UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

THIAGO SOARES FERNANDES

Framework para estimar requisitos não funcionais em aplicações móveis

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Profa. Dr. Érika Fernandes Cota
Co-orientador: Prof. Dr. Álvaro Moreira Freitas

Porto Alegre - RS
2015

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Fernandes, Thiago Soares

Framework para estimar requisitos não funcionais em aplicações móveis [manuscrito] / Thiago Soares Fernandes – 2015.

15 f.:il.

Orientador: Érika Fernandes Cota; Co-orientador: Álvaro Moreira Freitas.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2015.

1.ABNT. 2.Processadores de Texto 3.Formatação eletrônica de documentos. I. Cota, Érika Fernandes. II. Freitas, Álvaro Moreira. III. Título.

AGRADECIMENTOS

Primeiramente, agradeço a Deus pela vida, por ter me sustentado todos os dias e pela oportunidade de conhecer pessoas que me ajudaram não apenas no mestrado, mas também na vida. Agradeço aos meus pais (Edilton e Bárbara) e aos meus irmãos (Valéria, Talitha e Lucas) que sempre me ajudaram e suportaram a minha chatice. Agradeço, também, à minha namorada (Xero) que suportou bravamente a tensão do mestrado comigo. Agradeço, ainda, aos muitos amigos que fiz em Porto Alegre na pousada, na Congregação e no INF, incluindo os professores que ajudaram desde o meu ingresso, como o professor Álvaro, passando pelas "cadeiras" e até a conclusão deste trabalho. Em especial, agradeço à professora Érika por ter sido minha orientadora e amiga, além da paciência que teve comigo.

Um grande abraço a todos!

RESUMO

O desenvolvimento de aplicações móveis é guiado por uma especial atenção aos requisitos não funcionais (do inglês, NFR - Non Functional Requirements), sendo o principal objetivo proporcionar uma boa experiência ao usuário final. Entretanto, a avaliação de NFRs é ainda uma tarefa manual, não estruturada e que consome muito tempo. Esta dissertação apresenta um estudo de várias abordagens relacionadas à avaliação de desempenho (por exemplo, o uso de aplicações de *benchmark*) e de NFRs no âmbito de sistemas móveis. No entanto, os *benchmarks* atuais são genéricos, geralmente, voltados para a plataforma de execução e nem sempre instituem um consenso na classificação de dispositivos. Visando uma melhor avaliação de NFRs e uma classificação de dispositivos com base nas necessidades de aplicações reais, este trabalho propõe um *framework* para gerar *benchmarks* orientados às necessidades de cada aplicação e, assim, fornecer uma forma eficiente e eficaz para estimar requisitos não funcionais em sistemas móveis. Essa ferramenta é composta por uma biblioteca de testes parametrizáveis, métricas e uma estrutura para geração rápida de *benchmarks* orientados à aplicação. O *framework* foi construído utilizando o paradigma de programação orientada a aspectos para coleta das métricas por fornecer uma maior modularidade e separação de interesses, de modo que a sua *evolução*, através da adição de outras métricas ou testes, seja facilitada. Para validação da proposta, foram realizados experimentos com cinco aplicações Android reais disponíveis na *Play Store*, sendo que para cada aplicação foi gerado um *benchmark* específico cujos resultados foram comparados com os obtidos para as aplicações móveis reais. Os resultados são promissores, mostrando que é possível criar aplicações de teste com comportamento semelhante ao de aplicações reais e, assim, classificar dispositivos com base nas necessidades das aplicações, através da análise das métricas presentes no *framework*. Essas métricas podem, ainda, orientar o desenvolvedor na otimização de suas aplicações ou ainda na escolha de dispositivos com melhor custo benefício para executar seus aplicativos.

Palavras-chave: Requisitos não funcionais. Avaliação de desempenho. *Benchmark.* Programação orientada a aspectos. Sistemas Embarcados. Sistemas móveis. Android.

A Framework for Non-functional requirements estimation in mobile applications

ABSTRACT

The mobile application development is guided by a special attention to non-functional requirements (NFRs), where a good experience for the end user is the primary goal. However, NFRs evaluation is still a manual, unstructured and time-consuming task. This thesis presents a study of several approaches related to performance and NFR evaluation within mobile systems. Among these approaches is the use of benchmark applications. Currently available benchmarks are generic, usually focused on the execution platform and do not always establish a consensus on the classification of devices. For a better NFRs assessment and classification of devices based on real application needs, this work proposes a framework for generating application-oriented benchmarks for the early estimation of non-functional requirements in mobile systems. This framework is composed of a configurable test library, a set of metrics and an engine the assembling of the test program. The framework uses aspect-oriented programming to collect the metrics of interest. This approach provides increased modularity and separation of concerns, thus facilitating the improvement of the framework itself, by adding other metrics or testing operations. In order to validate the proposed framework we used five application from the Android Play store. For each application, a specific benchmark is generated and executed in different devices. The results are compared to those of the execution of the actual applications in the same devices. Experimental results are promising, showing that it is possible to create test applications with similar behavior to that of real applications and thus classify devices based on the actual application needs, by analyzing the metrics present in the framework. These metrics can also guide the developer in optimizing her applications or in choosing devices with the best trade-off between cost and performance to run a given application.

Keywords: Non-functional requirements. Performance evaluation. Benchmarks. Aspect-oriented programming. Embedded systems. Mobile systems. Android.

