



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

Performance QA Evolution: uma ferramenta para análise da evolução do atributo de qualidade de desempenho em sistemas de software

Leo Moreira Silva

Natal-RN
Junho de 2017

Leo Moreira Silva

Performance QA Evolution: 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

Junho de 2017

Performance QA Evolution: 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

As aplicações estão cada vez mais inseridas na sociedade mediante vários dispositivos. Muitas delas são complexas devido a sua larga escala, tornando a sua arquitetura também complexa para entender e manter. A área de visualização de software lança mão de técnicas cujo objetivo é melhorar o entendimento do software e tornar mais produtivo o seu processo de desenvolvimento. A inexistência de atividades para que os desenvolvedores e arquitetos possam entender a evolução arquitetural pode levar a sua degradação, fazendo com que os atributos de qualidade inicialmente definidos deixem de ser atendidos. Nesse sentido, em relação a medição do desempenho de aplicações, ferramentas de *profiling* e APM podem ser utilizadas, entretanto, falham ao prover métricas e visualizações adequadas para o acompanhamento da evolução desse atributo de qualidade. Este trabalho apresenta um conjunto de visualizações de software com o intuito de auxiliar a análise da evolução do desempenho entre versões de determinado software, possibilitando que desenvolvedores e arquitetos identifiquem métodos dos cenários que degradaram ou melhoraram o seu desempenho. O conjunto de visualizações proposto será avaliado a partir de estudos empíricos realizados em aplicações *open source*.

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

Performance QA Evolution: 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

Applications are increasingly inserted into society through various devices. Many of them are complex due to their scale, making their architecture also complex to understand and maintain. The software visualization area uses techniques that aim to improve software understanding and make its development process more productive. The lack of activities for developers and architects to understand the architectural evolution can lead to its degradation, causing the quality criteria initially set no longer met. In this sense, in relation to measuring the performance of applications, profiling tools and APM can be used, however, fail to provide adequate metrics and visualizations to monitor the evolution of this quality attribute. This work presents a set of software visualizations to help analyze the evolution of performance between versions of a software, allowing developers and architects to identify methods of scenarios that have degraded or improved their performance. The proposed set of visualizations will be evaluated from empirical studies conducted in open source applications.

Keywords: software architecture, software visualization, software evolution.

Lista de figuras

1	EvoSpaces.	p. 25
2	Areas of Interest.	p. 25
3	CrocoCosmos.	p. 26
4	CodeCity.	p. 27
5	Fase 1 do PerfMiner.	p. 33
6	Fase 2 do PerfMiner.	p. 35
7	Fase 3 do PerfMiner.	p. 36
8	Abordagem completa do PerfMiner.	p. 37
9	Fase 4 do PerfMiner.	p. 37
10	Página inicial da aplicação.	p. 39
11	Página da funcionalidade de Nova Análise.	p. 40
12	Funcionamento do processamento de uma nova análise.	p. 41
13	Passos 2 e 3 da realização de uma nova análise.	p. 42
14	Diagrama de classes parcial do <i>PerfMiner</i>	p. 44
15	Diagrama de classes da extensão proposta.	p. 45
16	Visão geral da Sumarização de Cenários.	p. 47
17	<i>Tooltip</i> com maiores informações sobre determinado cenário.	p. 47
18	Funcionamento geral das visualizações.	p. 48
19	Detalhamento do passo 4 da figura 18, na Sumarização de Cenários. . .	p. 49
20	Exemplo do grafo de chamadas.	p. 50
21	Seção de sumário do grafo de chamadas.	p. 51
22	Seção de histórico do grafo de chamadas.	p. 53

23	Nó que representa um método sem desvio de desempenho.	p. 53
24	Nó que representa um método com degradação de desempenho.	p. 54
25	Nó que representa um método com otimização de desempenho.	p. 55
26	Nó que representa um método adicionado.	p. 55
27	Nó que representa um método removido.	p. 56
28	Nó que representa um agrupamento de outros nós.	p. 56
29	Borda espessa de um nó, representando múltiplas execuções.	p. 57
30	Barra de ferramentas do grafo de chamadas.	p. 58
31	Efeito de destaque do caminho de execução de nós com desvio.	p. 58
32	Legenda do grafo de chamadas.	p. 59
33	Detalhes de um nó no grafo de chamadas.	p. 60
34	Detalhamento do passo 4 da figura 18, no Grafo de Chamadas.	p. 61
35	Grafo de chamadas de um cenário com degradação de desempenho. . .	p. 63
36	Detalhes de um cenário com degradação de desempenho.	p. 64
37	Grafo de chamadas de um cenário com otimização de desempenho. . .	p. 64
38	Detalhes de um cenário com otimização de desempenho.	p. 65
39	Dados demográficos dos participantes.	p. 74
40	Exemplos da visualização de Sumarização de Cenários.	p. 77
41	Exemplos da visualização de Sumarização de Cenários com excesso de cenários.	p. 78
42	Exemplo de nós agrupados durante a hierarquia de chamadas.	p. 82
43	Exemplo de nós agrupados em hierarquias adjacentes (A) e em nós filhos (B).	p. 82
44	Gráficos <i>boxplot</i> sobre o algoritmo de redução de nós.	p. 83
45	Grafo de Chamadas do cenário C3.	p. 85
46	Grafo de Chamadas do cenário C4.	p. 85
47	Grafo de Chamadas do cenário C5.	p. 86

48	Grafo de Chamadas do cenário C7.	p. 87
49	Grafo de Chamadas do cenário C8.	p. 87
50	Questões sobre os benefícios e utilidades da ferramenta proposta. . . .	p. 89
51	Cronograma de atividades.	p. 94

Lista de tabelas

1	Resumo das características visuais dos nós.	p. 57
2	Organização dos pares de releases para o Jetty e VRaptor.	p. 71
3	Distribuição dos grupos de participantes.	p. 73
4	Resumo para o Jetty.	p. 76
5	Resumo para o VRaptor.	p. 76
6	Características dos cenários exemplos de casos especiais.	p. 84

Lista de abreviaturas e siglas

TCAS – *Traffic Collision Avoidance System*

ABS – *Anti-Lock Break System*

UTI – Unidade de Terapira Intensiva

JDK – *Java Development Kit*

JVM – *Java Virtual Machine*

APM – *Application Performance Management*

CCET – Centro de Ciências Exatas e da Terra

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*

Sumário

1	Introdução	p. 13
1.1	Apresentação do Problema	p. 15
1.2	Limitações das Abordagens Atuais	p. 15
1.3	Abordagem Proposta	p. 18
1.4	Objetivos Gerais e Específicos	p. 19
1.5	Organização do trabalho	p. 20
2	Fundamentação Teórica	p. 21
2.1	Arquitetura de Software	p. 21
2.1.1	Atributos de Qualidade	p. 22
2.1.2	Avaliação Baseada em Cenários	p. 22
2.2	Visualização de Software	p. 23
2.2.1	Visualização de Arquitetura de Software	p. 24
2.2.2	Visualização da Evolução da Arquitetura de Software	p. 26
2.2.3	Estudos Empíricos em Visualização de Software fica ou sai?	p. 27
2.3	Ferramentas de Análise de Desempenho	p. 29
2.4	Considerações	p. 31
3	Performance QA Evolution	p. 32
3.1	PerfMiner	p. 32
3.1.1	Funcionamento	p. 33
3.1.1.1	Fase 1: Análise Dinâmica	p. 33

3.1.1.2	Fase 2: Análise de Desvio	p. 34
3.1.1.3	Fase 3: Mineração de Repositório	p. 35
3.2	Visão Geral	p. 38
3.2.1	Nova Análise	p. 39
3.2.1.1	Funcionamento	p. 40
3.2.1.2	Bancos de Dados	p. 43
3.2.1.2.1	DAM - <i>Dynamic Analysis Model</i>	p. 43
3.2.1.2.2	QAV - <i>Quality Attribute Visualization</i>	p. 44
3.3	Visualização da Sumarização de Cenários	p. 45
3.3.1	Interação	p. 47
3.3.2	Funcionamento	p. 48
3.4	Visualização do Grafo de Chamadas	p. 49
3.4.1	Sumário	p. 50
3.4.2	Histórico	p. 52
3.4.3	Grafo de Chamadas	p. 53
3.4.3.1	Nó sem desvio	p. 53
3.4.3.2	Nó degradado	p. 54
3.4.3.3	Nó otimizado	p. 55
3.4.3.4	Nó adicionado	p. 55
3.4.3.5	Nó removido	p. 56
3.4.3.6	Nó de agrupamento	p. 56
3.4.4	Interação	p. 57
3.4.5	Funcionamento	p. 60
3.4.6	Exemplo de Uso da Ferramenta: Jetty fica ou sai?	p. 62
3.5	Considerações	p. 65

4.1	Projeto	p. 67
4.1.1	Principais Contribuições	p. 68
4.1.2	Objetivos e Questões de Pesquisa	p. 68
4.1.3	Sistemas	p. 69
4.1.4	Procedimentos	p. 70
4.1.4.1	Passo 1 - Coleta e Preparação dos Dados	p. 70
4.1.4.2	Passo 2 - Aplicação da Abordagem	p. 70
4.1.4.3	Passo 3 - Elaboração e Aplicação dos Questionários . .	p. 72
4.1.5	Caracterização dos Participantes	p. 73
4.2	Resultados	p. 75
4.2.1	Análise do Comportamento das Visualizações	p. 75
4.2.1.1	Sumarização de Cenários	p. 75
4.2.1.2	Grafo de Chamadas	p. 80
4.2.1.2.1	Supressão de Nós Exibidos	p. 80
4.2.1.2.2	Casos Especiais	p. 83
4.2.2	Questionário Online	p. 87
4.3	Considerações	p. 90
5	Trabalhos Relacionados	p. 91
5.1	Ferramentas de Profiling	p. 91
5.2	Ferramentas APM	p. 92
5.3	Abordagens de Degradação de Desempenho	p. 93
6	Conclusão	p. 94
6.1	Cronograma de Atividades	p. 94
6.2	Limitações	p. 95
	Referências	p. 97

Apêndice A – Questionário p. 103

A.1 Tipo 1 p. 103

A.2 Tipo 2 p. 106

1 Introdução

Quando um avião decola do solo a partir de uma cidade de origem para uma cidade de destino, vários softwares presentes na aeronave auxiliam a tripulação durante o trajeto. Um deles é o TCAS II (*Traffic Collision Avoidance System*), um sistema anticolisão que, através de dados obtidos das aeronaves presentes nas imediações, calcula o nível de ameaça de cada uma delas. O software, inclusive, pode realizar manobras alternativas se o risco de colisão for iminente [1].

Existem também softwares para automóveis. Destes, pode ser citado o ABS (*Anti-Lock Break System*), que calcula a maior força de frenagem possível a ser aplicada nas rodas, no entanto, sem deixa-las travar. O ABS melhora a eficiência dos freios, reduzindo a distância de parada do automóvel e evitando que a dirigibilidade seja perdida durante a frenagem [2].

Além de estar presente em aeronaves e automóveis, os softwares também podem ser encontrados em televisores, celulares, tablets, computadores, *smart* TVs e em outros dispositivos. Ademais, os softwares estão presentes nas mais diversas áreas, como medicina, educação física e direito. Tais exemplos reforçam e justificam a importância que as aplicações têm, estando atualmente presentes no dia a dia da população, nas mais variadas atividades.

Os sistemas podem ser desde os mais simples, como aplicativos de calculadora para smartphones, até mais complexos, como sistemas de tempo real que monitoram pacientes em uma Unidade de Terapia Intensiva (UTI) de um hospital. Todos eles, no entanto, possuem uma arquitetura que serve como base para o seu desenvolvimento.

Como parte integrante da descrição de uma arquitetura, os softwares podem possuir requisitos de qualidade críticos. O TCAS II, por exemplo, requer que o cálculo das ameaças seja realizado com o maior *desempenho* possível. Por sua vez, o ABS necessita que o cálculo da força de frenagem aplicada nas rodas tenha um alto grau de *confiabilidade*, ou seja, quanto mais os freios funcionarem em relação ao número de tentativas, mais confiável ele

será. Além de desempenho e confiabilidade, outros exemplos de requisitos de qualidade comuns são segurança, portabilidade, robustez e escalabilidade.

As áreas de arquitetura dos softwares e de requisitos de qualidade são bem abordadas pela disciplina de Engenharia de Software. Não obstante, apesar dessa adequada abordagem, há um entendimento ainda deficiente devido a intrínseca característica que difere os programas de computador da maioria dos produtos industriais: um software é um artefato inerentemente intelectual. É invisível, imaterial [3]. Como menciona Ball e Eick [4], software é intangível, não tem forma física ou tamanho. Depois que é escrito, o código “desaparece” em arquivos armazenados em discos.

Presentes nos mais variados domínios, os softwares vêm sendo utilizados por usuários com diferentes características e executando em dispositivos distintos, fazendo com que seja alta a demanda por novas funcionalidades e tecnologias ou para reparação de erros. Caserta e Zendra [5] 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 [6]. 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.

Nesse contexto, a Visualização de Software é uma área da Engenharia de Software que objetiva usar representações visuais para melhorar o entendimento e compreensão de diferentes aspectos de um sistema de software. Alguns desses aspectos incluem 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 [7].

Este capítulo introduz este trabalho. A seção 1.1 exhibe o contexto do trabalho e apresenta o problema a ser tratado. A seção 1.2 apresenta as principais limitações das abordagens atuais. A seção 1.3 apresenta a abordagem proposta, ao passo que a seção 1.4 discute os objetivos gerais e específicos. Por fim, a organização do trabalho é mostrada na seção 1.5.

1.1 Apresentação do Problema

Com o intuito de tornar o software mais compreensível, a área de Visualização de Software lança mão de representações visuais para essa finalidade. 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 [8]. Essa área pode focar em vários aspectos de sistemas de software, como mencionado anteriormente, entretanto, um dos componentes essenciais é a visualização da arquitetura de um software.

Ghanam e Carpendale [7] 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.

No entanto, a visualização da arquitetura de um software 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 [5][9].

Durante o processo de manutenção, 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 [10]. Nesse contexto, a inexistência de atividades para que os desenvolvedores possam entender a evolução arquitetural pode levar a sua degradação [11], fazendo com que os atributos de qualidade inicialmente definidos, a partir de decisões arquiteturais e de *design* tomadas durante o processo de desenvolvimento, deixem de ser 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, de modo a diminuir a probabilidade de ocorrer a degradação desse atributo de qualidade durante o processo de manutenção.

1.2 Limitações das Abordagens Atuais

De acordo com a literatura, existem várias ferramentas que tratam da visualização da evolução arquitetural de softwares, inclusive com monitoramento de desempenho, cada uma com suas particularidades. Caserta e Zendra [5] 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 está relacionado com o último item dessa relação, especificamente com 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* [12], 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 entre eles ou de versões anteriores do software, tornando difícil a visualização da evolução do atributo de qualidade desempenho, uma vez que teria que ser feita manualmente para cada método desejado.

Já o *JProfiler* [13], 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. 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* [14] 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.[10] menciona que essas duas ferramentas, apesar de serem úteis para acompanhar o desempenho geral, saber 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.[15] 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 [15]. As ferramentas utilizadas no estudo foram *New Relic* [16], *AppDynamics* [17], *Dynatrace* [18] e *Pinpoint* [19]. 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 da arquitetura, 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 visualizações, outros requisitos de uma abordagem para análise de desvios de desempenho na evolução de sistemas são mencionados por Pinto [20]:

- 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.

A análise arquitetural dos atributos de qualidade pode ser feita baseada em cenários, conforme mencionados acima, no terceiro ponto. Nesse contexto, um cenário é definido como uma ação de alto nível que representa a maneira como os *stakeholders* interagem com o sistema.

1.3 Abordagem Proposta

Este trabalho apresenta uma ferramenta, chamada *Performance QA Evolution*, 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 o método/cenário 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 [25]. 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.

A ferramenta foi implementada como extensão a outra já existente, chamada *Perf-Miner* [20]. 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 do cenário que possuiu o maior 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.

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 aprimorar 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 *Automated Software Engineering*, do Centro de Ciências Exatas e da Terra (CCET) da UFRN.

A ferramenta estendida, chamada de *PerfMiner* [20], 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 questões de desenvolvimento responsáveis por alterá-los. Técnicas de análise dinâmica e mineração de dados são usadas pela ferramenta para atingir os seus objetivos. Estendê-la, adicionando visualizações, irá permitir aos usuários identificar adequadamente os cenários e métodos com desvio de desempenho ocorridos durante a evolução do sistema. As visualizações podem apontar os pontos onde o software degradou e, a partir de então, os usuários podem estabelecer formas de otimizar o desempenho da aplicação. Nesse contexto, os objetivos específicos deste trabalho são:

- Pesquisar as principais abordagens de visualização da evolução de arquiteturas de software, em especial 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 *Performance QA Evolution* como extensão ao *PerfMiner*. 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;
- Realizar estudos qualitativos com desenvolvedores e arquitetos de sistemas reais de diferentes domínios para avaliar a utilidade das visualizações, a facilidade de se encontrar as informações dispostas 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 reúne os principais conceitos necessários para o entendimento do trabalho, tais como arquitetura de software, visualização de software e ferramentas de análise de desempenho. O capítulo ?? 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 explica o estudo empírico a ser conduzido para avaliação do trabalho. O capítulo 5 exhibe trabalhos relacionados, mencionando as suas limitações e comparando-os com este trabalho. Finalmente, o capítulo 6 mostra o cronograma planejado para a conclusão do trabalho e discute as limitações percebidas até o momento. **VERIFICAR A ESTRUTURA DEPOIS...**

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 [21].

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 [22]. Adicionalmente, de acordo com [Taylor, Medvidovic e Dashofy](#)[21], 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 [23].

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 [24].

A qualidade não pode ser adicionada ao sistema de maneira tardia, pelo contrário, deve ser incorporada desde o início [24]. 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 [23]:

- *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 [26]. 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 [20].

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 [27]. Ainda de acordo com Diehl[27], 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 [8]. Tais dificuldades são ainda piores em sistemas de grande escala.

Existem três tipos de aspectos do software que a visualização pode abordar [27]:

- (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, Gómez Henriquez [28] 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 [29]. Diehl[27] 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 [7][30][31][32]. 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 [5]. 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 [33].

