espaço abaixo para incluir comentários adicionais que deseje. (caixa de texto)*

Questão 20. *Você está disponível para ser contactado para discutir nossos resultados relacionados ao projeto Jetty/VRaptor? Se sim, por favor deixe o seu e-mail abaixo. (caixa de texto)*

Você pode acessar e verificar todos os resultados gerados pela nossa ferramenta através deste link: <http://apvis.herokuapp.com/>

A.2 Tipo 2

PerfMiner Visualizer

Este questionário é parte de um projeto de pesquisa que propõe uma ferramenta de visualização de software que ajuda os desenvolvedores a gerenciar o desempenho dos sistemas de software, em termos de tempo de execução/resposta. Estamos atualmente conduzindo um estudo sobre o framework Jetty/VRaptor e ficaríamos gratos se você pudesse responder as perguntas abaixo.

Um conceito importante para o estudo é o cenário. Ele é definido pela interação entre os stakeholders e o sistema. Neste estudo, um cenário foi definido como um caso de teste automatizado do sistema.

Página 1/5 - Demográfico

Questão 1. Qual sua ocupação atual no seu trabalho? (caixa de texto)

Questão 2. Quantos anos de experiência você possui no desenvolvimento de software em Java? (múltipla escolha)

Questão 3. Quantas contribuições você fez para o projeto Jetty/VRaptor nos últimos 12 meses? (múltipla escolha)

Página 2/5 - Atributo de Qualidade de Desempenho

Questão 4. Você costuma usar ou já usou alguma ferramenta de análise de desempenho (ferramentas de profiling, APM) no Jetty/VRaptor? (múltipla escolha)

Questão 5. Se você respondeu “Uso essas ferramentas frequentemente” ou “Já usei essas ferramentas” na pergunta anterior, você poderia especificar qual(is) ferramenta(s) você usa ou já usou? (caixa de texto)

Questão 6. Suponha que uma funcionalidade do Jetty/VRaptor foi evoluída (por você ou outro membro da equipe). O que você faz para garantir que o tempo de execução/resposta de tal funcionalidade é aceitável em comparação com outras releases? (caixa de texto)

Página 3/5 - Grafo de Chamadas

O link abaixo fornece os dados brutos para os métodos que potencialmente causaram um desvio de desempenho para um determinado cenário (executado através de um caso de teste automatizado) do Jetty/VRaptor: <link_para_os_dados_tabulares>.

Questão 7. Considerando os dados para o caso de teste <nome_do_caso_de_teste> acessados através do link acima, você consegue identificar os possíveis métodos responsáveis pelo desvio de desempenho do cenário? Liste tais métodos em dois grupos: otimizados e degradados. (caixa de texto)

Questão 8. Quão fácil foi responder à pergunta anterior? (múltipla escolha - escala Likert de 5 itens)

Questão 9. Considerando os dados acessados através do link acima, identifique o hash do commit responsável pelo principal desvio de desempenho do sistema. (caixa de texto)