LISTA DE FIGURAS

Figura 1.1 – Camadas de um *software* 14
Figura 2.1 – Execução do *Littleeye* com gráfico de potência consumida. 30
Figura 2.2 – Aparência da ferramenta *NeoLoad* 31
Figura 3.1 – Processo de *clustering* de páginas web. 37
Figura 3.2 - Gráfico da métrica de tempo de CPU para os três primeiros grupos de páginas 39
Figura 3.3 – Gráfico da métrica de tempo de CPU para o grupo 4. 39
Figura 3.4 – Gráfico de uso de memória em relação a versão do sistema operacional 40
Figura 3.5 – Gráfico de uso de memória em relação a interação entre sistema operacional e *heap* 40
Figura 3.6 – Gráfico do tempo total de execução 41
Figura 3.7 – Diagrama de classe da versão base. 42
Figura 3.8 – Diagrama de classe da versão menos coesa. 42
Figura 3.9 – Diagrama de classe da versão mais coesa. 43
Figura 3.10 – Métrica de ocupação de CPU da versão base 44
Figura 3.11 – Métrica de ocupação de CPU versão menos coesa 44
Figura 3.12 – Métrica de ocupação de CPU versão mais coesa 44
Figura 3.13– Métrica de ocupação de memória da versão base 45
Figura 3.14 – Métrica de ocupação de memória da versão menos coesa 45
Figura 3.15 – Métrica de ocupação de memória da versão mais coesa 46
Figura 3.16 – Métrica de consumo de potência da versão base 46
Figura 3.17 – Métrica de consumo de potência da versão menos coesa 47
Figura 3.18 – Métrica de consumo de potência da versão mais coesa 47
Figura 3.19 – Métrica de ocupação de CPU *QuickSort* 48
Figura 3.20 – Métrica de ocupação de memória *QuickSort* 48
Figura 3.21 – Métrica de consumo de potência *QuickSort* 48
Figura 3.22 – Comparação da métrica de ocupação de memória 49
Figura 4.1 – Esquema de criação de um programa de teste 53
Figura 4.2 – Estrutura da biblioteca de testes 55
Figura 4.3 – Definição de elementos da biblioteca 55
Figura 4.4 – Exemplo de interesses transversais. 65
Figura 4.5 – Esboço do arquivo de aspecto para coleta de métricas. 70
Figura 4.6 – Exemplo de estruturação de uma aplicação gerada pelo *framework*. 72
Figura 4.7 – Exemplo do arquivo xml de testes. 74
Figura 4.8 – Diagrama de sequência para operação de envio de *e-mail*. 75
Figura 5.1 – Gráfico de comparação de tempo de execução do Lembrar 78
Figura 5.2 – Gráfico de comparação de consumo de bateria do Lembrar 79
Figura 5.3 – Gráfico de comparação de tempo de execução do Zirco Browser 81
Figura 5.4 – Gráfico de comparação de consumo de bateria do Zirco 82
Figura 5.5 – Gráfico de comparação de quadros por segundo do *Frozen Bubble* 83
Figura 5.6 – Gráfico de comparação de consumo de bateria do *Frozen Bubble* 84
Figura 5.7 – Gráfico de comparação de quadro por segundo GlTron 85
Figura 5.8 – Gráfico de comparação de consumo de bateria no GLTron 86
Figura 5.9 – Gráfico de comparação de tempo para envio de *e-mail*. 87
Figura 5.10 – Gráfico de comparação do consumo de bateria 88
Figura 5.11 – Gráfico de comparação da métrica de Objetos alocados globalmente 88
Figura 5.12 – Trecho de código do método execute na versão com AspectJ. 90
Figura 5.13 – Trecho de código do método execute sem AspectJ. 90

LISTA DE TABELAS

Tabela 1.1 – Comparação de pontuação de dispositivos em três aplicações de *benchmark* 15
Tabela 1.2 – Especificações dos dispositivos usados nos testes 16
Tabela 2.1 – Características dos emuladores Android 32
Tabela 2.2 – Características dos monitores de recurso Android 33
Tabela 3.1 – Tabela de mercado de *smartphones*. 35
Tabela 3.2 – Configurações de emuladores..... 38
Tabela 5.1 – Métrica de tempo de execução em milissegundos..... 92
Tabela 5.2 – Métrica de quantidade de objetos alocados globalmente. 92
Tabela 5.3 – Métrica de tamanho de objetos alocados globalmente em bytes. 92
Tabela 5.4 – Métrica de quantidade de objetos alocados pelo *thread*. 92
Tabela 5.5 – Métrica de tamanho de objetos alocados pelo *thread* em bytes. 93
Tabela 5.6 – Métrica de acionamento do coletor de lixo globalmente..... 93
Tabela 5.7 – Métrica de acionamento do coletor de lixo oriundo do *thread*..... 93

LISTA DE ABREVIATURAS E SIGLAS

ADB	Android Debug Brigde
API	Application Programming Interface
BFQ	Budget Fair Queueing
CFQ	Completely Fair Queuing
CPU	Central Processing Unit
EMMC	Embedded MultiMedia Card
FFT	Fast Fourier Transformation
GPS	Global Positioning System
GPU	Graphics Processing Unit
JNI	Java Native Interfaces
JVM	Java Virtual Machine
NDK	Native Development Kit
PDA	Personal Digital Assistant
POA	Programação Orientada a Aspectos
POO	Programação Orientada a Objetos
QOE	Quality Of Experience
QOS	Quality Of Service
RAM	Random Access Memory
SDK	Software Development Kit
SLA	Service Level Agreement
SMS	Short Message Service
SQL	Structured Query Language
TLB	Translation Lookaside Buffer
URL	Uniform Resource Locator
XML	Extensible Markup Language

Sumário

1 INTRODUÇÃO 13

2 AVALIAÇÃO DE DESEMPENHO EM SISTEMAS MÓVEIS 17

2.1.1 Avaliação de desempenho 17

2.1.2 Benchmarks 20

2.1.3 Qualidade de serviço 24

2.2 Ferramentas para avaliação de desempenho em sistemas móveis 26

2.2.1 Emulador Android 26

2.2.2 AndroProf 27

2.2.3 Genymotion 28

2.2.4 Android Device Monitor 28
2.2.4.1 Dalvik Debug Monitor Server 29
2.2.4.2 Tracer for OpenGL ES 29
2.2.4.3 Hierarchy viewer 29
2.2.4.4 Traceview 29