As representações visuais mais comuns para visualizar a estrutura hierárquica da arquitetura de um software é uma estrutura em árvore [9]. Exemplos de outras formas de se representar a arquitetura são o *Treemap* retangular [34] e circular [35] e o *Icicle Plot* [36]. Já para a visualização dos relacionamentos, podem ser mencionados: *Dependency Structure Matrix* [37], *DA4Java* [38], *EvoSpaces* [39] e *Clustered Graph Layout* [40]. Para a visualização das métricas de um software, são exemplos as soluções: *Metric View* [41], *Areas of Interest* [42] e *CodeCrawler* [43].

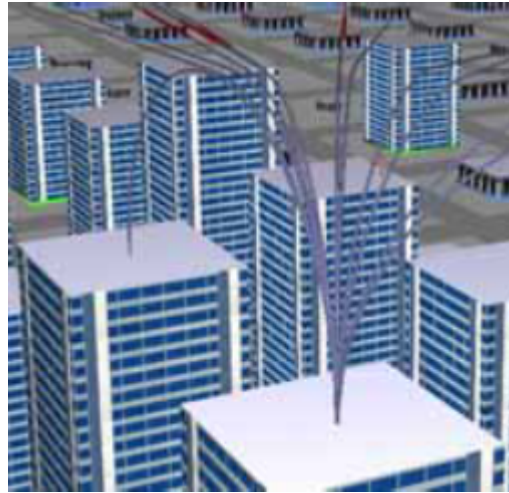


Figura 1: EvoSpaces [39].

O EvoSpaces [39], 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.

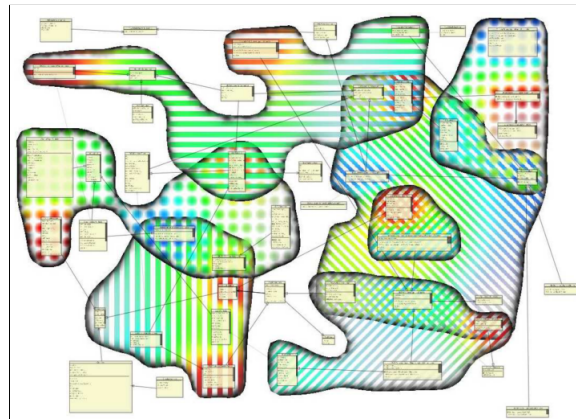


Figura 2: Areas of Interest [42].

O Areas of Interest [42], 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 [44]. 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 [45]. 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 [9].

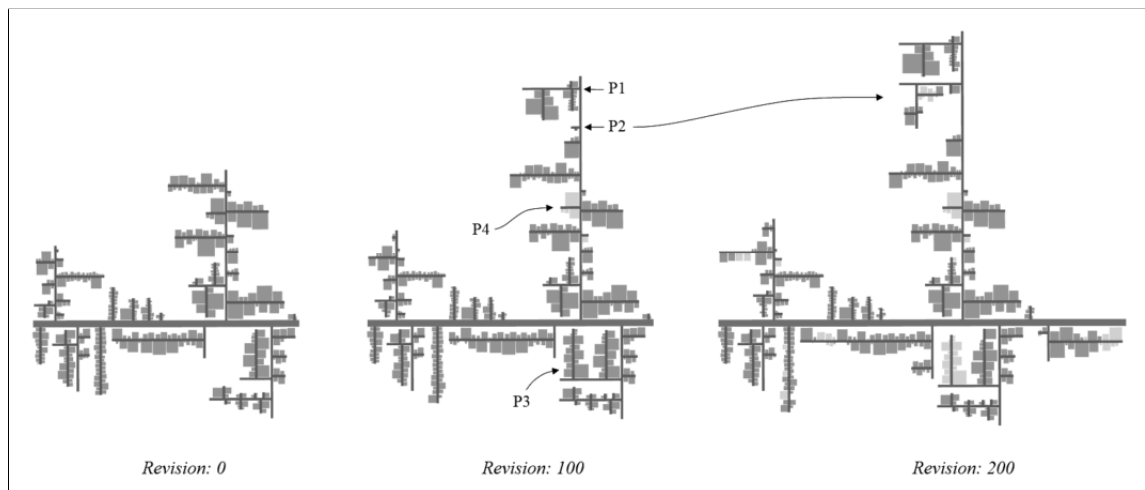


Figura 3: CrocoCosmos [47].

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 provêem 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 [9].

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.

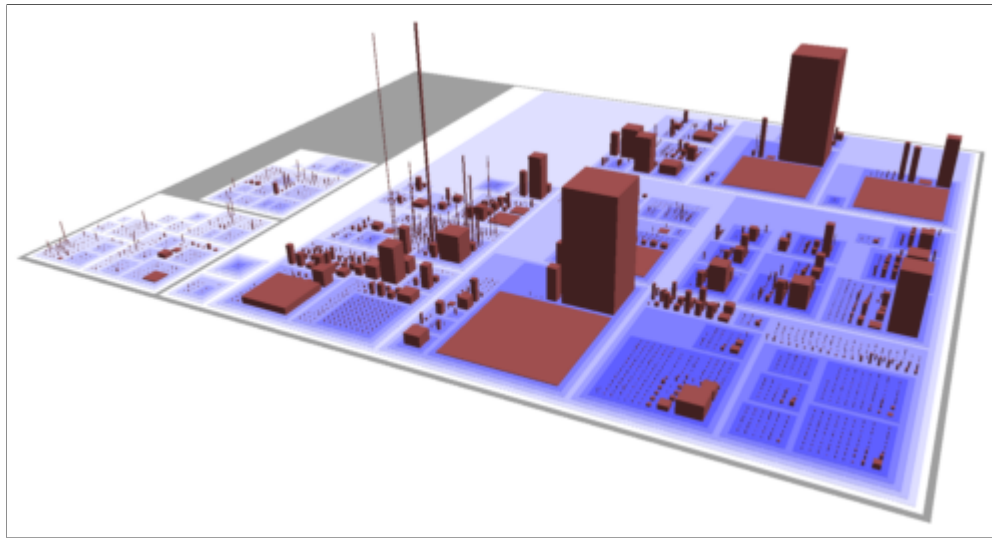


Figura 4: Codecity [51].

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 **fica ou sai?**

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 [27]. O objetivo principal de uma visualização é transmitir informações de forma compreensível, eficaz e fácil de lembrar. Diehl[27] agrupa as avaliações das visualizações em dois grupos: quantitativa e qualitativa.

Os métodos quantitativos de avaliação medem propriedades da visualização ou propriedades do usuário interagindo com a visualização. Esse tipo de avaliação requer uma

análise estatística dos resultados de um experimento controlado [27].

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* [12]: 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* [13]: 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* [14]: 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 [15]. Exemplos desse tipo de ferramentas são o *New Relic* [16], *AppDynamics* [17], *Dynatrace* [18] e *Pinpoint* [19].

2.4 Considerações

Os conceitos apresentados neste capítulo são importantes para o entendimento da proposta da seguinte forma. Com relação ao conceito de arquitetura de software, a ferramenta estendida, *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 extensão proposta utiliza representações visuais para exibir aspectos dinâmicos e de evolução da arquitetura de software. Dessa forma, é exibido o atributo de qualidade de desempenho relacionado a arquitetura do software, além de sua evolução 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 extensão.

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 extensão proposta ao *PerfMiner*, 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 Performance QA Evolution

rever texto a seguir...

para cada visualização, comentar quais seriam os benefícios e o que os desenvolvedores podem tirar de informações delas...

Este capítulo apresenta o conjunto de visualizações proposto como extensão à ferramenta *PerfMiner*. A seção 3.1 destaca o funcionamento dessa ferramenta. A seção ?? mostra uma visão geral sobre o funcionamento das visualizações propostas. As seções ??, 3.4 e ?? descrevem cada uma das visualizações, sendo esta última em detalhes, com suas características, propriedades visuais, funcionamento e exemplo de uso. Por fim, são reportadas considerações finais sobre o capítulo na seção 3.5.

3.1 PerfMiner

A implementação das visualizações está sendo desenvolvida como uma extensão da ferramenta *PerfMiner*, a qual provê a avaliação contínua de cenários no que diz respeito a análise de desvios de desempenho, de forma a minimizar a erosão de atributos de qualidade em cenários arquiteturalmente relevantes [20].

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 [20].

3.1.1 Funcionamento

Para atingir o objetivo de realizar a análise de desvio de desempenho, foram definidas três fases, descritas com maiores detalhes nas subseções a seguir: (i) análise dinâmica; (ii) análise de desvio; e (iii) mineração de repositório.

3.1.1.1 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 [20]. A figura 5 a seguir ilustra esta fase.

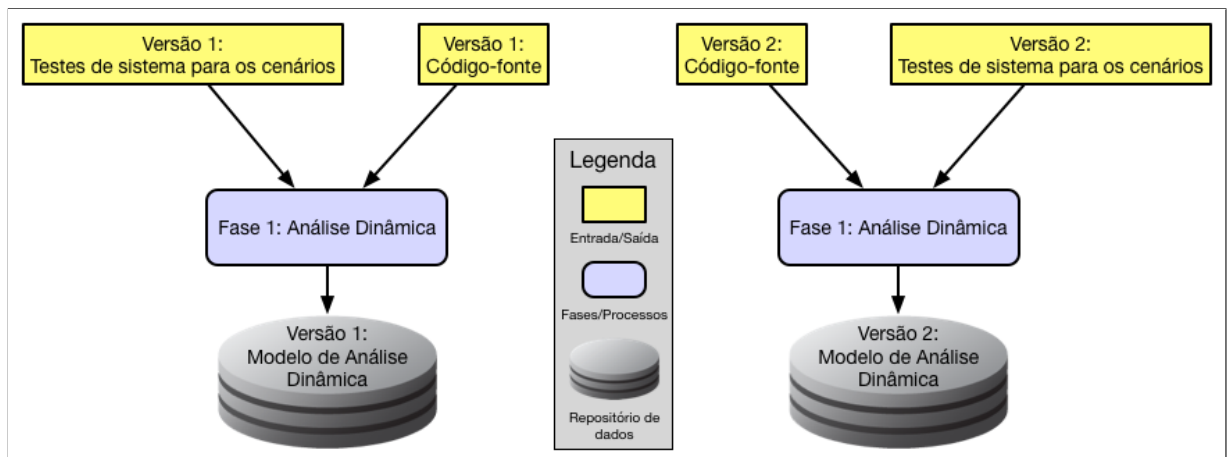


Figura 5: Fase 1 do PerfMiner.

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 [20].

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 [20].

De maneira sumária, o processo da figura 5 se inicia com a execução dos testes de sistema, que são, então, interceptados pelo *AspectJ* a fim de calcular seus tempos de execução. Após isso, o modelo de análise dinâmica resultante é persistido no banco de dados. 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.

3.1.1.2 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 [20]. Esta fase pode ser vista na figura 6 adiante.

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

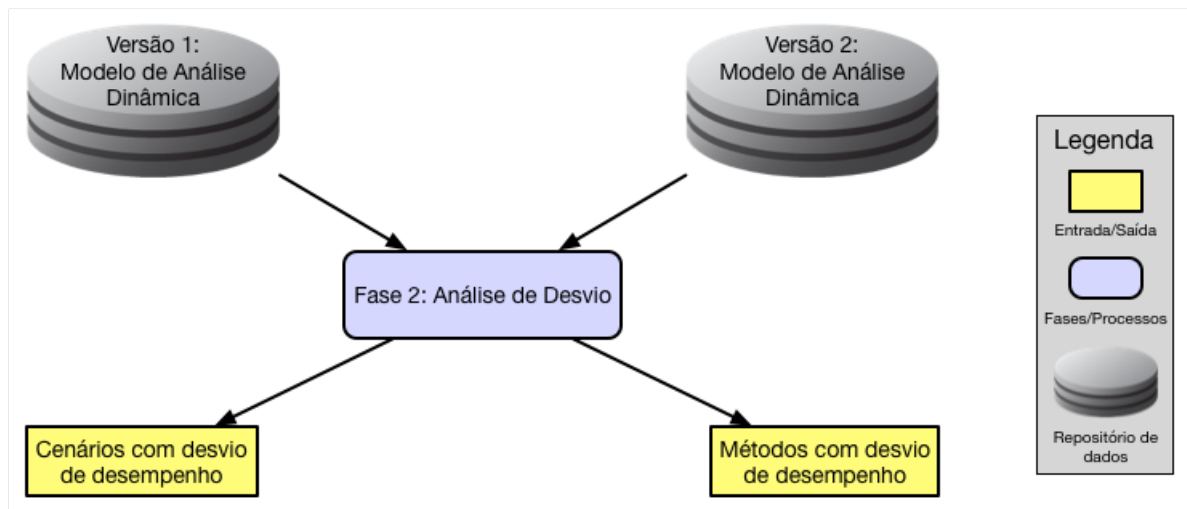


Figura 6: Fase 2 do PerfMiner.

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 [20].

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 diminuíram o tempo de execução [20].

3.1.1.3 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) [20]. A figura 7 ilustra essa fase.

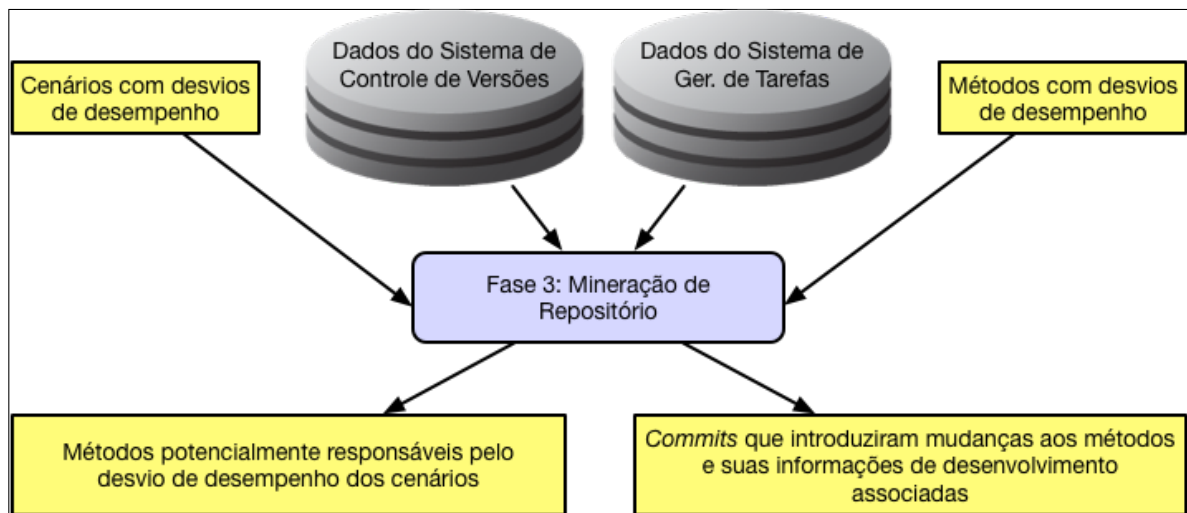


Figura 7: Fase 3 do PerfMiner.

É importante notar que os métodos identificados com desvios de desempenho nas fases anteriores, mas que não foram alterados durante a evolução, são também selecionados 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 [20].

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.

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



Figura 8: Abordagem completa do PerfMiner.

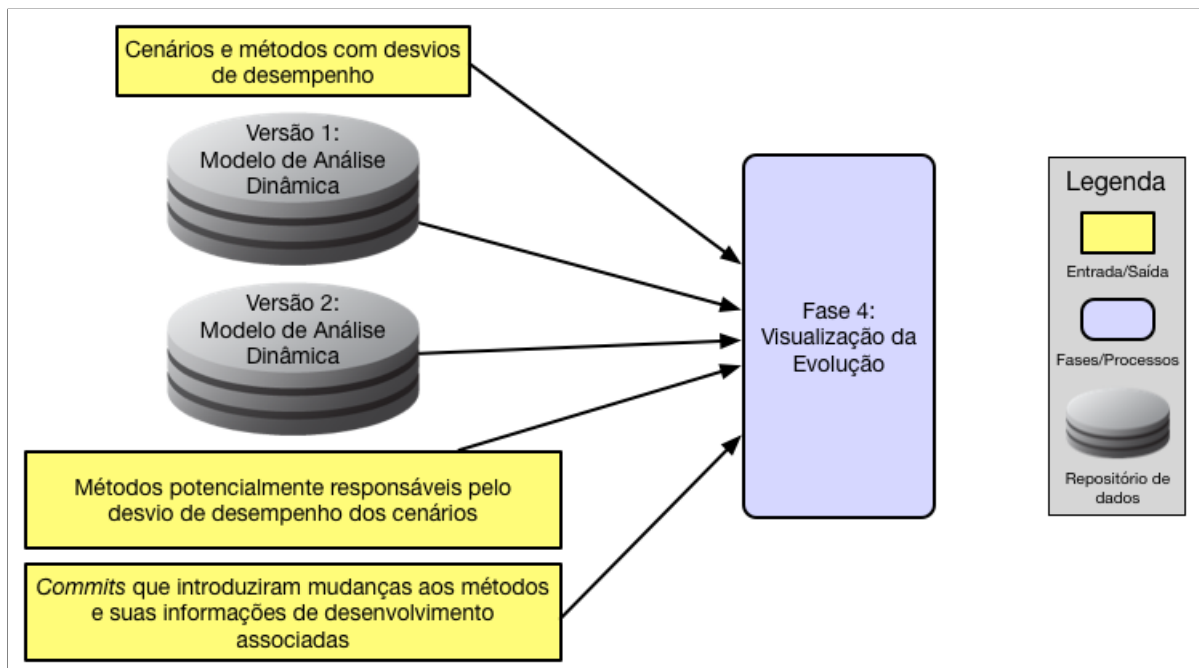


Figura 9: Fase 4 do PerfMiner.

3.2 Visão Geral



A extensão ao *PerfMiner* foi implementada na forma de uma aplicação web, intitulada de *Architecture QA Evolution*. 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 elaborado lançando mão do *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 **Performance QA Evolution** 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

¹<https://grails.org>

²<https://www.postgresql.org>

³<http://getbootstrap.com>

⁴<https://www.jointjs.com>

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.