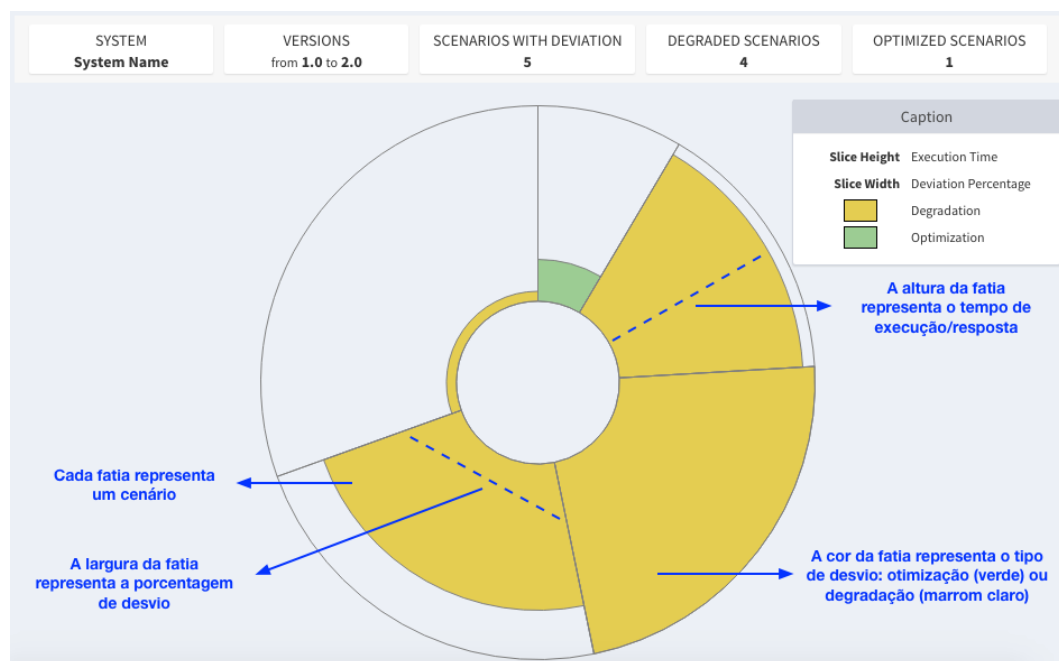
Questão 10. *Quão fácil foi responder à pergunta anterior? (múltipla escolha - escala Likert de 5 itens)*

Questão 11. *Este desvio parece plausível de acordo com o seu conhecimento do sistema? (caixa de texto)*

Página 4/5 - Sumarização de Cenários

A figura abaixo mostra uma breve explicação da visualização interativa da Sumarização de Cenários, que mostra os cenários (executado através de um caso de teste automatizado) com desvios de desempenho para duas versões do Jetty/VRaptor.

As próximas perguntas serão mostradas e se referirão a uma visualização concreta da evolução de uma release do projeto Jetty/VRaptor. Use versões recentes do Google Chrome ou Mozilla Firefox para abrir os links das visualizações da ferramenta.



Questão 12. *Considerando a visualização "Sumarização de Cenários" da ferramenta acessada através do link <link_para_a_ferramenta>, você consegue identificar qual cenário (isto é, caso de teste) possui maior desvio de desempenho (degradação ou melhoria do tempo de execução/tempo de resposta) dentre os exibidos? O cenário foi otimizado ou degradado? (caixa de texto)*

Questão 13. *Quão fácil foi responder à pergunta anterior? (múltipla escolha - escala Likert de 5 itens)*

Questão 14. Considerando a visualização "Sumarização de Cenários" da ferramenta acessada através do link <link_para_a_ferramenta>, você consegue identificar qual cenário possui maior tempo de execução/resposta dentre os exibidos? O cenário foi otimizado ou degradado? (caixa de texto)

Questão 15. Quão fácil foi responder à pergunta anterior? (múltipla escolha - escala Likert de 5 itens)

Questão 16. Você poderia mencionar aspectos dessa visualização que você gostou? E quais outros aspectos você não gostou? Você tem alguma sugestão de melhoria ou comentário para essa visualização? (caixa de texto)

Página 5/5 - Geral

Questão 17. Você vê benefícios de usar a ferramenta de visualização de desvios de desempenho apresentada? Se sim, quais? (caixa de texto)

Questão 18. Você utilizaria a ferramenta como parte integrante do processo de desenvolvimento de software do Jetty/VRaptor? Se sim, como você vislumbra que ela seria utilizada? (caixa de texto)

Questão 19. Utilize o espaço abaixo para incluir comentários adicionais que deseje. (caixa de texto)

Questão 20. Você está disponível para ser contactado para discutir nossos resultados relacionados ao projeto Jetty/VRaptor? Se sim, por favor deixe o seu e-mail abaixo. (caixa de texto)

Você pode acessar e verificar todos os resultados gerados pela nossa ferramenta através deste link: <http://apvis.herokuapp.com/>