2.2.5 Littleeye 30

2.2.6 NeoLoad 31

3 ABORDAGEM INICIAL 34

3.1 Android 34

3.2 Avaliação de desempenho para aplicações móveis 35

3.2.1 Objetivo 35

3.2.2 Metodologia 36

3.2.3 Experimentos 38

3.2.4 Resultados 38

3.3 Avaliação de impacto das refatorações 42

4 *BENCHMARKS* ORIENTADOS À APLICAÇÃO PARA SISTEMAS MÓVEIS 52

4.1 Estrutura 53

4.2 Biblioteca de testes 56
4.2.1 Teste de banco de dados 57
4.2.2 Teste de *download* de arquivos 58

4.2.3 Teste de manipulação de arquivos ... 58
4.2.4 Teste de operações de ponto flutuante ... 58
4.2.5 Teste de gráficos em 2D ... 59
4.2.5.1 Desenho de arcos ... 59
4.2.5.2 Desenho de círculos ... 59
4.2.5.3 Desenho de imagens ... 59
4.2.5.4 Desenho de retângulos ... 59
4.2.5.5 Desenho de textos ... 60
4.2.6 Testes de gráficos 3D ... 60
4.2.6.1 Desenho de cubos coloridos com OpenGL 2 ... 60
4.2.6.2 Desenho de cubos com texturas utilizando OpenGL 1.2 ... 60
4.2.6.3 Desenho de cubos com textura utilizando OpenGL 2 ... 60
4.2.6.4 Desenho de motos do Jogo GLTron ... 61
4.2.7 Teste de operações com grafos ... 61
4.2.8 Teste de operações com números inteiros ... 61
4.2.9 Teste de GPS ... 61
4.2.10 Teste de envio de *e-mail* ... 62
4.2.11 Teste de manipulações de memória ... 62
4.2.12 Teste de manipulação de arquivos em C++ ... 62
4.2.13 Teste de operações de ponto flutuante em C++ ... 62
4.2.14 Teste de operações em números inteiros em C++ ... 63
4.2.15 Teste de operações na memória em C/C++ ... 63
4.2.16 Teste de operações com *strings* ... 63
4.2.17 Teste de operações com *strings* em C++ ... 63
4.2.18 Teste de ordenação ... 63
4.2.19 Teste de *streaming* de vídeo ... 64
4.2.20 Teste exibição de vídeo ... 64
4.2.21 Teste de acesso a páginas *web* ... 64
4.2.22 Teste de latência de página ... 64
4.2.23 Teste de consumo de *web service* ... 64
4.2.24 Teste exibição de telas ... 65

4.3 Programação Orientada a aspectos ... 65
4.3.1 AspectJ ... 66

4.4 Métricas ... 69
4.4.1 Métricas coletadas ... 71

4.5 Construção da aplicação de teste ... 73

5 EXPERIMENTOS ... 77

5.1 Experimentos para validação ... 77
5.1.1 Lembrar Beta ... 78
5.1.2 Zirco Browser ... 80
5.1.3 *Frozen Bubble* ... 82
5.1.4 GLTron ... 84
5.1.5 K9 Mail ... 86

5.2 Experimentos com POA e POO ... 89
5.2.1 Resultados experimentais ... 91

6 CONCLUSÃO E TRABALHOS FUTUROS ... 94

BIBLIOGRAFIA ... 96

APÊNDICE <ARQUIVO DE COLETA DE MÉTRICAS COM POA> ... 101

1 INTRODUÇÃO

O conceito de sistemas embarcados está ligado a uma plataforma que combina *hardware* e *software* buscando sempre a máxima eficiência devido às fortes restrições de recursos como energia, área ocupada pelo sistema, memória. Um dos mecanismos para se alcançar o melhor aproveitamento do *hardware* é o desenvolvimento de um *software* específico com propósitos delimitados.

O *software* embarcado está presente no cotidiano de inúmeras formas, desde sistemas aeroespaciais até um simples aparelho de telefonia celular. Esses últimos sofreram evoluções no passar dos anos, melhorando a capacidade de processamento e otimizando o espaço dos periféricos permitindo o surgimento dos "telefones inteligentes" (*smartphones)*. Esses dispositivos trouxeram uma nova experiência para os usuários finais e assim popularizou o seguimento de dispositivos e viabilizou muitas opções em modelos e dispositivos, como *tablets, smartwatches, smartbands*.

Com a popularização, o conceito de sistemas embarcados foi modificado, pois tais dispositivos, apesar de ainda sofrerem restrições, como energia, não possuem mais sistemas de propósito específico. Um *smartphone* pode ser utilizado como despertador, leitor de *e-mails* e de livros, tocador de música e vídeos, bússola, lanterna, *Global Positioning System* (GPS), entre outras funções, pois tais dispositivos executam um sistema operacional que possibilita a instalação de aplicações diversas que têm acesso a recursos do *hardware* do dispositivo.

A mudança de um sistema de propósito específico para um de propósito geral nos sistemas embarcados ampliou a gama de funcionalidades do dispositivo. Entretanto, ainda há restrições no poder de processamento, no tamanho da memória disponível. Enquanto encontramos facilmente *desktops* com opção de 16GB de memória RAM (*Random-access Memory*), os *smartphones* mais atuais não ultrapassam 3GB. Outro problema constante em sistemas móveis é a necessidade de uma fonte de energia, sendo que essa não evoluiu na mesma velocidade dos outros componentes.

O usuário de um dispositivo móvel espera ter uma experiência de uso semelhante àquela observada no uso de *desktop*, mesmo sabendo, ainda que superficialmente, das restrições. Assim, os desenvolvedores de aplicações móveis têm buscado melhorar a experiência do usuário, composta por fatores como: funcionalidade, usabilidade (incluindo acessibilidade, ergonomia, interatividade), utilidade, confiabilidade, resposta a erros e desempenho ou performance.

Normalmente, o conceito de desempenho é relacionado ao tempo de execução e resposta, mas é possível afirmar que, no ambiente de dispositivos móveis, esse conceito está relacionado, também, ao uso eficiente de outros recursos de *hardware*, como memória, bateria, *Graphics Processing Unit* (GPU), tela e sensores. Ainda, há os requisitos não funcionais, características de uma aplicação não estritamente ligadas à realização de uma funcionalidade, mas com a qualidade de execução de uma ou mais funcionalidades, Sommerville (2010).