PE

☰

☰

+

+

New Analysis

System A

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

System B


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

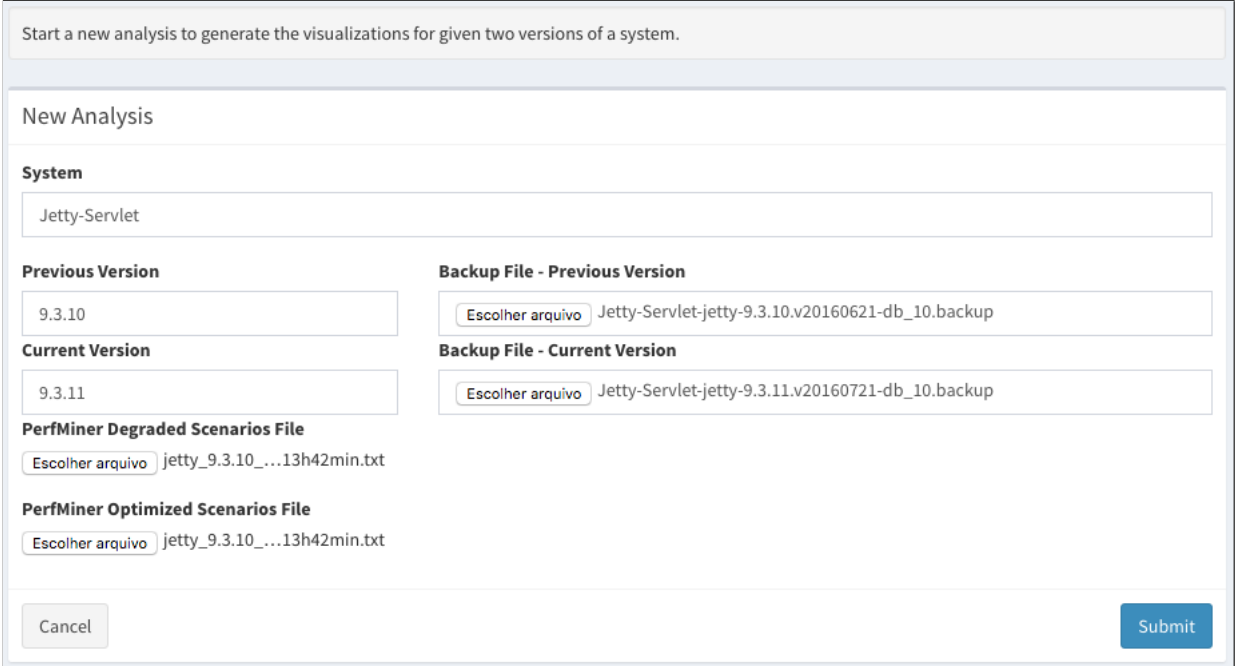
Figura 10: Página inicial da aplicação.

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,

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.



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

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

PerfMiner Degraded Scenarios File

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

PerfMiner Optimized Scenarios File

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

Cancel Submit

Figura 11: Página da funcionalidade de Nova Análise.

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**.

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.

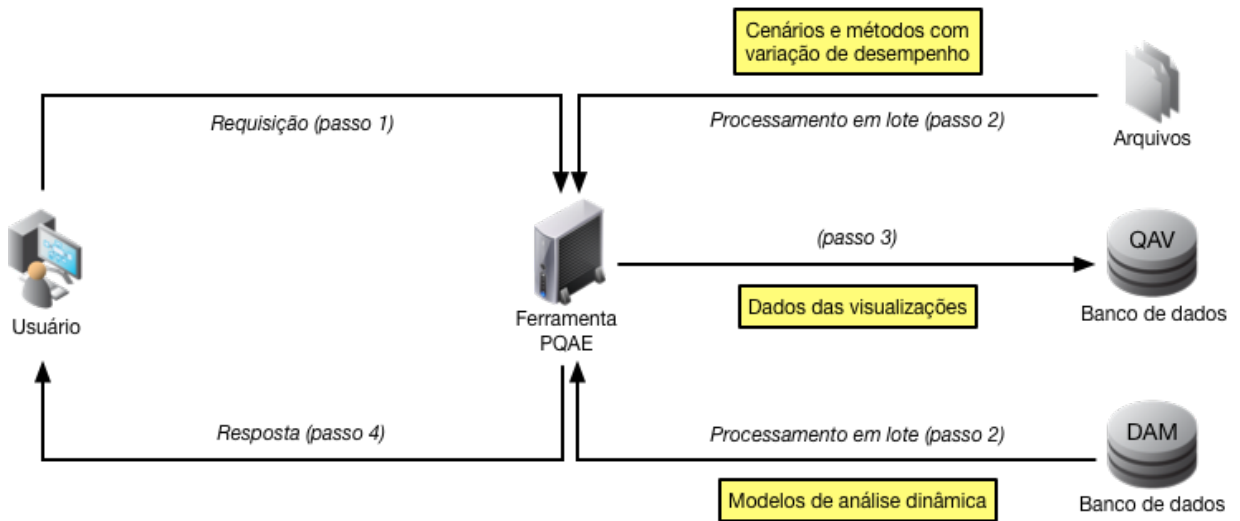


Figura 12: Funcionamento do processamento de uma nova análise.

Quando o servidor (Ferramenta PQAE) recebe essa requisição, inicia-se o processamento 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.

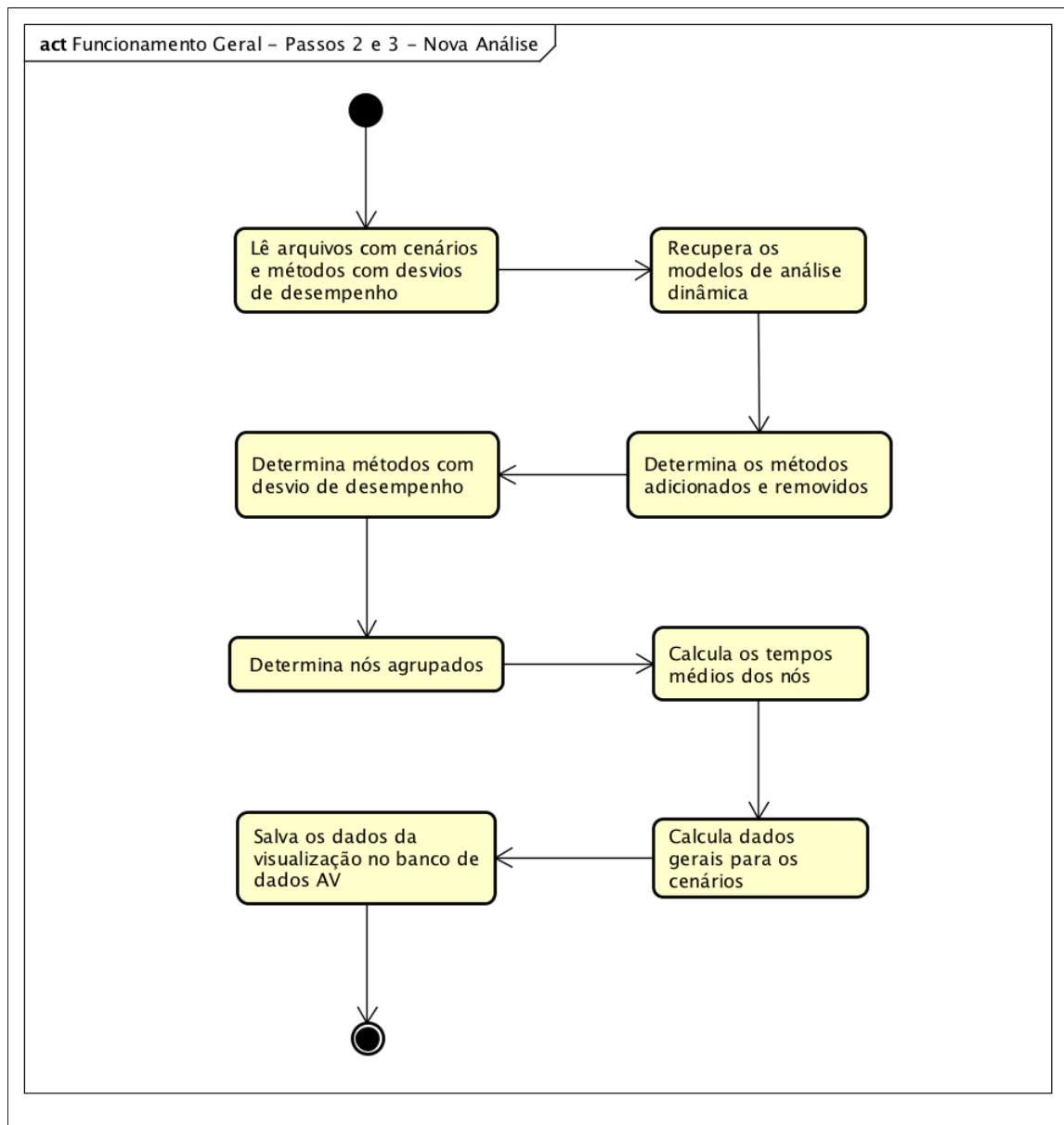


Figura 13: Passos 2 e 3 da realização de uma nova análise.

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 calculados 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 *Performance QA Evolution* e armazena os dados que dão suporte às visualizações, funcionando também como um *cache*, minimizando o tempo de resposta do servidor para o usuário.

3.2.1.2.1 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 3.1.1.1. 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 a seguir mostra o diagrama de classes parcial desse modelo.

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:

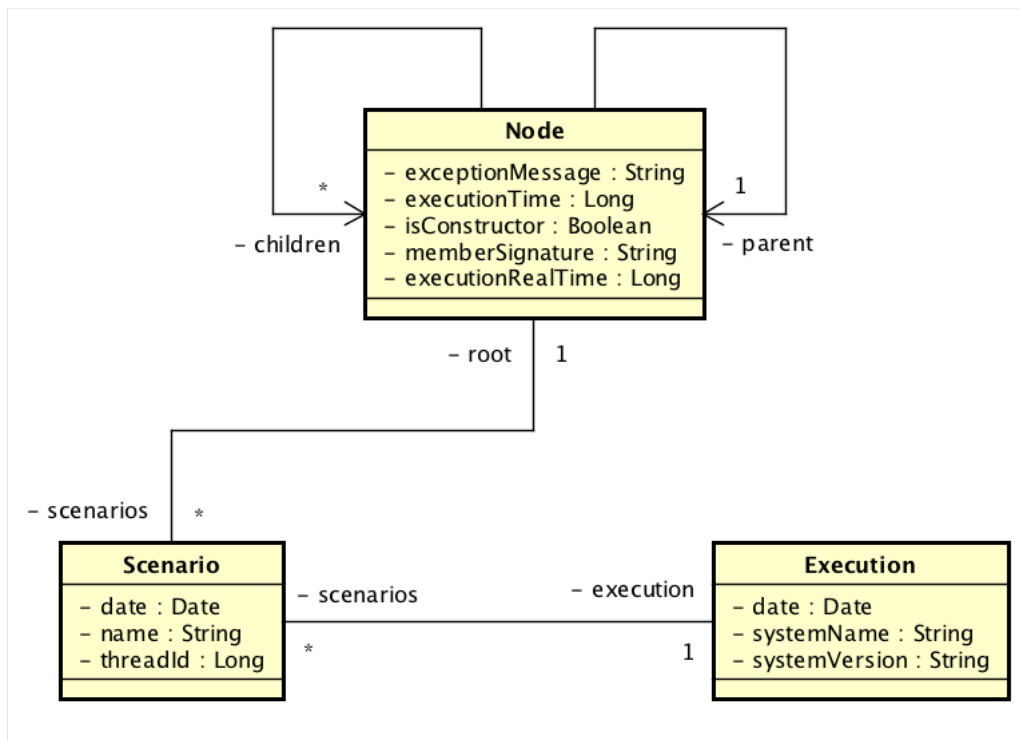


Figura 14: Diagrama de classes parcial do *PerfMiner*.

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.

3.2.1.2.2 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.

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

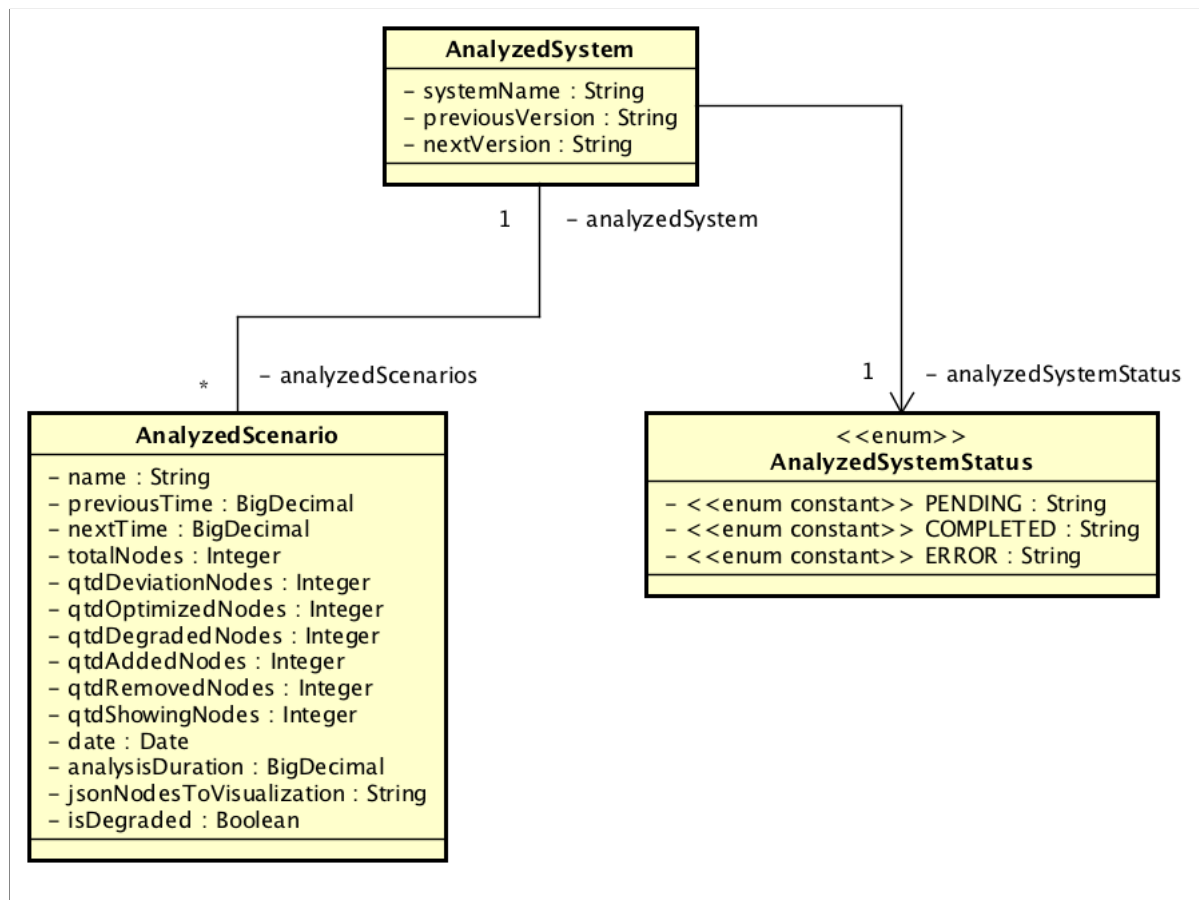


Figura 15: Diagrama de classes da extensão proposta.

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 a todas as 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.

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.



Figura 16: Visão geral da Sumarização de Cenários.

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.

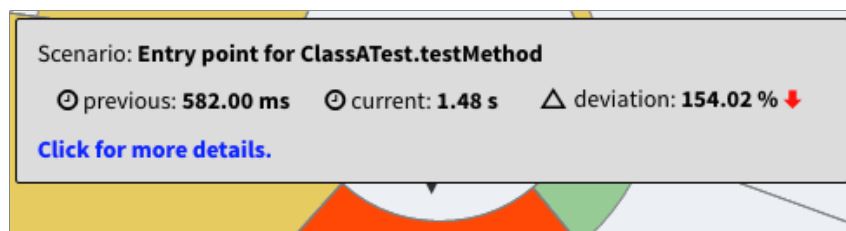


Figura 17: *Tooltip* com maiores informações sobre determinado cenário.

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.

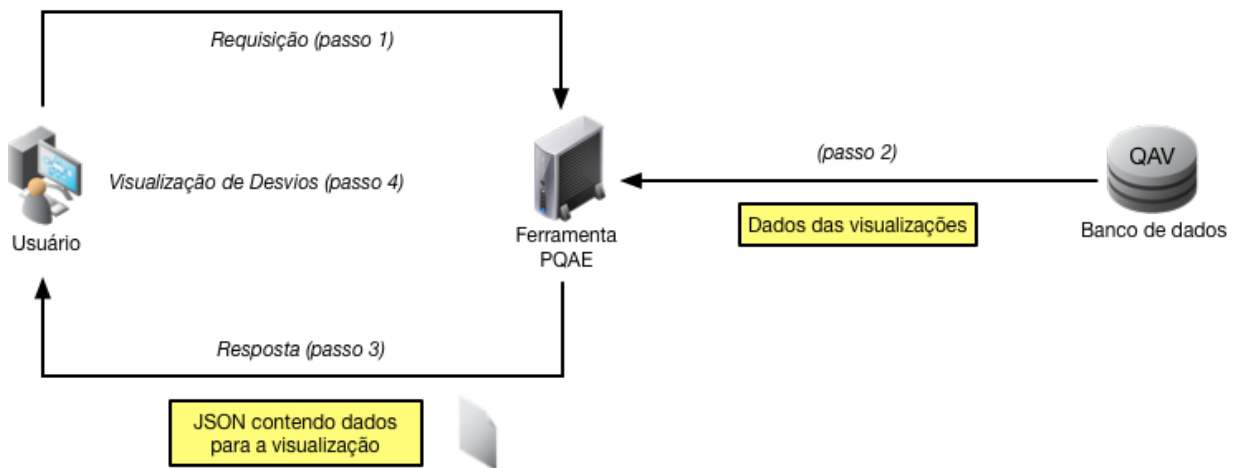


Figura 18: Funcionamento geral das visualizações.

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 (Ferramenta PQAE) 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 arquivo 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.

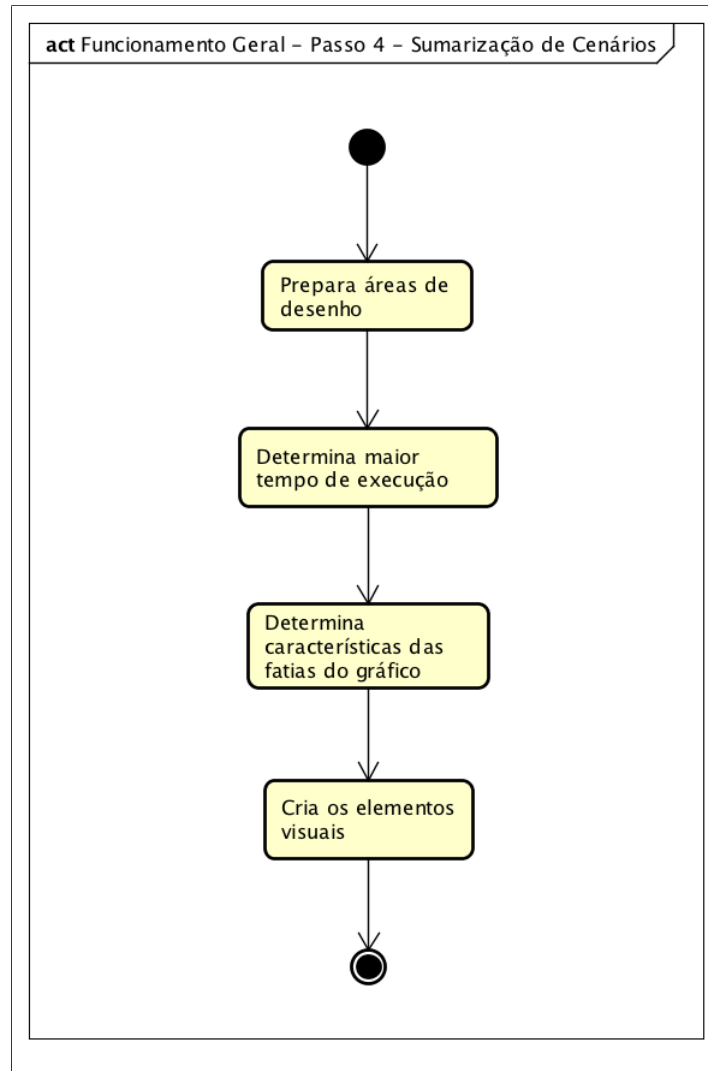


Figura 19: Detalhamento do passo 4 da figura 18, na Sumarização de Cenários.

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.

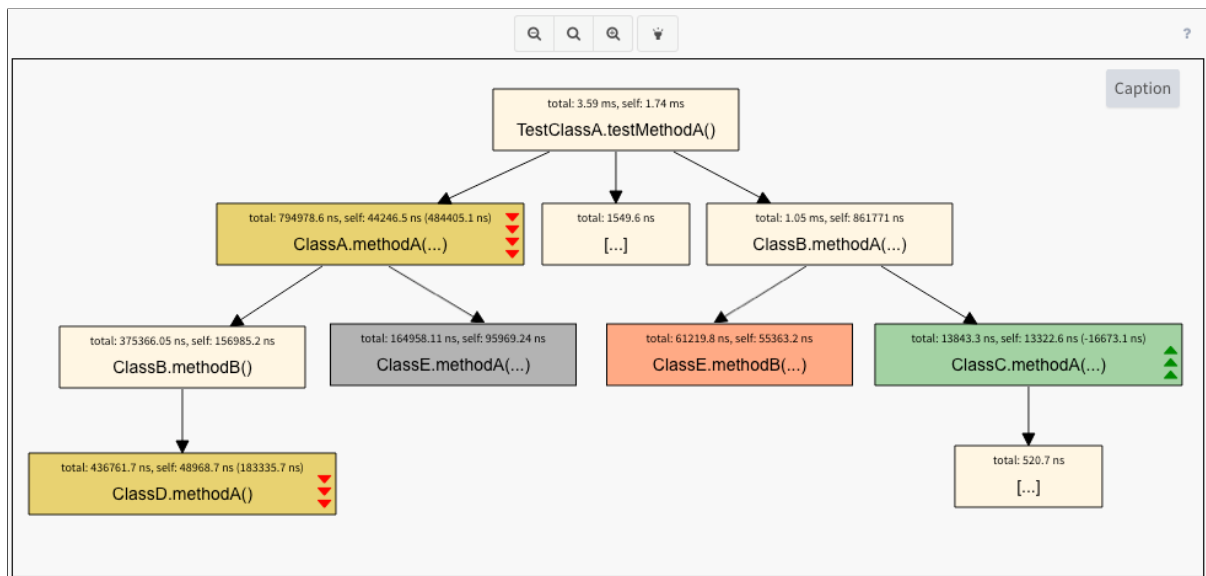


Figura 20: Exemplo do grafo de chamadas.

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;

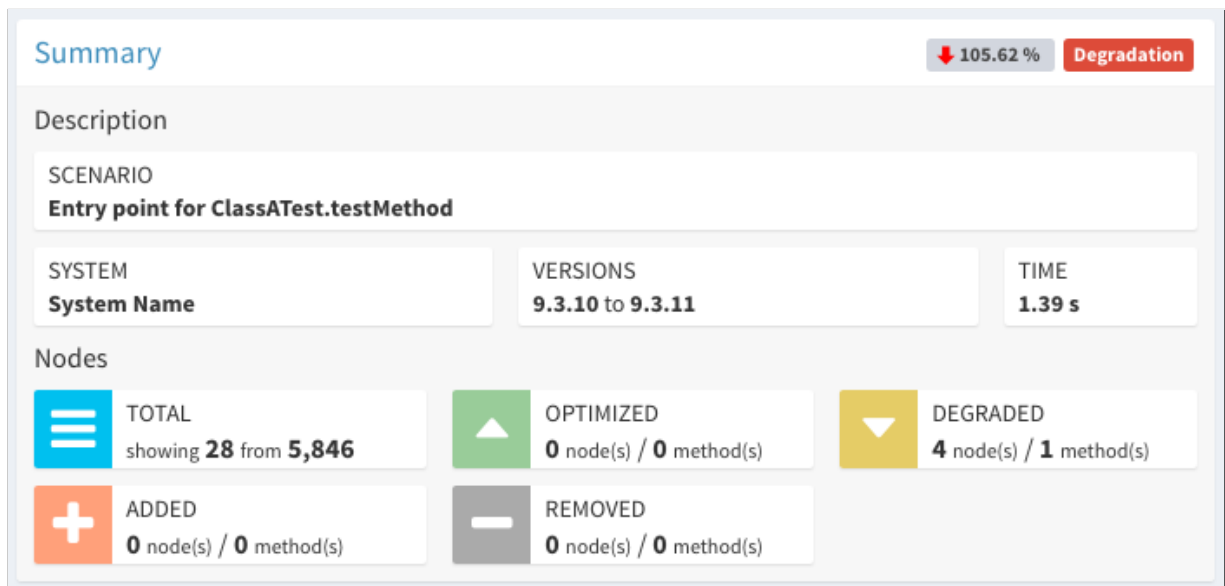


Figura 21: Seção de sumário do grafo de chamadas.

- *Versions*: versões do sistema que foram analisadas. É mostrado no formato $\langle versionA \rangle$ to $\langle versionB \rangle$. 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. 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*: exhibe 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.

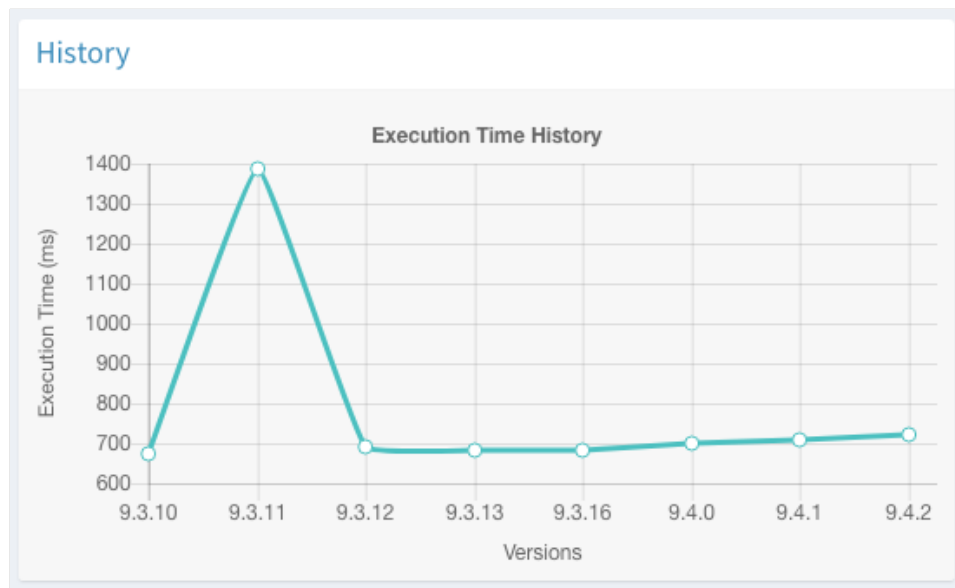


Figura 22: Seção de histórico do grafo de chamadas.

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

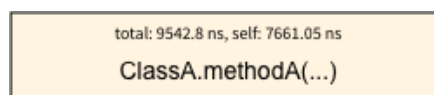


Figura 23: Nó que representa um método sem desvio de desempenho.

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:

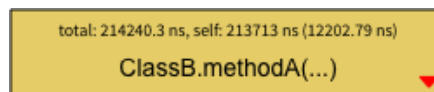


Figura 24: Nó que representa um método com degradação de desempenho.

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:

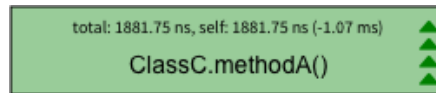


Figura 25: Nó que representa um método com otimização de desempenho.

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:

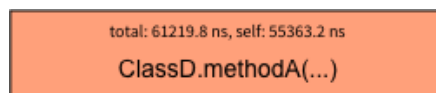


Figura 26: Nó que representa um método adicionado.

- *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:

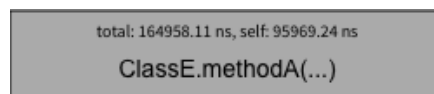


Figura 27: Nó que representa um método removido.

- *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 que não estão diretamente relacionados com nós de desvio, adicionados ou removidos, são omitidos e representados por um único nó agrupado. A figura 28 a seguir ilustra esse tipo de nó, seguido da descrição dos seus atributos visuais.

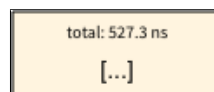


Figura 28: Nó que representa um agrupamento de outros nós.

- *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.

Cada nó no grafo de chamadas, da forma como foi apresentado até o momento, representa uma única execução deste. No entanto, um nó, para determinada hierarquia de chamadas, pode ser executado mais de uma vez (por exemplo, em um *loop*). Com o intuito de não acrescentar nós ao grafo para representar essas situações, foi estabelecido que se um nó possuir uma borda espessa, significa que este executou mais de uma vez naquela hierarquia. A figura 29 adiante exemplifica um nó com uma borda espessa.

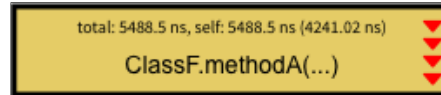


Figura 29: Borda espessa de um nó, representando múltiplas execuções.

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.

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 30. 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.

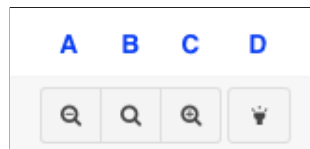


Figura 30: Barra de ferramentas do grafo de chamadas.

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 31. 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.

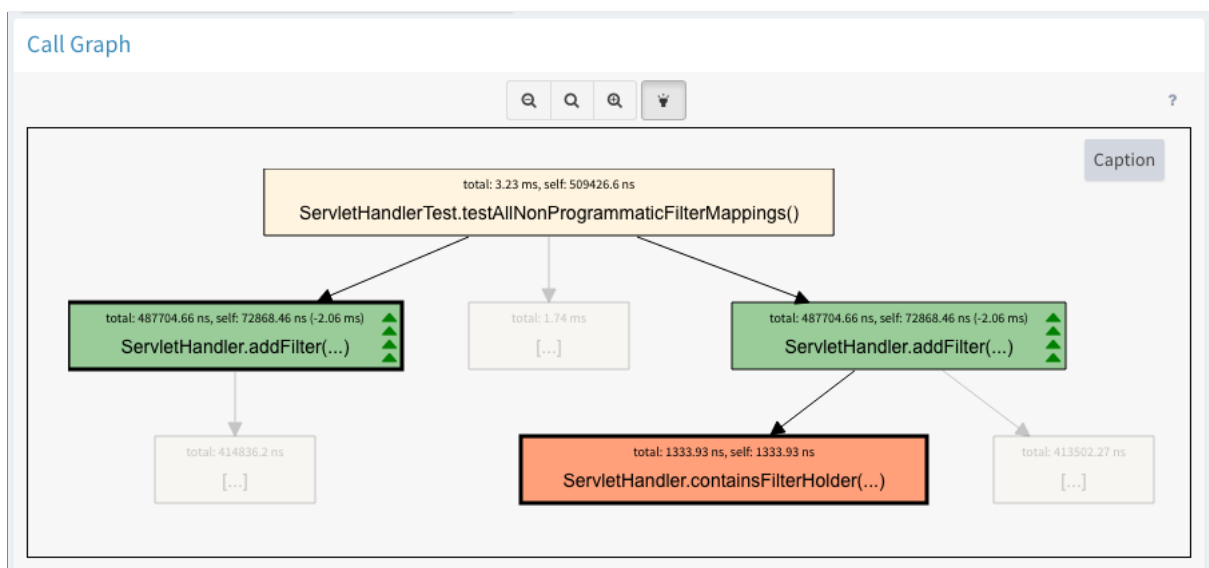


Figura 31: Efeito de destaque do caminho de execução de nós com desvio.

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 32.

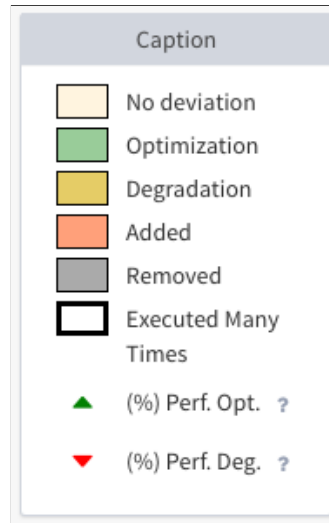


Figura 32: Legenda do grafo de chamadas.

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 33 exemplifica os detalhes de um nó.

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.

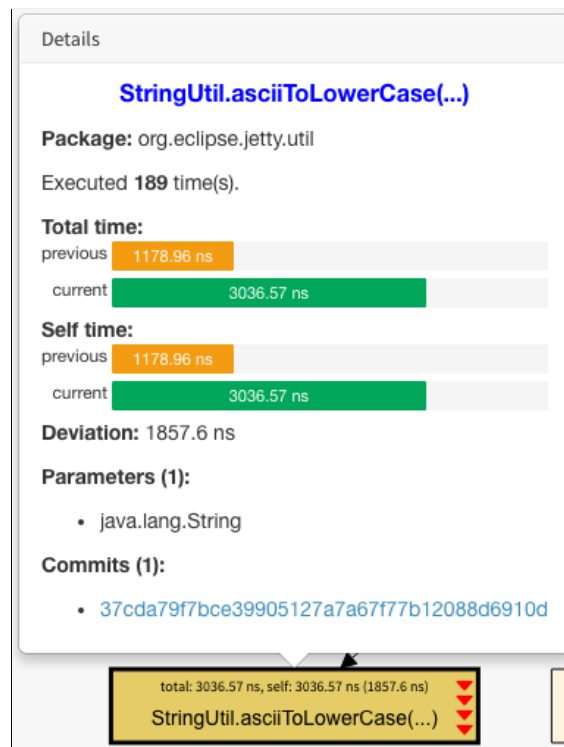


Figura 33: Detalhes de um nó no grafo de chamadas.

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 34 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.

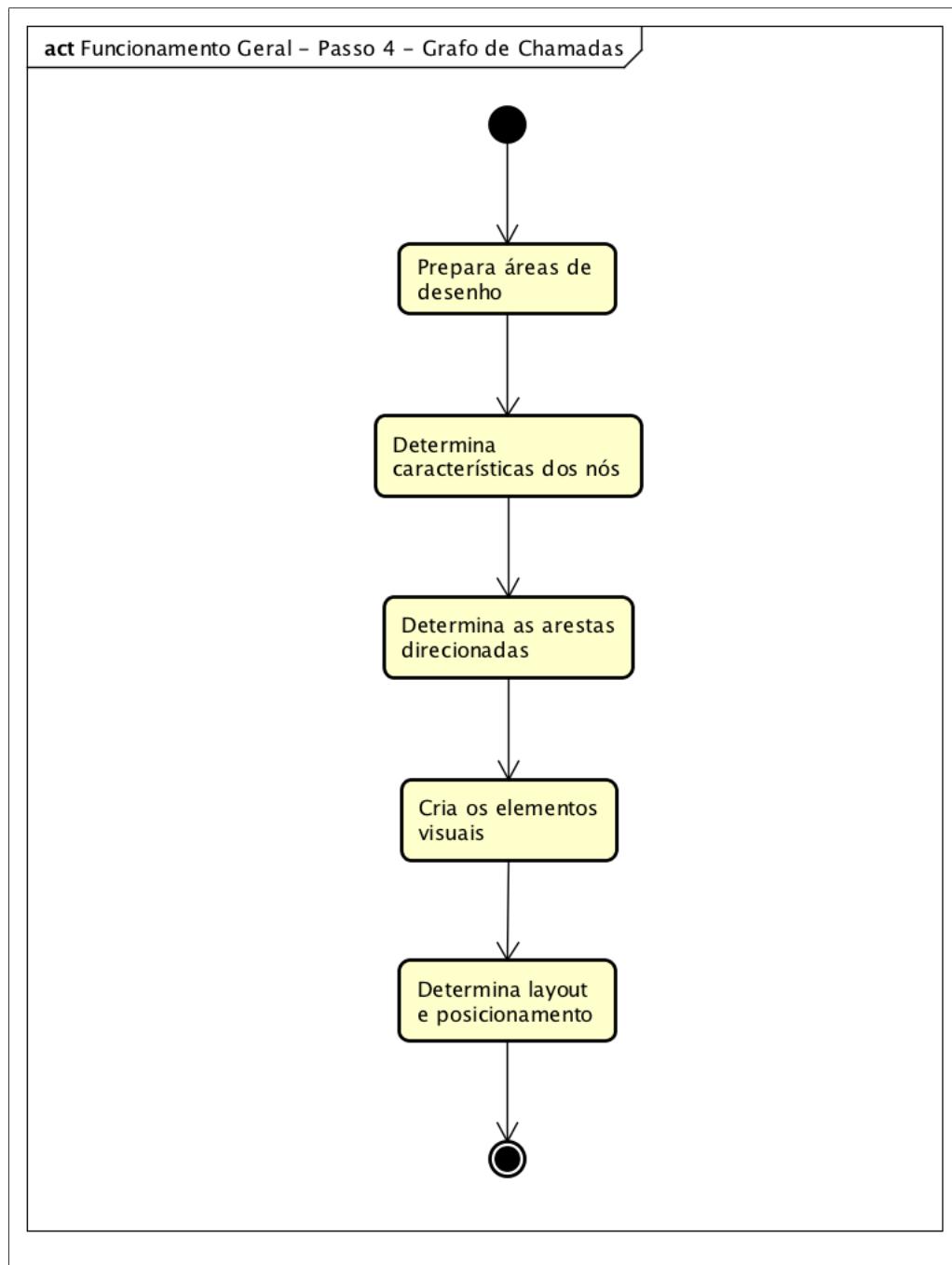


Figura 34: Detalhamento do passo 4 da figura 18, no Grafo de Chamadas.