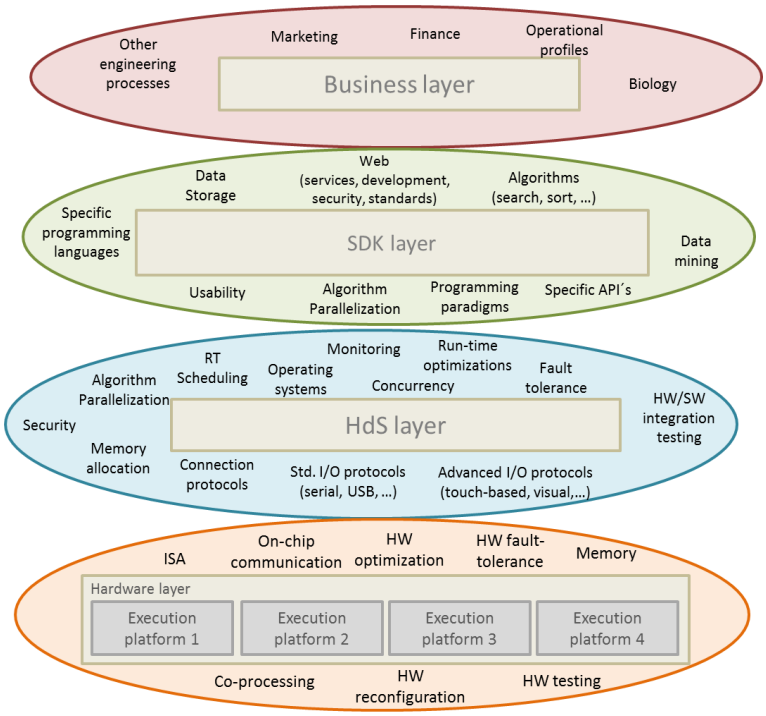
De acordo com Sommerville (2010), requisitos não funcionais de uma aplicação, como o desempenho e o consumo de recursos, podem ser mais importantes do que uma funcionalidade específica. Essa importância é baseada na capacidade de um usuário de superar inadequação de uma funcionalidade em uma aplicação e, no entanto, inutilizar um

sistema devido à insatisfação de um requisito não funcional. Por exemplo, um usuário pode deixar de usar um navegador de internet pelo espaço ocupado na memória do dispositivo.

De modo geral, o desempenho de um *software* pode ser influenciado por elementos de todas as suas camadas, desde as especificidades do modelo de negócio até as características de *hardware* da plataforma de execução. A Figura 1.1 apresenta quatro camadas que constituem um sistema em diferentes níveis de abstração, sendo que uma decisão tomada na camada de negócios influenciará todo o restante do *software* e, consequentemente, trará algum reflexo no desempenho da aplicação.

Do ponto de vista do desenvolvimento de *software*, é conhecido que mudanças no projeto de um *software* são mais baratas quando realizadas no início desse projeto, pois a correção será mais fácil e menos onerosa para o desenvolvedor.

Figura 1.1 – Camadas de um *software*



Do ponto de vista de empresarial, pode ser necessário adquirir dispositivos móveis para utilização como ferramentas de coleta, consulta e manipulação de dados em campo. Mas, recomenda-se uma aquisição baseada em pesquisa de necessidades do usuário para que a escolha do dispositivo não seja equivocada, gerando prejuízos. Cabe ressaltar que a aquisição de um dispositivo muito caro pode ser substituída por um modelo com melhor custo benefício, mas que ainda atenda corretamente às funcionalidades e tenha o desempenho necessário.

Apesar da importância dos requisitos não funcionais refletidos no desempenho das aplicações (observado pelo usuário final), o seu processo de avaliação é ainda manual, *ad-hoc* e exige muito tempo do desenvolvedor. Os atuais esforços para avaliar esses requisitos estão focados, principalmente, nas plataformas de execução e não nas necessidades reais das aplicações, sendo a forma mais comum o desenvolvimento de *benchmarks*, aplicações

sintéticas em sua maioria direcionados ao processador de um dispositivo. A partir da utilização dess*a solução* pode-se comparar duas ou mais plataformas com relação às funcionalidades exercitadas. Entretanto, uma aplicação real tem requisitos não funcionais muito distintos dos códigos exercitados quando rodamos uma aplicação de *benchmark*.

Muitos *benchmarks* de dispositivos estão disponíveis nas lojas oficias das diversas plataformas (*Windows Phone, IO*S e *Android*) e possuem, de modo geral, o intento de avaliar, gerar uma pontuação e, posteriormente, comparar essa pontuação com outros dispositivos no mercado. No entanto, tais aplicações possuem mecanismos e metodologias distintas para classificar os dispositivos avaliados. Em alguns casos essa diferença no processo de avaliação pode resultar em classificações distintas para um mesmo conjunto de dispositivos. A Tabela 1.1 contém a pontuação que permite a classificação de quatro smartphones que utilizam o sistema operacional *Android*. As especificações detalhadas dos dispositivos podem ser visualizadas na

Tabela 1.2.

Tabela 1.1 – Comparação de pontuação de dispositivos em três aplicações de *benchmark*

Dispositivos / App	Quadrant	Antutu	Vellamo
Samsung Galaxy 551	624	NA[1]	93
Sony Xperia U	2766	9859	372
Motorola Atrix[2]	2288	11485	330
Samsung Galaxy S3	4673	22656	568

Fonte: Próprio autor.

Observando a Tabela 1.1, percebemos que os *benchmarks* não fornecem uma classificação precisa do dispositivo ou informam qual desses é mais o adequado a uma determinada aplicação. Outra desvantagem é a não disponibilização de informações sobre como e quais testes são realizados.

Mediante esse cenário onde existe a necessidade de se estimar requisitos não funcionais de aplicações móveis de forma rápida e possibilitando a realização de mudanças de uma aplicação móvel no início do seu projeto, esse trabalho busca: i) avaliar as ferramentas já existentes para avaliação de desempenho; ii) averiguar possíveis relações entre a estrutura do código de uma aplicação móvel e seu desempenho; e, iii) propor ferramentas que auxiliem o desenvolvedor nas decisões iniciais no projeto da aplicação, bem como as empresas na escolha de dispositivos que atendam suas reais necessidades e, assim, economizarem recursos.

[1] Esse dispositivo não é compatível com a aplicação.

[2] Nessa avaliação o dispositivo avaliado utilizava a versão 2.3.6 do Android.

Tabela 1.2 – Especificações dos dispositivos usados nos testes

Dispositivo	SO	Chipset	CPU	GPU	Memória	Wlan
Galaxy 551 (Dev 1)	Android 2.3.6	Qualcomm MSM7227	600 MHz	Adreno 200	280 MB	Wi-Fi 802.11 b/g/n
Xperia U (Dev 2)	Android 4.4.4	NovaThor U8500	Dual-core 1 GHz	Mali-400	512 MB	Wi-Fi 802.11 b/g/n
Atrix (Dev3)	Android 4.0.4	Nvidia Tegra 2 AP20H	Dual-core 1 GHz	ULP GeForce	1GB	Wi-Fi 802.11 a/b/g/n
Galaxy S3 (Dev 4)	Android 4.4.4	Exynos 4412 Quad	Quad-core 1.4GHz	Mali-400MP4	1 GB	Wi-Fi 802.11 a/b/g/n

Fonte – Próprio autor.