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.

3.4.6 Exemplo de Uso da Ferramenta: Jetty fica ou sai?

Foi realizado um estudo de caso para exemplificar o uso das visualizações. Para tanto, foi utilizado duas versões da aplicação Jetty [63]: 9.2.6 e 9.3.0.M1. Esse sistema faz parte do Projeto Eclipse e é um *framework open-source* que provê um servidor web além de um servlet container Java. Essa aplicação foi escolhida para o estudo de caso pois todos os tipos de nós foram capazes de ser exemplificados.

Para a primeira fase do *PerfMiner*, a análise dinâmica, todos os testes automatizados foram considerados cenários. Como os testes são implementados em JUnit 4, o aspecto interceptou todos os métodos marcados com a anotação `@Test` como métodos de entrada de um cenário. A análise dinâmica, como mencionado anteriormente, foi executada no mesmo computador para ambas as versões, nas mesmas condições e com todos os serviços não essenciais desabilitados. A suíte de testes do Jetty foi executada 30 vezes para que as medições de desempenho fossem mais precisas em termos de tempo de execução [20].

A resultado da análise foi um total de 11 cenários, sendo 7 com degradação do desempenho e 4 com otimização. No grafo gerado, é possível que dois nós representem o mesmo método, indicando duas execuções, e as arestas direcionadas representam invocações entre os métodos. A figura 35 exemplifica o grafo de um cenário com degradação.

O grafo exibido destaca que o nó raiz, `DefaultServletTest.testFiltered()`, teve uma degradação de desempenho de mais de 75%, indicado pela cor do nó, laranja amarelo, e pelas quatro setas vermelhas para baixo. É possível notar também que dois nós que representam o método `Server.doStart()` foram adicionados. Isso significa que durante a execução do cenário `Entry point for DefaultServletTest.testFiltered` duas cha-

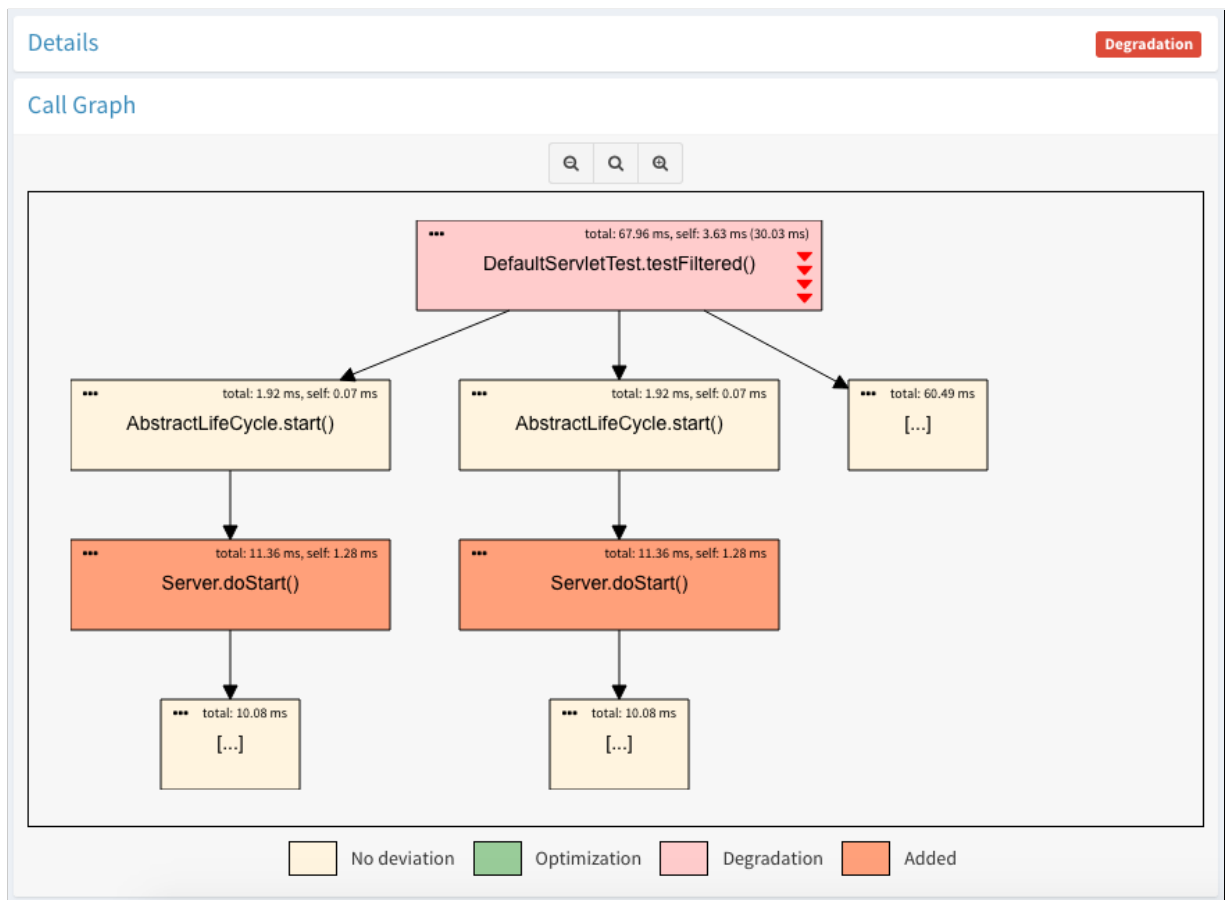


Figura 35: Grafo de chamadas de um cenário com degradação de desempenho.

Adições a esse método foram efetuadas na versão 9.3.0.M1, e elas não existiam na versão 9.3.6. Os nós adicionados são indicados como potenciais causadores de degradação de desempenho, de modo que, possivelmente, esses nós influenciaram na degradação da raiz. Na visualização há nós que não têm relação com os desvios, representados como agrupados, e nós sem desvio, representados na cor marrom claro.

Na mesma figura, além do próprio grafo, na parte superior da seção *Call Graph* é encontrada uma barra de ferramentas com botões de *Zoom Out*, *Zoom to Fit* e *Zoom In*; e na parte inferior é exibida uma legenda ajudando os usuários a identificarem os nós do grafo. Os botões de *zoom* e a legenda são exibidos para todos os grafos, independente da quantidade de nós ou do tipo de desvio ocorrido.

Na figura 36 adiante, a seção *Details* desse cenário é exibida expandida, indicando, no rótulo, que o cenário teve o seu desempenho degradado entre as versões, e que o total de nós para esse cenário foi de 1695.

A figura 37 a seguir apresenta o cenário *Entry point for ServletContextHandler Test.testFallThrough* com otimização de desempenho. Dois nós contribuíram para esse

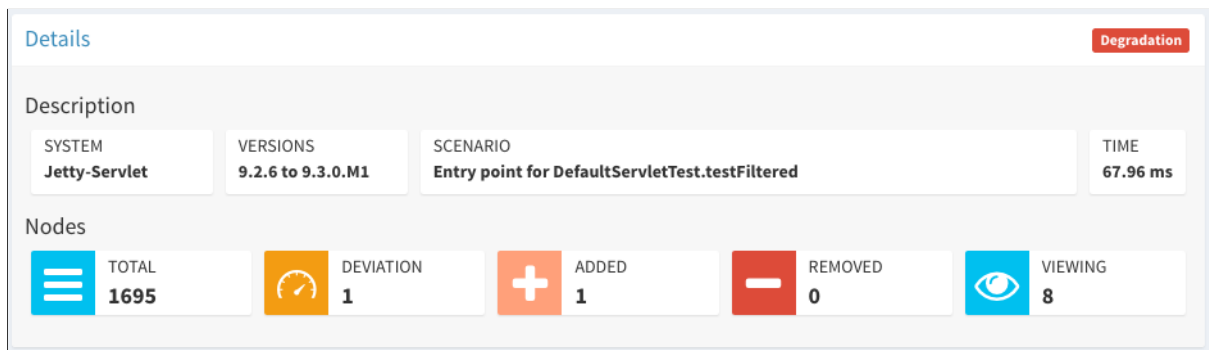


Figura 36: Detalhes de um cenário com degradação de desempenho.

desvio, identificados pela cor verde claro: `Server.doStart()` e `ServletContextHandler.relinkHandlers()`. No primeiro, a variação foi entre 50% e 75%, ao passo que no segundo foi de mais de 75%. Há ainda a representação de nós agrupados e sem desvio.

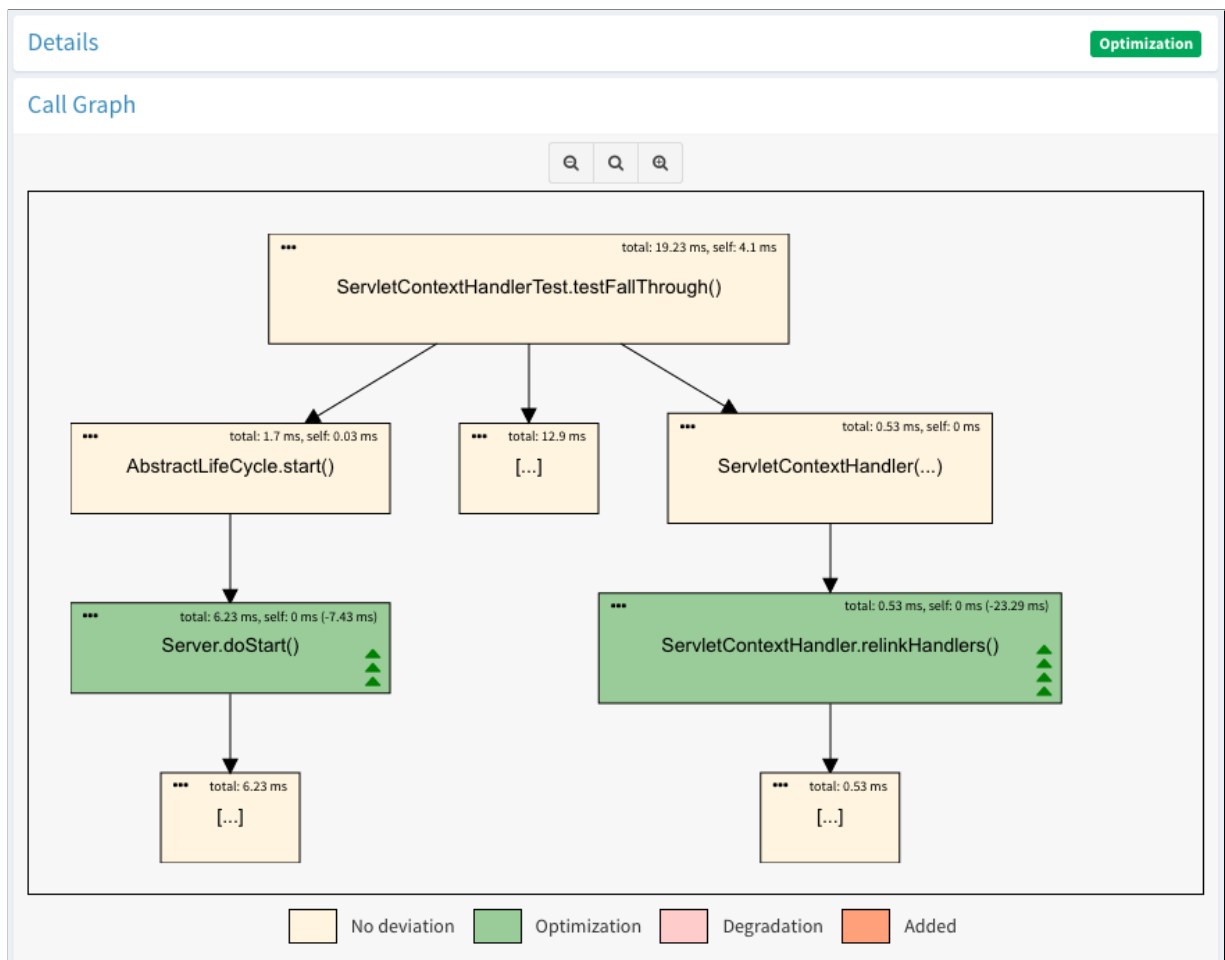


Figura 37: Grafo de chamadas de um cenário com otimização de desempenho.

A seção *Details* expandida pode ser vista na figura 38. O rótulo de otimização é exibido no canto superior esquerdo indicando o tipo de desvio ocorrido no cenário. A figura indica um total de 986 nós para esse cenário.

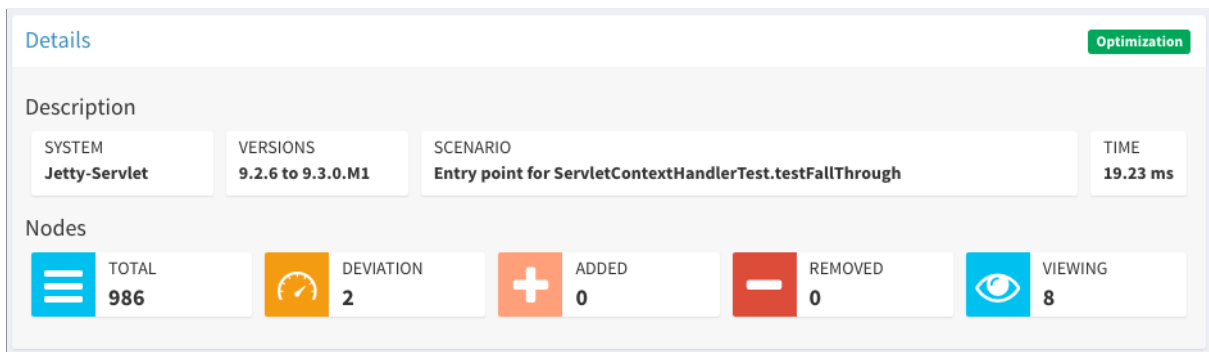


Figura 38: Detalhes de um cenário com otimização de desempenho.

3.5 Considerações

Um dos principais desafios do processo de manutenção de um software é compreendê-lo adequadamente, em especial, a sua arquitetura. A falta de acompanhamento da evolução da arquitetura ao longo do tempo pode levar a sua degradação, impactando os seus atributos de qualidade. Este capítulo apresentou visualizações de arquitetura de software que visam tornar direta e fácil a monitoração da evolução arquitetural, com relação ao 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 cujo objetivo é avaliar se as visualizações são úteis para representar os desvios de desempenho encontrados durante a análise desses sistemas, além de avaliar a facilidade de se encontrar informações e a possível aplicabilidade da ferramenta como parte integrante dos processos de desenvolvimento dos sistemas analisados.

O restante deste capítulo está organizado como segue: seção 4.1 apresenta como o estudo foi projetado, incluindo as principais contribuições (subseção 4.1.1), os objetivos e questões de pesquisa (subseção 4.1.2), os sistemas analisados (subseção 4.1.3) e os procedimentos do estudo (subseção 4.1.4); a seção 4.2 exibe os resultados do estudo e a seção 4.3 conclui o capítulo, apresentando as ameaças à validade e discussões sobre os resultados.

4.1 Projeto

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.

De maneira geral, foram aplicadas as fases discutidas no capítulo 3, subseção 3.1.1, além do procedimento necessário para gerar os dados para as visualizações, conforme descrito na seção 3.2 do mesmo capítulo. Em suma: (i) primeiro, os casos de testes automatizados (cada um deles representando um cenário) dos sistemas foram executados e monitorados, resultando em múltiplos bancos de dados; (ii) depois, os dados coletados foram comparados, agrupando releases subsequentes para cada sistema; (iii) então, os elementos identificados com desvios de desempenho foram minerados nos seus sistemas de controle de versão, com o intuito de descobrir os *commits* que alteraram esses artefatos;

(iv) por fim, foi executado, para cada versão de cada sistema, o procedimento necessário para gerar os dados que dão suporte às visualizações. Após isso, o questionário online foi elaborado e aplicado.

O questionário foi enviado para todos os contribuidores dos sistemas que possuíam endereço de email registrado na página de contribuidores de cada sistema ou na página do seu perfil. Dos 114 contribuidores que receberam o questionário, 16 deles responderam com algum feedback.

4.1.1 Principais Contribuições

As principais contribuições desta avaliação 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) para os mesmos sistemas, a percepção visual do tipo de desvio dos cenários através da análise de múltiplas releases; (iii) a identificação visual das potenciais causas dos desvios de desempenho dos cenários dos sistemas analisados; (iv) ...

CONTINUAR...

4.1.2 Objetivos e Questões de Pesquisa

O principal objetivo deste estudo é investigar se a ferramenta *Performance QA Evolution*, com suas visualizações e propriedades visuais oferecidas, é capaz de representar as evoluções de desempenho dos cenários analisados, se proporciona uma fácil identificação desses desvios pelos usuários 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.

QP2. Há indícios de que os usuários conseguem identificar os métodos com desvios de desempenho e os respectivos *commits* usando as visualizações disponíveis na ferramenta?

QP1. A ferramenta proposta é capaz de exibir representações visuais que

fornece informações sobre os cenários de determinados releases de um sistema que tiveram o maior desvio de desempenho dentre os analisados? Após as análises realizadas nos releases dos sistemas, a expectativa é que a visualização da sumarização de cenários seja capaz de exibir metáforas visuais que permitam aos usuários identificar informações sobre os cenários. É esperado que a complexidade na identificação dessas informações seja baixa através da metáfora visual e interações oferecidas para ajudar os usuários nessa identificação.

QP2. A ferramenta proposta é capaz de exibir visualmente, para determinado cenário analisado, os métodos identificados com desvios de desempenho, além dos *commits* que possivelmente causaram esses desvios? A expectativa é que a visualização do grafo de chamada seja capaz de exibir elementos visuais que permitam aos usuários encontrar as informações sobre os métodos com desvios de desempenho, além das suas possíveis causas. Espera-se, também, que a complexidade de identificação dessas informações seja baixa através das metáforas visuais oferecidas e das interações implementadas. É importante que os usuários tomem conhecimento sobre as modificações no código-fonte que geraram algum impacto no desempenho dos sistemas.

QP3. Os usuários dos sistemas analisados veem benefícios ou vantagens de se usar a ferramenta proposta em seus processos de desenvolvimento? Espera-se, com essa questão de pesquisa, verificar se os usuários dos sistemas analisados utilizariam a ferramenta em seus processos de desenvolvimento. Além disso, outro aspecto interessante é saber em qual momento desse processo os desenvolvedores vislumbram que a ferramenta pode ser usada.

4.1.3 Sistemas

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

- 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;
- Estar listado em uma das categorias do site <http://java-source.com> (site de projetos *open-source* em Java). Os sistemas foram escolhidos de modo que não houvesse repetição de categorias;

- 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 um 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.

4.1.4 Procedimentos

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

4.1.4.1 Passo 1 - Coleta e Preparação dos Dados

Após a escolha dos sistemas e suas releases, os códigos-fonte de todas elas foram baixadas 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.

4.1.4.2 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 com-

putador 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 *Performance QA Evolution*. A tabela 2 a seguir mostra como ficaram organizados esses pares para cada sistema.

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

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 contrutores com degradação ou otimização de desempenho. Já na fase 3, os elementos com desvios de desempenho foram minerados nos seus sistemas de controle de versão e sistemas de gerenciamento de tarefas. Ao final das 3 fases do *PerfMiner*, são gerados os artefatos de saída requeridos para o *Performance QA Evolution*. 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 *Performance QA Evolution*.

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, pu-

blicar o resultado das análises em um ambiente na nuvem, o Heroku¹, cujo endereço é <http://apvis.herokuapp.com>.

4.1.4.3 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 e nos seus respectivos projetos. A segunda seção continha questões sobre a experiência de uso de ferramentas de *profiling* e APM, e sobre estratégias que os participantes utilizam para gerenciar o desempenho das funcionalidades dos respectivos sistemas. Na terceira seção, foram perguntadas questões sobre a visualização do grafo de chamadas. Foram dadas situações 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:

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 ba-

¹<http://www.heroku.com>

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

seadas 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. Os dois tipos do questionário podem ser consultados no apêndice A.

Com o questionário elaborado, foram coletados os dados de email 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 publicização do seu endereço de email 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.

4.1.5 Caracterização dos Participantes

Dos 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, consequentemente, as suas respostas; (ii) outro

usuário, ao invés de abrir a ferramenta para responder às questões 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 39a) e possuem entre 1 e 10 contribuições no seu projeto (figura 39b). 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 podem dar credibilidade às suas respostas.

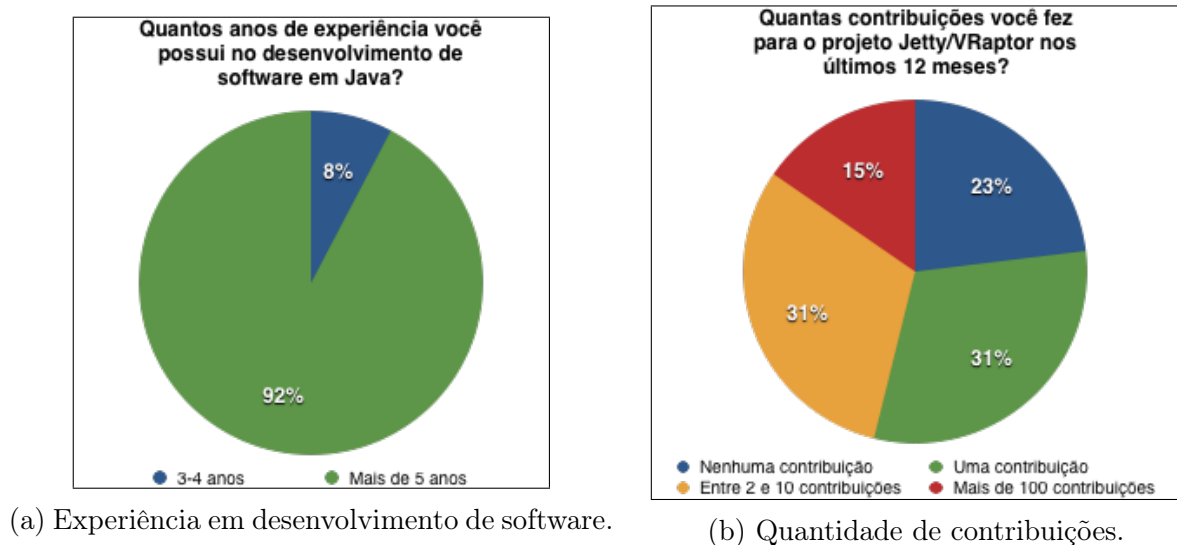


Figura 39: Dados demográficos dos participantes.

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

Nesta seção serão discutidos os resultados obtidos após a execução da ferramenta para os sistemas mencionados e após 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 seção de Procedimentos (4.1.4) 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.

A figura 40 apresenta exemplos de visualizações de sumarização de cenários. Na figura 40a 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. Já a figura

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

Tabela 5: Resumo para o VRaptor.

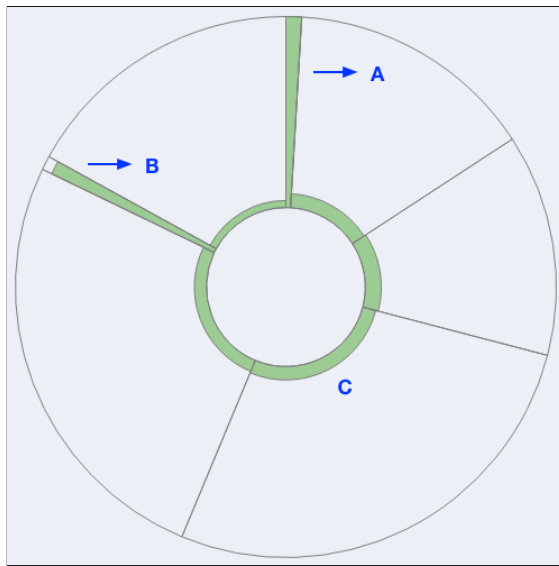
Versão Inicial	Versão Final	Cenários Degradados	Cenários Otimizados	Total de Cenários
4.0.0.Final	4.1.0.Final	36 (4,86%)	41 (5,54%)	740
4.1.0.Final	4.1.1	0 (0%)	0 (0%)	745
4.1.1	4.1.2	20 (2,66%)	10 (1,33%)	750
4.1.2	4.1.3	9 (1,19%)	2 (0,26%)	752
4.1.3	4.1.4	0 (0%)	0 (0%)	739
4.1.4	4.2.0.RC1	0 (0%)	1 (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

40b 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%.

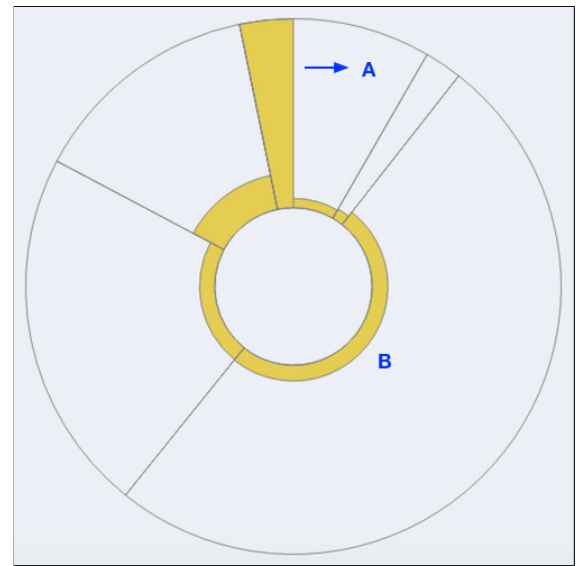
Os dois exemplos comentados anteriormente mostram cenários com apenas um tipo de desvio de desempenho entre as versões. Na figura 40c são exibidos cenários de ambos os tipos, 3 degradações e 2 otimizações, para o Jetty, versões 9.3.14 para 9.3.15. O cenário indicado pela letra A foi de maior tempo de 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 40d 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.

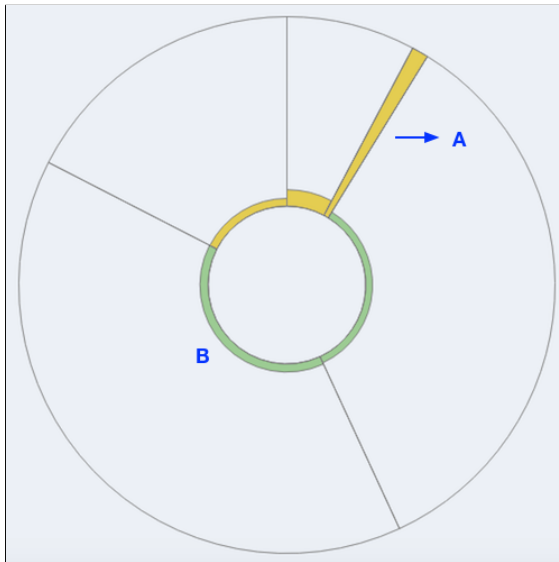
É importante salientar que os cenários apresentados na figura 40 não são todos os cenários existentes para as versões analisadas, mas sim apenas os cenários identificados com desvios de desempenho após a execução da ferramenta. De acordo com a tabela 4, existem 155 cenários entre as versões 9.3.12 e 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.



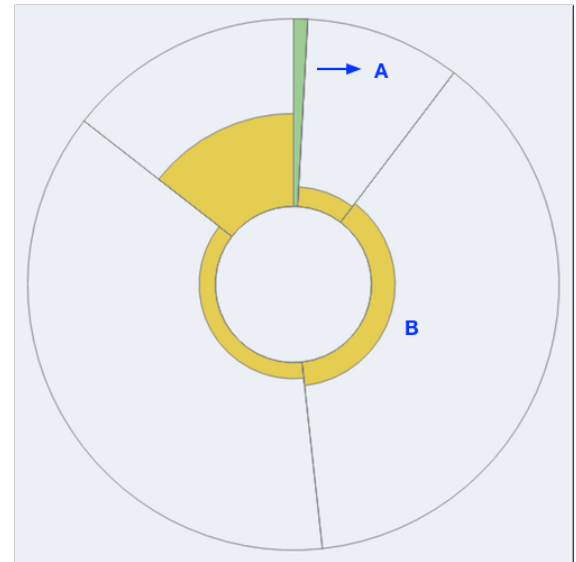
(a) Jetty, 9.3.12/9.3.13.



(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 40: Exemplos da visualização de Sumarização de Cenários.

Houve casos em que a identificação dos cenários na visualização pode ser prejudicada, de maneira geral: quando há muitos cenários com desvios de desempenho para o par

de versões analisados. 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 41 apresenta esses casos.

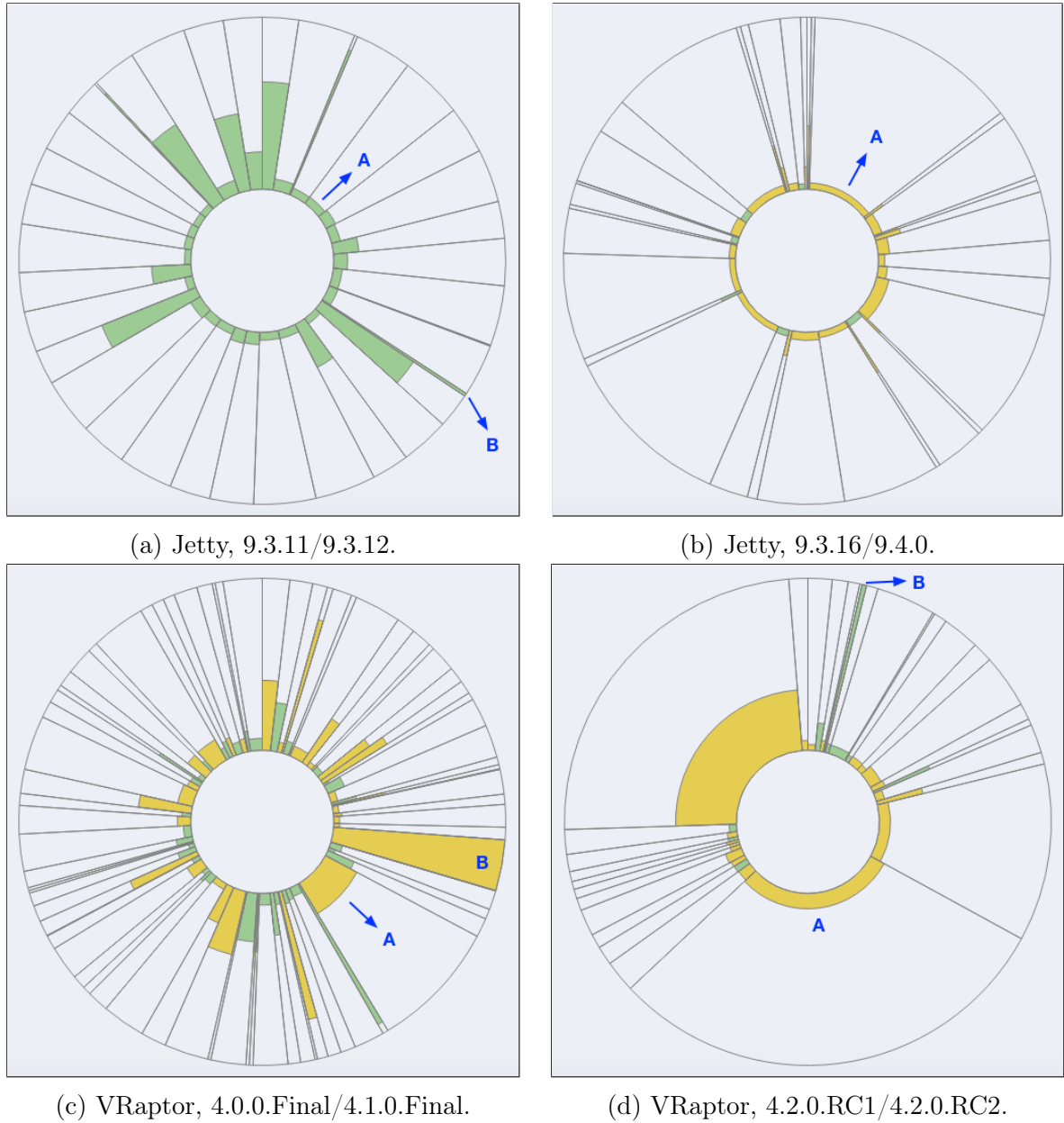


Figura 41: Exemplos da visualização de Sumarização de Cenários com excesso de cenários.

Na figura 41a são mostrados 36 cenários de otimização para o sistema **Jetty**, versões 9.3.11 para 9.3.12. Nesse caso, os cenários estão bem distribuídos, exceto 4 deles que tiveram baixa porcentagem de desvio e, portanto, suas fatias estão finas com relação às demais. Por causa da boa 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 41b 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 41c 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 também 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 41d 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 41a e 41c, tanto o cenário com maior desvio de desempenho quanto o cenário com maior tempo de execução podem ser identificados.

Essa dificuldade em se localizar os cenários não necessariamente está ligada 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 e cenários com baixas porcentagens de desvio com relação aos demais, a localização destes pode se tornar uma tarefa difícil.

Para resolver ou minimizar essa situação, podem ser feitas adaptações na implementação dessa visualização: (i) os cenários poderiam ser agrupados em módulos do sistema, de acordo com critérios definidos pelos usuários. Dessa forma, várias visualizações de sumariação 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 cenários com maior porcentagem de desvio de desempenho, e assim por diante nas páginas seguintes.

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.

4.2.1.2.1 Supressão de Nós Exibidos

A ferramenta dispõe de um algoritmo de supressão de nós que 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 desses nós, ou seja, se, para a mesma hierarquia de chamadas, um mesmo nó foi executado novamente, ele será marcado como repetição 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.

Uma vez identificadas todas as repetições, o algoritmo verifica todos os nós que podem ser transformados em nós de agrupamento (seção 3.4.3.6). Os nós que não estão diretamente relacionados com os nós a serem exibidos (degradados, otimizados, adicionados e removidos) são agrupados em um único nó, exceto os nós pai e, se houver, nós avôs, bisavôs, tataravôs e pentavôs. 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 nós existentes nessa hierarquia de chamadas são agrupados em um único nó. A figura 42 ilustra esse caso, onde o nó com desvio de desempenho a ser exibido é o `DefaultConverters.findConverterType(...)` e o nó 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 43, onde o nó degradado `Exclusions.shouldSkipField(...)` 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 43, o nó indicado com a letra A mostra que há uma ou mais hierarquias de chamadas agrupadas.

Com esse algoritmo, a redução da quantidade de nós a serem exibidos para determinado cenário foi importante, consequentemente, diminuindo a complexidade de se encontrar informações. O gráfico exibido na figura 44a 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.