O texto está organizado da seguinte maneira: no Capítulo 2 é abordada a avaliação de desempenho em dispositivos móveis, com apresentação de trabalhos e ferramentas correlacionados ao tema e ao final a motivação desta dissertação; o Capítulo 3 apresenta a abordagem inicial utilizada neste trabalho para auxiliar o desenvolvedor de aplicações móveis; no Capítulo 4 é apresentada a proposta do *framework* para gerar *benchmarks* orientados à aplicação para sistemas móveis; no Capítulo 5 são apresentados os experimentos realizados para validar o *framework* proposto e no Capítulo 6 é apresentada a conclusão do trabalho e as atividades a serem realizadas em trabalhos futuros.

2 AVALIAÇÃO DE DESEMPENHO EM SISTEMAS MÓVEIS

A avaliação de sistemas computacionais, quanto à satisfação de requisitos não funcionais, tem sido explorada na literatura e muitas técnicas têm sido reportadas para definir o processo de avaliação, as quais são baseadas em conceitos distintos como *benchmarking* e QoS (do inglês, *Quality of Service*). Esses conceitos estão bem estabelecidos para computadores pessoais e alguns domínios de aplicação, como por exemplo, aplicações do tipo *benchmark* são comumente usadas para a avalição de desempenho de microprocessadores. Já QoS é usualmente estabelecido como meio de avaliação para estabilidade e confiabilidade de redes de telecomunicações.

No entanto, um método efetivo para avaliação de requisitos não funcionais para sistemas móveis, principalmente em relação aos *software*s, ainda não foi estabelecido. De fato, *benchmarks* têm sido usados para a criação de *rankings* de dispositivos em relação a alguns parâmetros pré-definidos Uti; Fox (2010) e Kim; Kim (2012) principalmente quanto à CPU (do inglês, *Central Processing Unit*), memória e uso de bateria. Nesse caso, o usuário realiza o *download* da aplicação, executa-a e compara os resultados obtidos nessa execução com os resultados de outros dispositivos disponíveis para a mesma aplicação. No entanto, como será detalhado à frente, essa abordagem não provê muitas informações que poderiam ser úteis a desenvolvedores visando aperfeiçoarem suas aplicações.

As seções a seguir apresentam trabalhos que discutem métodos para avaliar o desempenho de sistemas móveis, pois é notória a importância de avaliar requisitos não funcionais e, consequentemente, o desempenho de aplicações móveis implicando na aceitação ou não de um aplicativo.

2.1.1 Avaliação de desempenho

A avaliação de desempenho de um sistema (*software* ou *hardware*) tem por objetivo obter dados relacionados ao desempenho em um determinado ambiente ou cenário de execução. Essa avaliação deve ter um foco bem definido, além de um razoável conhecimento do sistema sob avaliação, de modo que os dados coletados sejam corretamente relacionados aos aspectos do sistema analisado. De acordo com Jain (1991), a avaliação de desempenho é uma arte, isto é, apesar da existência de procedimentos, fórmulas e métodos, a avaliação não consiste na aplicação de passos pré-determinados, mas em um processo guiado por objetivos.

Os trabalhos desenvolvidos para realizar a avaliação de desempenho em sistemas móveis podem ser classificados em dois grupos: 1) análise de consumo de recursos e seu impacto no desempenho das aplicações e do sistema operacional; 2) comparação de desempenho de código de aplicações móveis produzidos com diferentes linguagens de programação e suas respectivas características. Os quatro primeiros trabalhos discutidos pertencem ao primeiro grupo e os três últimos ao grupo dois.

Kim et al. (2011) avaliam o impacto do sistema de armazenamento do dispositivo no desempenho de aplicações móveis. O estudo de caso foi realizado em dois modelos de *smartphones* (Google Nexus One e HTC One) e mostrou que operações de escrita e leitura de arquivos afetam o desempenho da maioria das aplicações, mesmo aquelas interativas nas

quais poucas operações de armazenamento são tipicamente esperadas, como um navegador. Também, ficou evidenciado que o baixo desempenho do sistema de armazenamento do dispositivo acarretará uma sobrecarga no processador, ocasionando um aumento no consumo de energia.

(KAYANDE; SHRAWANKAR, 2012) afirmam que, devido à escassez de memória em dispositivos móveis, um sistema operacional para esses dispositivos precisa ter um bom gerenciamento daquele recurso. Essa afirmação também pode ser confirmada pelas discussões sobre melhores práticas para aumentar o desempenho geral do sistema, realizadas em fóruns de desenvolvimento, conforme Caellum (2002). Os autores defendem que uma baixa ocupação de memória RAM e um tamanho de código reduzido são boas características em uma aplicação móvel. De acordo com os pesquisadores, apesar de ser baseada em Linux, a plataforma Android não dispõe dos mesmos mecanismos de gerenciamento de memória do sistema operacional original. Nesse trabalho, é proposta uma aplicação para gerenciamento de mensagens SMS (do inglês, *Short Message Service*) (SMSLingo) e uma posterior comparação com similares existentes na loja de aplicativos Android, além da aplicação padrão disponível no sistema operacional. A métrica utilizada foi a ocupação de memória, sendo que o SMSLingo reduziu a necessidade de ocupação da memória em, aproximadamente, 50% em comparação com as demais aplicações. No entanto, a pesquisa não evidenciou quais estratégias de desenvolvimento possibilitaram essa redução e não forneceram informações sobre outras métricas comuns, como tempo de execução ou consumo de energia.

Em Wang et al. (2012) o objetivo foi avaliar um conjunto de seis fatores e determinar qual tem maior influência no desempenho de aplicações móveis. O trabalho avaliou os seguintes fatores, de forma independente: frequência do processador; número de núcleos no processador; tamanho de memória cache; tamanho da memória principal; sistema de armazenamento e versão do *kernel* do sistema operacional. A técnica utilizada para ponderar os fatores em relação ao seu impacto no desempenho foi a análise de autovalor, apresentada no mesmo trabalho. Baseando-se nos pesos atribuídos a cada fator, por essa técnica, os autores concluíram que atualizar a versão do *kernel* e alterar os mecanismos do sistema de armazenamento são práticas mais importantes para melhorar o desempenho das aplicações do que os demais fatores.