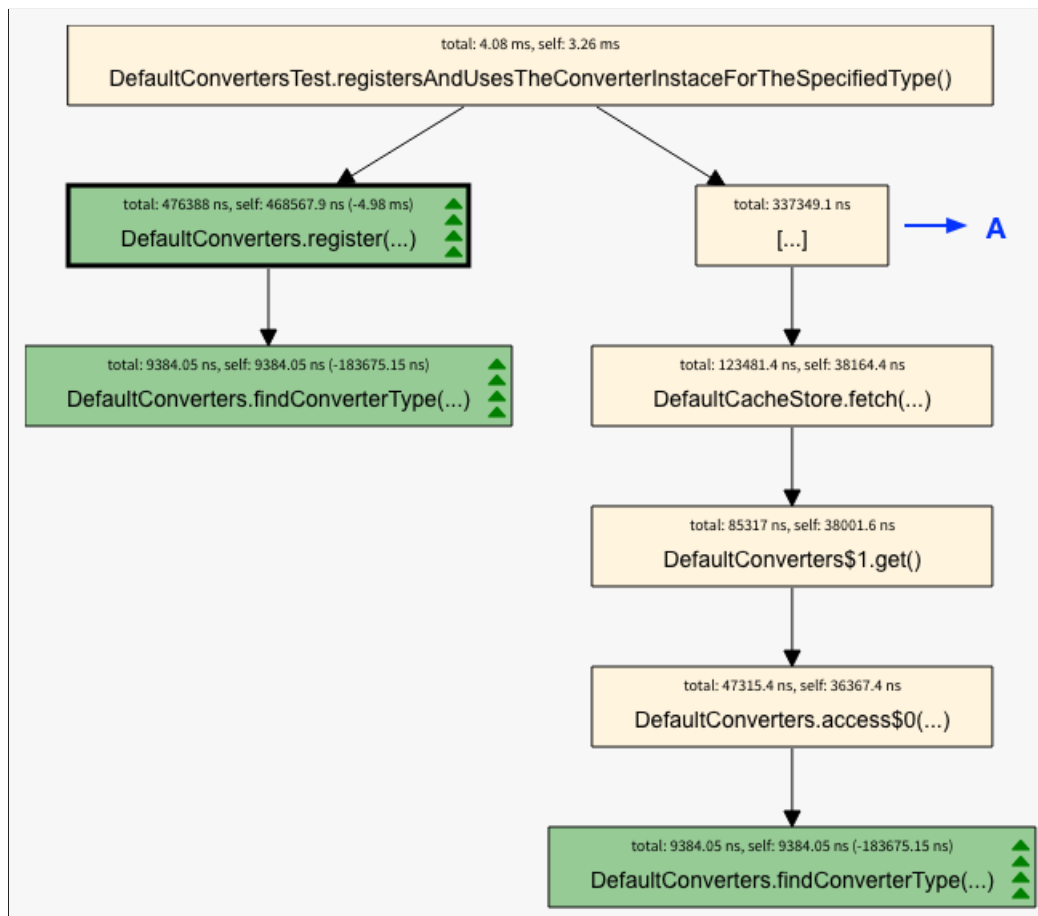


Figura 42: Exemplo de nós agrupados durante a hierarquia de chamadas.

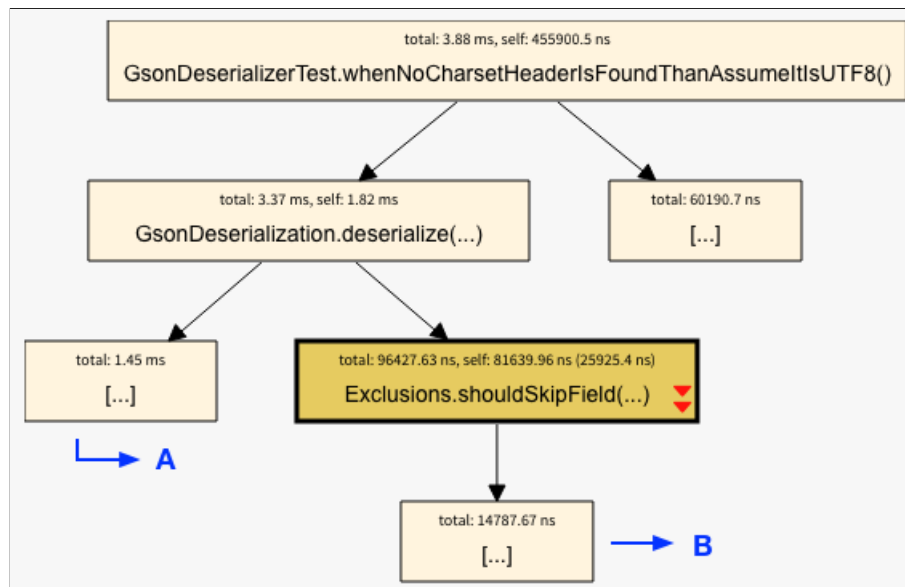


Figura 43: Exemplo de nós hierarquias em árvores adjacentes (A) e em nós filhos (B).

O gráfico da figura 44b 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 44b, a porcentagem de redução foi de 98,29%, indicando a eficiência do algoritmo.

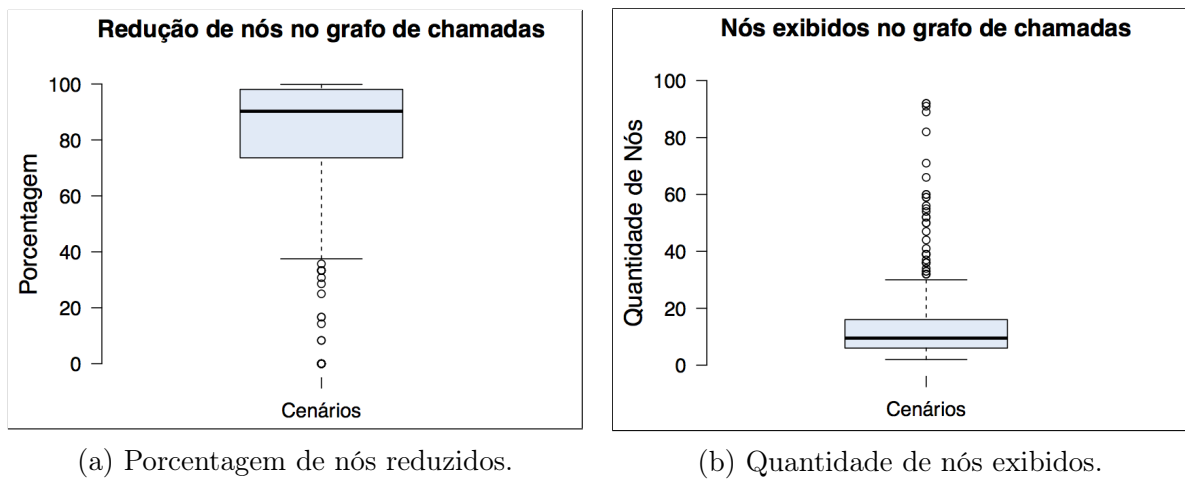


Figura 44: Gráficos *boxplot* sobre o algoritmo de redução de nós.

4.2.1.2.2 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*: `Entry point for AsyncContextListenersTest.testAsyncDispatchAsyncCompletePreservesListener (C1)`
- *Alta quantidade de nós otimizados*: `Entry point for ServletHolderTest.testUnloadableClassName (C2)`
- *Baixa porcentagem de redução de nós*: `Entry point for SafeResourceBundleTest.shouldReturnKeyBetweenQuestionMarksWhenKeyDoesntExist (C3)`

- *Alta porcentagem de redução de nós*: Entry point for `ServletContextHandlerTest.testInitOrder` (C4)
- *Maior degradação*: Entry point for `DefaultEnvironmentTest.shouldUseContextInitParameterWhenSystemPropertiesIsntPresent` (C5)
- *Menor degradação*: Entry point for `PostServletTest.testGoodPost` (C6)
- *Maior otimização*: Entry point for `XStreamXmlDeserializerTest.shouldBeAbleToDeserializeADogWhenMethodHasMoreThanOneArgumentAndTheXmlIsTheLastOne` (C7)
- *Menor otimização*: 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.

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

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 em 4,07%, apesar da alta quantidade de nós otimizados. Esses nós percentem 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 45 a seguir ilustra o grafo de chamadas do cenário C3.

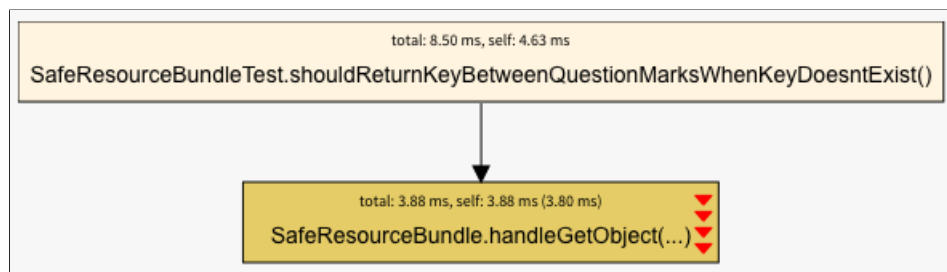


Figura 45: Grafo de Chamadas do cenário C3.

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 46 adiante exemplifica esse grafo.

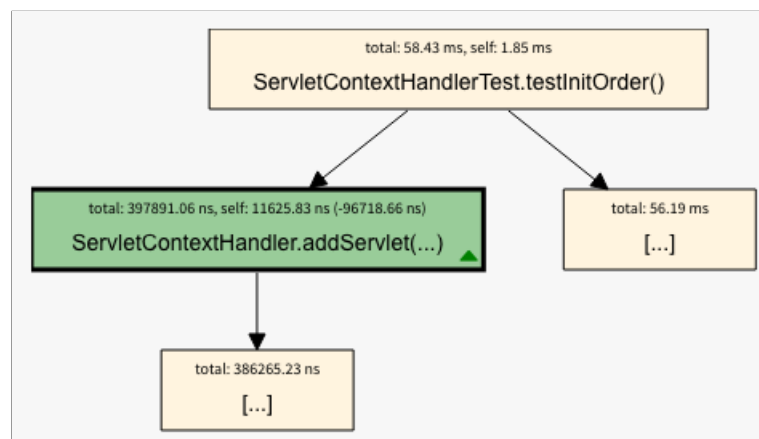


Figura 46: Grafo de Chamadas do cenário C4.

O cenário C5 foi o que mais degradou o seu tempo de execução em relação à versão

anterior: 1.646%. O grafo da figura 47 mostra que o único nó com desvio de desempenho 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.

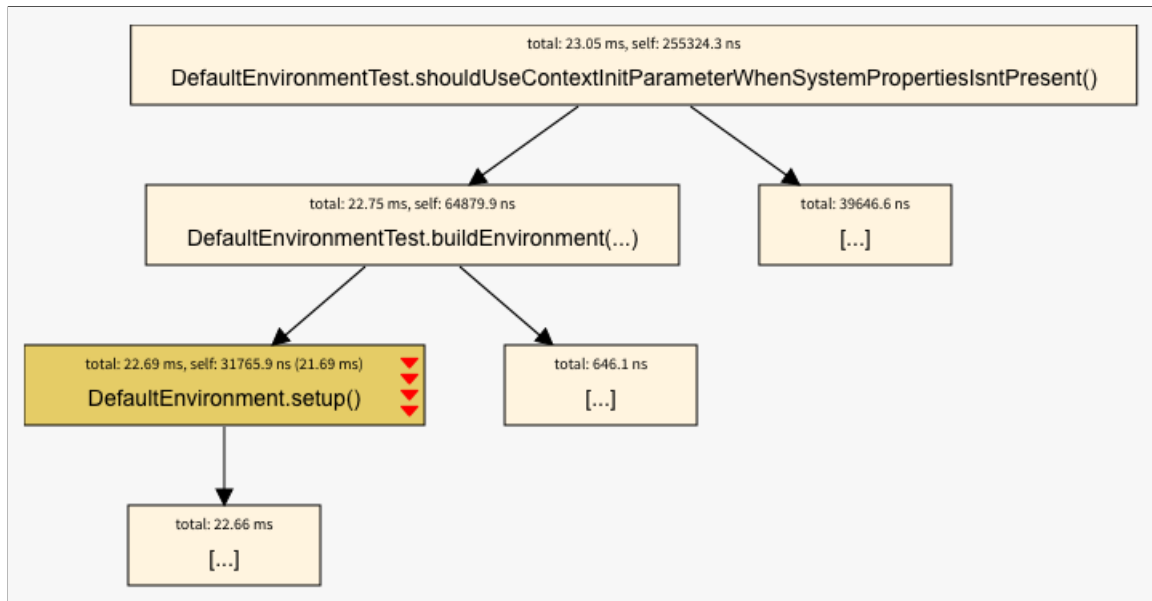


Figura 47: Grafo de Chamadas do cenário C5.

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 48 exemplifica esse grafo.

O último cenário destacado é o C8, que possuiu a menor otimização. Como exibe o grafo da figura 49, um único nó foi responsável por esse desvio, `XStreamBuilderImpl.xml-Instance()`. 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%.

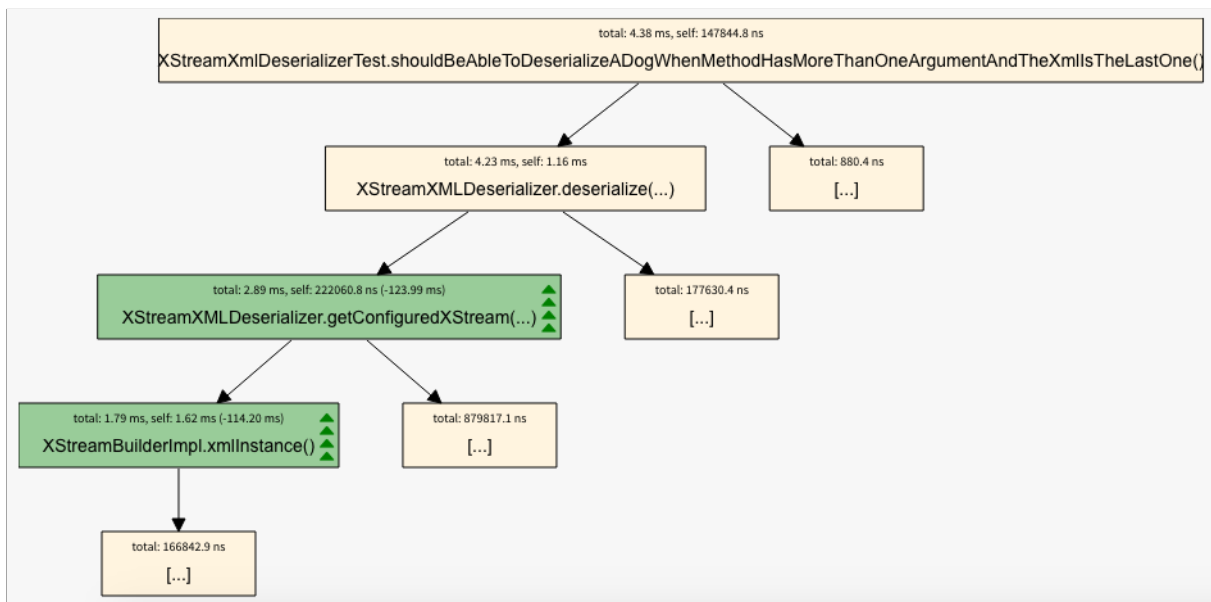


Figura 48: Grafo de Chamadas do cenário C7.

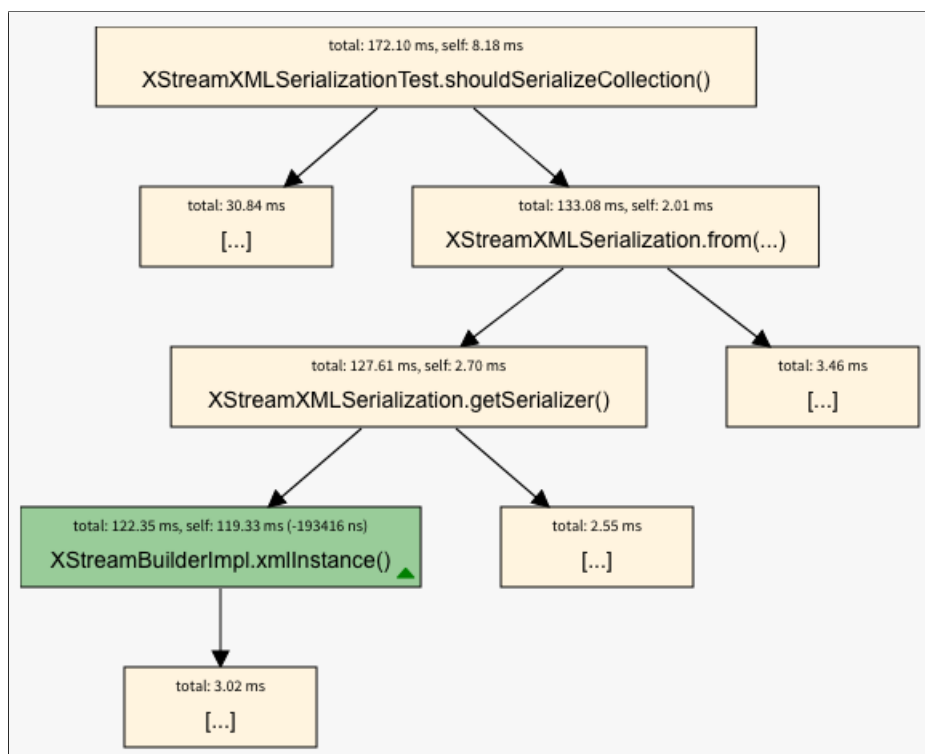


Figura 49: Grafo de Chamadas do cenário C8.

4.2.2 Questionário Online

QP1. A ferramenta proposta é capaz de exibir representações visuais que fornecem informações sobre os cenários de determinado release de um sistema que tiveram o maior desvio de desempenho dentre os analisados? Os questionários aplicados coletaram dados sobre a utilização da visualização de sumaria-

ção de cenários para encontrar informações específicas, bem como da utilização de dados tabulares para encontrar as mesmas informações. Foi observado que...

QP2. A ferramenta proposta é capaz de exibir visualmente, para determinado cenário analisado, os métodos identificados com desvios de desempenho, além dos *commits* que possivelmente causaram esses desvios?

FALTAM MUITAS QUESTÕES AQUI...

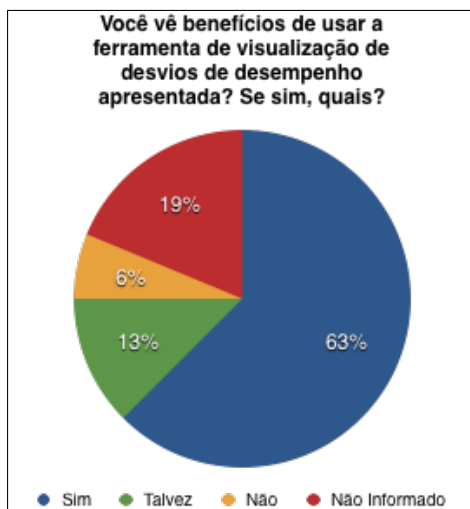
QP3. Os usuários dos sistemas analisados veem benefícios ou vantagens de se usar a ferramenta proposta em seus processos de desenvolvimento? Através do questionário aplicado foi verificado se os participantes veem benefícios na utilização de uma ferramenta como o *Performance QA Evolution*, além de se eles utilizariam, e de que maneira, a ferramenta em seus processos de desenvolvimento.

Dentre as perguntas dos questionários continha uma específica para os participantes relatarem suas impressões positivas e negativas para cada visualização implementada: “*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*”.

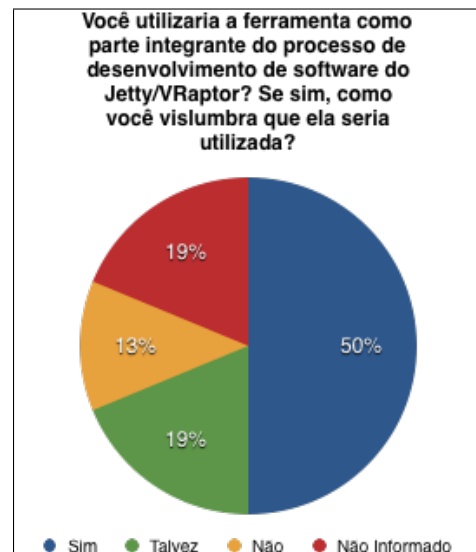
Para a visualização da Sumarização de Cenários o sentimento foi dividido. O participante P1GD não se sentiu confortável com o gráfico de rosca utilizado para a visualização: “*gráficos em torta ou semelhantes normalmente querem representar 100% de algo. Usar a*

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”*.

Quando responderam a questão *“Você vê benefícios de usar a ferramenta de visualização de desvios de desempenho apresentada? Se sim, quais?”*, 63% dos participantes disseram que veem benefícios no uso da ferramenta, conforme exibe a figura 50a. Dentre os benefícios comentados pelos participantes, alguns exemplos do que foi informado são: o participante P8GC disse que *“aparentemente pode trazer análises mais assertivas do atual desempenho do sistema”*, 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”*, já o participante P6GD informa que *“sim. É interessante saber de maneira precisa e quantitativa a quantidade e nível de melhorias e degradações entre versões”*.



(a) Benefícios da ferramenta.



(b) Utilidade da ferramenta.

Figura 50: Questões sobre os benefícios e utilidades da ferramenta proposta.

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?”*, metade (50%) 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 50b sumariza as respostas à essa questão.

Vale destacar que 19% 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*” e o participante P19GD comentou que “*não neste momento. Posteriormente poderia fazer parte de uma análise de status do projeto antes de um release*”.

4.3 Considerações

5 Trabalhos Relacionados

Este capítulo confronta este trabalho contra 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 exibidas 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][67][68]. Outras abordagens focam em visualizar as métricas do software em diferentes versões [50][52][53][69]. 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* [13] e o *YourKit Java Profiler* [14] são ferramentas comerciais para realizar profiling de aplicações na linguagem Java. Sandoval Alcocer et al.[10] 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.\[15\]](#) 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* [16], *AppDynamics* [17], *Dynatrace* [18] e *Pinpoint* [19]. 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.\[15\]](#) 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.\[10\]](#) propõe uma abordagem visual para entender a causa de degradações de desempenho, comparando o desempenho de duas versões do sistema. Trata-se de uma visualização polimétrica, 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.

O trabalho proposto se diferencia do de [Sandoval Alcocer et al.\[10\]](#) pelo fato de: (i) ser compatível com a linguagem Java; (ii) exibir a evolução do desempenho global do software entre as versões; (iii) exibir a evolução de cada cenário entre as versões do softwares; e (iv) utilizar diferentes elementos visuais para destacar a evolução do desempenho na visualização do *call graph* dos cenários.

[Bergel, Robbes e Binder\[70\]](#) propôs uma abordagem cujo objetivo é comparar duas versões de um software. A visualização exibe 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. No entanto, essa abordagem considera apenas se um método gastou ou não mais tempo de execução do que na versão anterior. O trabalho proposto se diferencia por adicionar mais métricas, como a porcentagem de desvio de desempenho dos métodos e se houve métodos adicionados ou removidos. Ainda, facilita a identificação da causa do desvio de desempenho, e irá prover aos desenvolvedores e arquitetos a possibilidade de verificar as mudanças no código-fonte ocorridas entre as duas versões comparadas.

[Mostafa e Krintz\[71\]](#) 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. Eles usam como base o algoritmo de correspondência de árvores comuns para comparar duas CCTs. No entanto, a abordagem, diferentemente deste trabalho, não detecta nenhum nó adicionado ou removido, o que é fundamental para o entendimento do real impacto de um novo nó no desempenho de determinada árvore de chamadas.

6 Conclusão

Este capítulo apresenta na seção 6.1 o cronograma das atividades que guiarão a continuidade do trabalho durante o próximo semestre. Na seção 6.2 são mostradas algumas limitações já conhecidas deste trabalho.

6.1 Cronograma de Atividades

A figura 51 adiante apresenta o cronograma de atividades a ser seguido durante o próximo semestre. Vale salientar que todas as datas apresentadas na figura são estimativas, portanto, são datas aproximadas. Dessa forma, podem ocorrer variações nas datas planejadas.

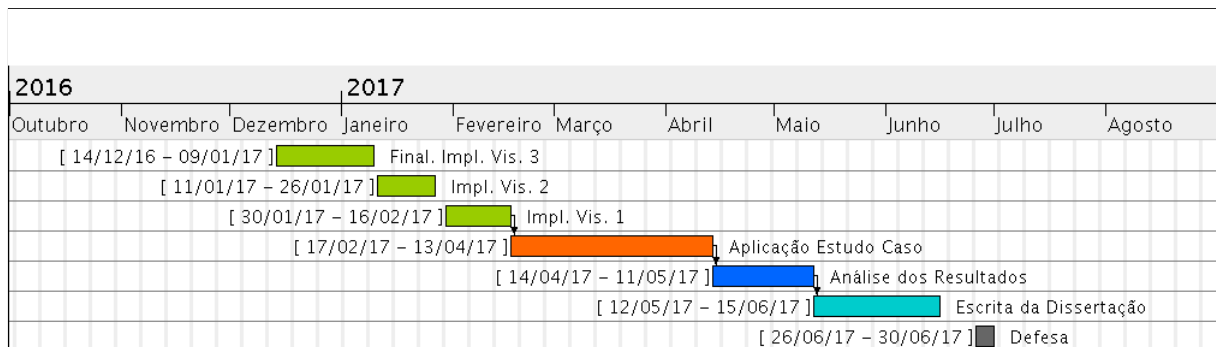


Figura 51: Cronograma de atividades.

Após a qualificação, será dada continuidade a implementação da visualização 3, o grafo de chamadas. Será implementada a opção de o desenvolvedor ou arquiteto usuário do sistema obter acesso ao código-fonte responsável por desvios de desempenho indicados na visualização. Esta atividade durará até a primeira metade do mês de janeiro/17.

Após a conclusão da implementação da visualização do grafo de chamadas, será implementada a visualização 2 - sumarização dos cenários. Essa atividade foi escolhida por ser menos complexa do que a visualização 1, evolução do desempenho, e a previsão de duração é até o final do mês de janeiro/17. Na sequência, para finalizar a etapa de desen-

volvimento do trabalho, a visualização de evolução do desempenho será implementada, em tarefa que tem conclusão prevista para a primeira metade de fevereiro/17.

Passada a etapa de desenvolvimento, serão iniciados os estudos empíricos planejados, conforme descritos neste documento. Essa etapa, por envolver desenvolvedores e arquitetos dos sistemas escolhidos, terá maior tempo de duração e está prevista para ser finalizada na primeira quinzena de abril/17. Terá duração total de aproximadamente 50 dias.

Em abril/17, está planejado que se inicie as análises dos estudos de caso realizados, em tarefa que durará, aproximadamente, 25 dias. Pretende-se analisar qualitativamente os dados obtidos para, então, obter os resultados do trabalho.

Após essa etapa, será retomada a escrita do documento, relatando o desenvolvimento das visualizações 1 e 2, os detalhes da aplicação dos estudos de caso, os resultados obtidos nesses estudos, bem como será revisto e, se cabível, refinado cada capítulo do documento. A defesa desta dissertação de mestrado é planejada para a última semana de junho/17.

6.2 Limitações

Algumas limitações do trabalho são conhecidas, em particular para a visualização do grafo de chamadas:

- *Tamanho do grafo*: a quantidade de nós com desvios de desempenho de uma versão para outra do software pode ser em grande quantidade de modo a comprometer o desempenho da própria visualização, comprometer a usabilidade e a 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 ou adicionados - há possibilidades de acontecer;
- *Evoluções espaçadas ou bruscas*: caso sejam analisadas duas versões muito distantes entre si, as modificações podem ser enormes, fazendo com que o grafo gerado seja grande e complexo, resultando no problema do tamanho do grafo apresentado no item anterior. Para análises desse tipo, pode ser recorrido a uma análise em alta granularidade. O ideal ao utilizar a ferramenta é analisar versões próximas, para beneficiar-se da baixa granularidade apresentada pelo grafo de chamadas.

Requisitos não-funcionais: segurança e desempenho da aplicação web...

Daltônicos (acessibilidade): podem não conseguir distinguir corretamente as cores...

Número de aplicações analisadas: podem não refletir todos as situações e quantidade de informações para as visualizações podem ser excessivas, tornando a visualização difícil de compreender...

Automatização: apesar de possuir rotinas automatizadas, a abordagem não está completamente automatizada... falar de maneira sumária dos passos em que ainda necessita de intervenção manual, de que não será mais necessário o AWS com a automatização...

Touch support: a aplicação foi testada e não está preparada para dispositivos móveis... adaptações na implementação da visualização de sumarização de cenários

Também pode ser considerada uma limitação, herdada do *PerfMiner*, o fato de o conjunto de visualizações apresentar apenas dados do atributo de qualidade de desempenho. No entanto, a ferramenta está apta a atender a outros atributos de qualidade.

Referências

- [1] WILLIAMSON, T.; SPENCER, N. A. Development and Operation of the Traffic Alert and Collision Avoidance System (TCAS). *Proceedings of the IEEE*, v. 77, n. 11, p. 1735–1744, 1989. ISSN 15582256.
- [2] Manfred Burckhardt, Franz Brugger e Andreas Faulhaber. *Anti-lock brake system*. 1989.
- [3] STEINBRÜCKNER, F. Coherent software cities: Supporting comprehension of evolving software systems. In: *IEEE International Conference on Software Maintenance, ICSM*. IEEE, 2010. p. 1–2. ISBN 9781424486298. Disponível em: <<http://ieeexplore.ieee.org/document/5610421/>>.
- [4] BALL, T.; EICK, S. Software visualization in the large. *Computer*, v. 29, n. 4, p. 33–43, apr 1996. ISSN 00189162. Disponível em: <<http://ieeexplore.ieee.org/document/488299/>>.
- [5] 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.
- [6] 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>>.
- [7] 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>.
- [8] 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>>.
- [9] 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>>.
- [10] 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.
- [11] D'AMBROS, M.; LANZA, M. Visual software evolution reconstruction. *Journal of Software Maintenance and Evolution*, v. 21, n. 3, p. 217–232, 2009. ISSN 1532060X.

- [12] VISUALVM. *VisualVM*. 2016. Disponível em: <<https://visualvm.github.io>>.
- [13] JPROFILER. JProfiler. 2016. Disponível em: <<https://www.ej-technologies.com/products/jprofiler/overview.html>>.
- [14] PROFILER, Y. J. *YourKit Java Profiler*. 2016. Disponível em: <<https://www.yourkit.com/java/profiler/features/>>.
- [15] 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 Workshop on Mining Software Repositories - MSR '16*, p. 1–12, 2016. Disponível em: <<http://dl.acm.org/citation.cfm?doid=2901739.2901774>>.
- [16] New Relic, I. *New Relic*. 2015. Disponível em: <<http://newrelic.com/>>.
- [17] APPDYNAMICS. *Appdynamics*. 2016. Disponível em: <<https://www.appdynamics.com/>>.
- [18] DYNATRACE. *Dynatrace*. 2016. Disponível em: <<http://www.dynatrace.com/>>.
- [19] PINPOINT. *Pinpoint*. 2016. Disponível em: <<https://github.com/naver/pinpoint>>.
- [20] 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.
- [21] 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.
- [22] BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. [S.l.]: Pearson Education India, 2012. 1–426 p. ISSN 03008495. ISBN 0321154959.
- [23] 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>>.
- [24] 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>>.
- [25] 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: <<http://ieeexplore.ieee.org/document/6606651/?arnumber=6606651&tag=1%5Cnhttp://ieeexplore>>.
- [26] 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>.

- [27] 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.
- [28] 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>.
- [29] 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.
- [30] 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/>.
- [31] GALLAGHER, K.; HATCH, A.; MUNRO, M. A framework for software architecture visualization assessment. 2005. Disponível em: <http://ieeexplore.ieee.org/document/1684309/>.
- [32] 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.
- [33] 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/>.
- [34] 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.
- [35] 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>.
- [36] 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.
- [37] 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.
- [38] 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.
- [39] 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.

- [40] 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.
- [41] 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.
- [42] 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.
- [43] 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>>.
- [44] 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>>.
- [45] CORBI, T. A. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, v. 28, n. 2, p. 294–306, 1989. ISSN 0018-8670.
- [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=tZOtx3y1>>.
- [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] WICKET, A. *Wicket*. Tese (Doutorado), 2016. Disponível em: <<https://wicket.apache.org>>.
- [66] NETTY. *Netty*. 2016. Disponível em: <<http://netty.io>>.
- [67] 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.

- [68] 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.
- [69] WETTEL, R.; LANZA, M. Visual exploration of large-scale system evolution. *Proceedings - Working Conference on Reverse Engineering, WCRE*, p. 219–228, 2008. ISSN 10951350.
- [70] 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.
- [71] 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>.

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

Performance QA Evolution

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)

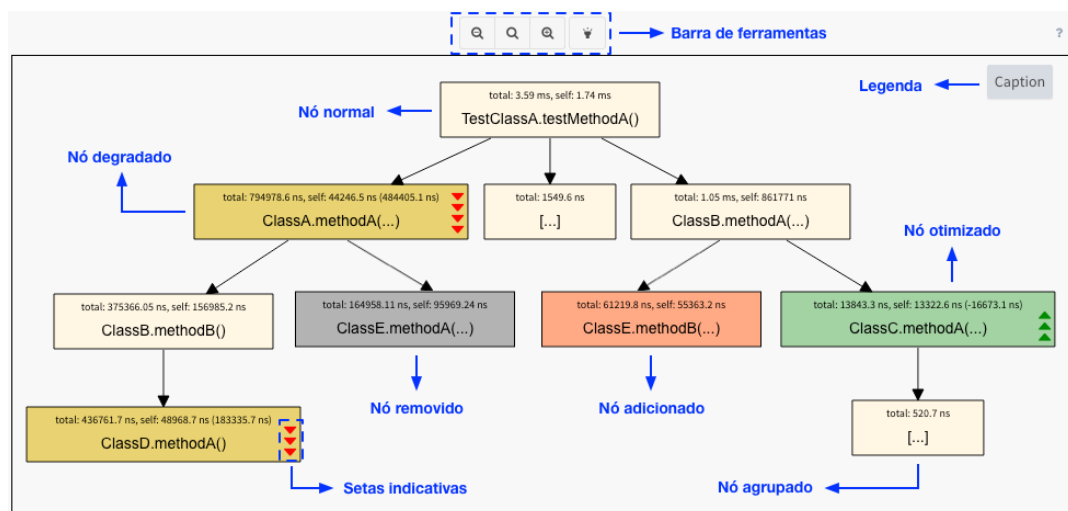
Questão 5. Se você respondeu “Uso essas ferramentas frequentemente” ou “Já usei essas ferramentas” na pergunta anterior, você poderia especificar qual(is) ferramenta(s) você usa ou já usou? (caixa de texto)

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 deviations of performance for two versions of Jetty/VRaptor: <link_para_a_ferramenta>.

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 email 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

Performance QA Evolution

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_a_ferramenta>.

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)

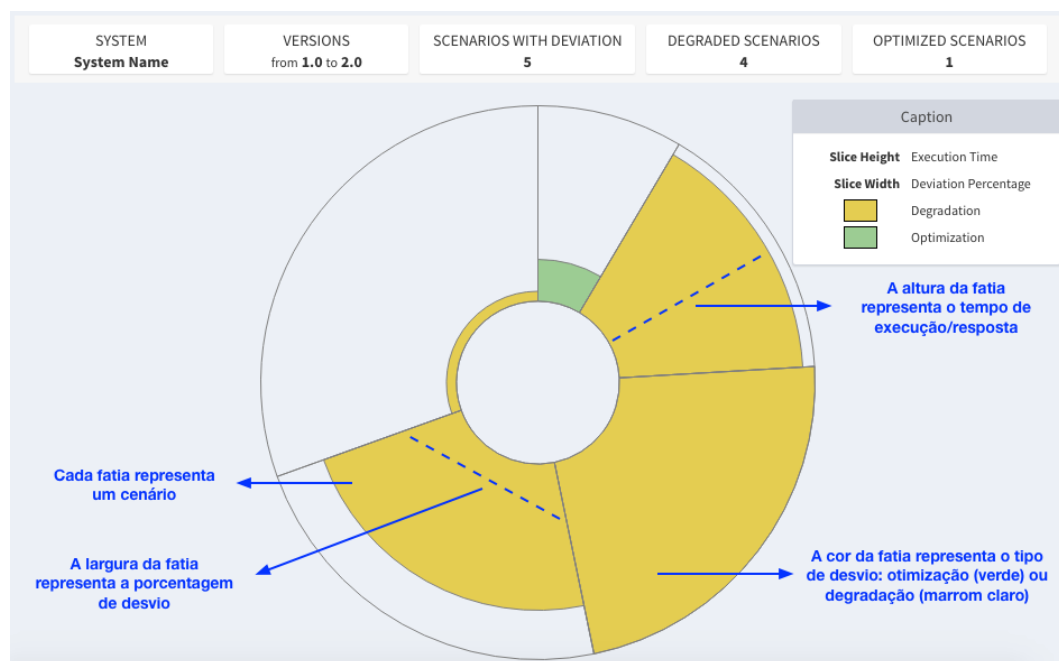
Questão 10. *Quão fácil foi responder à pergunta anterior? (múltipla escolha - escala Likert de 5 itens)*

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 email abaixo. (caixa de texto)

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