Em Nguyen (2013), o autor aborda o impacto de determinadas configurações do sistema de armazenamento no desempenho das baterias de dispositivos Android. O estudioso afirma ser possível adaptar componentes do sistema operacional usados nas operações de escrita e leitura, tais como o algoritmo de escalonamento e a profundidade da fila, de acordo com os requisitos da aplicação para economizar energia. Para validar sua afirmação, foi realizado um estudo de caso com as vinte aplicações mais populares da *Play Store*, executando-as em oito possíveis configurações de um aparelho Nexus One. Cada configuração representa uma combinação diferente entre profundidade da fila (4, 128) e o algoritmo de escalonamento (BFQ, CFQ, Deadline e Noop). Os resultados mostram que, para a aplicação *Gmail*, por exemplo, a combinação mais econômica preservou cerca 21% da bateria em comparação com o consumo apresentado na versão menos econômica. Já para a aplicação Pandora, a economia foi de até 52% da bateria no Nexus One. Apesar de mostrar essa economia, o autor não abordou se existe um padrão que determina qual configuração é mais adequada a outras aplicações que não foram usadas nos testes.

O trabalho de Lee; Jeon (2010) visa comparar o desempenho em relação ao tempo de execução de aplicações desenvolvidas puramente em JAVA utilizando SDK (do inglês, *Software Development Kit*) do Android e os mesmos algoritmos, mas desenvolvidos em C utilizando o NDK (do inglês, *Native Development Kit*). O código nativo (C) e a aplicação

Android foram ligados através de *Java Native Interfaces* (JNI). As aplicações utilizadas nos experimentos foram: algoritmo para calcular o enésimo número primo (operações com inteiros); cálculo de seno/cosseno (operações com ponto flutuante); algoritmo de ordenação *bubblesort* (operações de acesso a memória) e multiplicação de uma entrada do usuário por 4Kbytes para determinar o tamanho de *heap* alocado. Os resultados dos experimentos realizados mostraram que, em relação a cálculos com números inteiros, acesso à memória e alocação de *heap*, as aplicações desenvolvidas em C tiveram um tempo de execução consideravelmente menor do que suas similares desenvolvidas em JAVA. No entanto, quanto às operações com números de ponto flutuante, a diferença não foi significativa, não ultrapassando 18 milissegundos. Dessa maneira, é importante frisar que a utilização do código nativo não necessariamente garante um melhor desempenho para uma aplicação, sendo as características da aplicação que definem se o código nativo deve ser utilizado.

Em Lin et al. (2011) também são comparadas aplicações JAVA e aplicações desenvolvidas em C/C++ utilizando NDK e JNI. O conjunto de códigos avaliados é formado pelas seguintes categorias, definidas pelos autores: cálculos numéricos com recursão (Ackermann e Fibonacci); uso de bibliotecas (Hash e Hash2); manipulação de estruturas de dados (Heapsort e Matrix); polimorfismo (Methcall e Objinst); laços aninhados (NestedLoops); geração de números aleatórios (Random); cálculo de números primos (Sieve of Erastosthenes) e operações com strings (Strcat). Para realizar os experimentos foi utilizado um dispositivo HTC Desire, executando todos os códigos e avaliando o tempo de duração de cada execução. Os resultados mostraram que em sete algoritmos o código desenvolvido em C/C++ é aproximadamente 34% mais rápido do que o código JAVA. Os autores afirmam que o desempenho do algoritmo de Ackermann é muito ruim na versão em JAVA, mas não fornecem dados de tempo, pois com a entrada utilizada o algoritmo retornou uma exceção de estouro de pilha. No entanto, para três dos algoritmos utilizados (Hash, Random e Heapsort) o código em JAVA foi mais rápido do que o C/C++. Exceto pela categoria de polimorfismo, que continha dois algoritmos e ambos foram melhores quando desenvolvido em C/C++, as demais não têm um consenso sobre qual linguagem deverá ter um melhor desempenho. É importante ressaltar que se observarmos o algoritmo de ordenação usado nesse trabalho (Heapsort) ele apresenta um resultado distinto do trabalho de Lee; Jeon (2010) que utilizou o bubblesort. Tal fato demonstra não ser possível dizer que em uma categoria sempre teremos o melhor desempenho de uma só linguagem, pois o melhor desempenho depende de como o algoritmo é implementado.

Em Kim et al. (2012) são comparadas aplicações Android desenvolvidas em C e acionadas via JNI e aplicações compiladas utilizando o CodeSourcery[3], um *cross-compiler* nativo para ARM. Nesse trabalho, o conjunto de códigos avaliados é formado por um subconjunto do *benchmark* para processadores embarcados Mibench, apresentado na seção 2.1.2. Os códigos avaliados são: Basicmath, Qsort, Dijkstra, Patricia, FFT e StringSearch, sendo os experimentos realizados em uma Nvidia Tegra com Android 2.2, CPU Cortex A9 1GHz e 1GB de memória RAM. Em cinco dos algoritmos avaliados o código C acionado através de JNI teve um menor tempo de execução quando comparado ao código compilado pelo *cross-compiler*. Este resultado, segundo os autores, deve-se ao JNI do Android NDK utilizar a biblioteca Bionic C/C++ ao invés da GNU C. No caso do algoritmo de Dijkstra o código acionado via JNI é mais lento e esse comportamento é atribuído, de acordo com os pesquisadores, por esse algoritmo gerar mais instruções de *load* de memória que os demais.

[3] http://www.mentor.com/embedded-software/codesourcery

Assim, os estudiosos orientam pela utilização da biblioteca bionic C/C++ em aplicações Android, por ser otimizada para dispositivo móveis, ao invés da biblioteca GNU C.

Os trabalhos relacionados nessa seção apresentaram uma variedade de metodologias para avaliar o desempenho em sistemas móveis, dificultando o estabelecimento de um padrão de avaliação. Mas essa variedade fornece também uma visão geral da quantidade de elementos que podem compor o processo de avaliação de requisitos não funcionais em sistemas móveis, por exemplo, a linguagem escolhida para desenvolvimento e como essa pode afetar o desempenho final do *software*.

2.1.2 Benchmarks

O uso de *benchmarks* para comparar sistemas computacionais é um método estabelecido no campo de avaliação de desempenho. Uma suíte de *benchmark* é uma coleção de aplicações reais ou sintéticas, que são usadas para exercitar algum elemento de uma plataforma ou sistema com o objetivo de simular o uso de um ou mais recursos intensivamente. Utilizando os dados obtidos a partir desses exercícios, é possível comparar diferentes sistemas. O processo de utilizar essas aplicações é chamado *benchmarking*, Jain (1991).

Algumas suítes para sistemas *desktop* como SPEC2006[4] são conhecidas para estressar CPUs, assim como outras para redes e GPUs (do inglês, *Graphics Processing Unit*), além daquelas para sistemas computacionais, como gerenciadores de bancos de dados, navegadores de internet. Para dispositivos embarcados, o *Embedded Microprocessor Benchmark Consortium* (EEMBC) possui um catálogo de *benchmarks* proprietários para diversos tipos de aplicações embarcadas, desde sistemas automotivos até os atuais dispositivos móveis. Essas suítes focam na avaliação das plataformas móveis de execução e normalmente não disponibilizam detalhes do código executado pelo *benchmark*, dificultando assim o entendimento de como essas plataformas estão sendo avaliadas.

Para usuários de dispositivos móveis, as lojas oficiais como *Play Store, Windows Phone Store* e *App Store* oferecem aplicações que realizam *benchmarking* e permitem ao usuário comparar diferentes dispositivos. Por exemplo, em meados de 2013, a *Play Store* possuía, aproximadamente, 70 aplicações classificadas como *benchmark*. Esse conjunto de aplicações cobre diferentes elementos de *hardware* dos dispositivos, pois, de modo geral, são avaliados a CPU, GPU, memória e armazenamento de dados. O usuário dessas aplicações, geralmente gratuitas, mas de código proprietário, deve baixá-las, executá-las e comparar o desempenho do seu dispositivo com outros através de um *ranking* global.

Embora sejam úteis para o usuário final, essas aplicações de *benchmark* normalmente não oferecem muitas informações relevantes dos testes realizados para os desenvolvedores de aplicações móveis. Os trabalhos discutidos a seguir apresentam soluções desenvolvidas no meio acadêmico e que envolvem o uso ou a definição de *benchmarks* para o ambiente de aplicações móveis.

A suíte Mibench proposta por Guthaus et al. (2001) é uma solução conhecida no mercado de sistemas embarcados, sendo que o conjunto de testes foi desenvolvido para prover uma opção de *benchmark* de código aberto. O Mibench tem por objetivo representar aplicações embarcadas típicas com a utilização de algoritmos implementados na linguagem C e organizados em seis grupos. Cada grupo é direcionado a uma área específica dessa

[4] https://www.spec.org/cpu2006/

plataforma: dispositivos embarcados automotivos, redes, aplicativos para escritório, segurança, telecomunicações e dispositivos pessoais. Essa solução é focada em avaliar os processadores de dispositivos embarcados em cada uma dessas áreas. No entanto não avaliam outros elementos dos atuais dispositivos móveis.

Lee et al. (1997) propuseram o MediaBench, uma solução de *benchmark* direcionada à avaliação de operações de comunicação e multimídia em processadores embarcados, composto por 19 aplicações que exercitam o poder de processamento para comunicação, processamento de imagens e operações DSP (do inglês, *Digital Signal Processing*). Especificamente, quatro métricas são consideradas nessa solução: instruções-por-clock; taxa de acerto na cache de instruções, taxa de acerto na cache de dados e utilização do barramento de memória. Essa solução é conhecida no mercado de sistemas embarcados, mas não tem foco nos atuais dispositivos móveis que possuem muitos sensores e podem ter suas capacidades aumentadas a partir da utilização de aplicações móveis.

Chen et al. (2002) propuseram uma solução de *benchmark* para sistemas embarcados que utilizam sistemas JAVA. Esse sistema é composto por doze aplicações sintéticas para simular o uso de atividades comuns a PDAs (do inglês, *Personal Device Assistant*), a saber: calculadora, navegador de internet, visualizador de imagem, aplicação para *streaming* de vídeos, jogo de xadrez, jogo *puzzle*, aplicação de exibição de vídeos, jogo 3D em primeira pessoa, aplicativo de envio de *e-mail*, agenda de compromissos, aplicativo de mapas, um aplicativo de desenho de bolas coloridas na tela. O objetivo desses *benchmarks* é avaliar a JVM (do inglês, *Java Virtual Machine*) implementada pela empresa produtora de dispositivo através das seguintes métricas: quantidade de chamadas de métodos, tempo de espera por ação do coletor de lixo, tempo de vida de objetos, alocação de memória e gerenciamento de *threads*. Além disso, dados sobre o comportamento e o tamanho dos objetos visam auxiliar os desenvolvedores das JVMs no dimensionamento do tamanho do *heap* assim como a política do coletor de lixo, beneficiando a execução das aplicações e proporcionando economia de energia para gerenciamento do *heap*. A partir da execução dessas aplicações, os autores apontam que essas possuem um uso frequente de bibliotecas, fato evidenciado pela frequência relativamente maior de invocações dos métodos das bibliotecas do que métodos da aplicação propriamente dita.

Schoeberl et al. (2010) propuseram um *benchmark* de código-aberto, JemBench, no qual o conjunto de testes que compõem essa solução é dividido em códigos de execução sequencial ou *multithread*. Com relação aos códigos sequenciais, o JemBench é subdividido em micro *benchmarks*, *benchmarks* de aplicações e de *kernel*. O micro *benchmark*, avalia implementações básicas da máquina virtual e contém código para exercitar ações como: operações aritméticas; acesso a *array*; acesso a classes; invocações e execuções de métodos; checagem de tipos e sincronização de bloco. Os *benchmarks* de aplicação contêm adaptações de soluções codificadas em JAVA para a indústria Schoeberl (2008): Kfl (controle de vagões), Lift (controle de elevador) e UdpIp (transmissões de rede).O subconjunto de *kernel* possui dois algoritmos: *bubblesort* e *Sieve of Erastosthenes*. A avaliação *multithread* é composta por códigos paralelos, multiplicação de matrizes e implementação do problema de N-rainhas e códigos de *streaming*, implementação do algoritmo AES de criptografia. Nesse trabalho, os autores não apresentaram estudo de caso com execução da solução proposta.

Uti; Fox (2010) desenvolveram um *benchmark* para avaliar CPU de dispositivos móveis. O conjunto de códigos é composto por algoritmos baseados em operações aritméticas (adição, multiplicação, divisão e raiz quadrada) com números inteiros e de ponto-flutuante. Os autores avaliam ainda o impacto nas operações matemáticas ao se utilizar posições de *array* como operandos e a influência das otimizações dos compiladores. O estudo de caso foi realizado

com diferentes dispositivos (Samsung Omnia, Samsung Epic e HTC Touch Pro) e os experimentos mostraram resultados normalmente esperados como as otimizações do CPU reduzindo significantemente o tempo de execução. As operações baseadas em operadores do tipo *array* foram consideravelmente mais lentas do que aquelas não utilizando *arrays*. Foi mostrado, também, que operações de ponto-flutuante são mais caras do que operações com números inteiros. Como esperado, as operações de raiz quadrada, que foram realizadas apenas com números ponto-flutuante, são mais demoradas do que as demais e houve pequenas diferenças entre as demais operações, sendo que a multiplicação e adição apresentaram resultados próximos em alguns casos. Esses dados podem ser explicados, segundo os autores, pelo uso do *pipeline superscalar* dos processadores presentes nesses dispositivos.

O Androbench foi proposto por Kim; Kim (2012) para avaliar o armazenamento de dados em dispositivos Android. Essa aplicação contém operações de leitura e escrita de arquivos e também operações no sistema de banco de dados SQLite. O estudo de caso foi realizado com um grande conjunto de aparelhos (124), pois, após o desenvolvimento da aplicação, a mesma foi disponibilizada na loja de aplicativos, avaliando *throughput* de escrita e leitura de arquivos e *throughput* de operações SQL (do inglês, *Structured Query Language*). Também foi analisado o impacto de diferentes versões do sistema de arquivos e do eMMC (do inglês, *Embedded MultiMediaCard*) em um mesmo dispositivo. Para o sistema de arquivos Ext4 as leituras e escritas sequencias são mais eficientes do que no sistema de arquivo YAFFS2, sendo que nas leituras e escritas aleatórias o YAFFS2 é mais rápido. Para mudanças do eMMC, não ficou clara a contribuição, pois nesse experimento foi utilizado um mesmo dispositivo (GT-I9000) que possui dois eMMCs e não foi possível distinguir o desempenho de cada um deles. O estudo não apontou um dispositivo que seja superior em todas as métricas. No entanto, pontualmente, os autores afirmam que o Samsung Galaxy S2 teve o melhor desempenho em leitura sequencial de arquivos, o Samsung Galaxy Tab foi o melhor na escrita sequencial de arquivos e o HTC Desire apresentou resultados superiores em todas as operações de banco de dados.

O trabalho de Gutierrez et al. (2011), avalia a capacidade de resposta dos dispositivos nos cenários de uso comuns de usuários os quais foram representados pela seleção de aplicações como jogos, aplicações de execução de áudio e vídeo, além de um simples navegador de internet. As métricas avaliadas são: desempenho da cache de instrução; taxa de erro da TLB (do inglês, *Translation Lookaside Buffer*) e erro de predição de *branch*. Os resultados apontaram que as aplicações interativas têm valores ruins nas métricas avaliadas e os autores atribuem à alta abstração utilizada no desenvolvimento de tais aplicações como causa para esse pior desempenho.

O trabalho desenvolvido por Kim et al. (2013) utiliza HTML5 e JavaScript para avaliar dispositivos móveis em relação à métrica de tempo de execução. A variedade de linguagens utilizadas pelas plataformas de dispositivos móveis foi a motivação para utilização de tecnologias que sejam compatíveis com o maior número de dispositivos. O conjunto de testes, que foi implementado em JavaScript, é formado por operações para CPU, memória e combinação desses componentes; já os testes realizados com HTML5 têm por objetivo avaliar páginas *web* textuais e com multimídia. Foi conduzido um estudo de caso para mostrar que o resultado dos testes em JavaScript são semelhantes ao de aplicações nativas desenvolvidas em C. Utilizando de um *Ratio Game*, técnica apresentada em Jain (1991), os autores comparam quatro plataformas (Nexus S, Iphone 4, Galaxy Tab e Tegra 250) e, nos resultados coletados, o Galaxy Tab mostrou-se a plataforma mais eficiente, com um menor tempo de execução dos testes.

Punniyakotti (2012) propôs o BASCin, aplicação Android para avaliar dispositivos com base nas necessidades do usuário final, tomando como dados de entrada para a avaliação aplicações frequentemente utilizadas por aquele usuário no dispositivo. O autor aborda que os *benchmarks* não deveriam apenas classificar os dispositivos com base em sua configuração, mas baseado nas necessidades do usuário final as quais, nesse trabalho, são fornecidas através da frequência de utilização de algumas aplicações. Essas informações são enviadas a um servidor que as analisa e fornece uma lista de possíveis dispositivos ordenados por algum critério definido pelo usuário, possivelmente sendo: desempenho de processamento, consumo de energia, duração da bateria e a combinação entre desempenho e duração da bateria. Esse trabalho mostra a necessidade do alinhamento das reais necessidades de uso de um aparelho com os testes realizados por um processo de avaliação. No entanto, foram usadas aplicações completamente desenvolvidas como parâmetros para indicar as necessidades de recursos, porém ocorre que essas aplicações podem ter muitas funcionalidades extras às reais necessidades do usuário final, podendo influenciar a lista de dispositivos oferecidos.

No trabalho de Lee et al. (2012) é apresentado o AM-Benchmark, solução que visa avaliar dispositivos Android em relação ao seu desempenho ao processar conteúdo multimídia. O conjunto de serviços avaliados é composto por seis aplicações: leitor de e-*book*, leitor de código de barras (QR *code*), aplicativo de câmera, aplicativo de gerenciamento de imagens, além de aplicações de processamento 2D e 3D que compõem a biblioteca libGDX, Zechner (2013). Um estudo de caso foi realizado com a execução de todas as aplicações em quatro plataformas (Nvidia Tegra 250, Galaxy Tab 7, Galaxy Player e Nexus One), tendo por métricas apenas tempo de execução e taxa de frames por segundo. Nas aplicações de uso intenso da CPU, de acordo com os autores, (leitor de e-*book*, QR *code*, operações de matrizes da libGDX, aplicativo de gerenciamento de imagens) a Nvidia Tegra teve o melhor desempenho na média geral. Esse resultado advém da configuração mais robusta dessa plataforma, possuindo um processador *multicore* e com *clock* mais alto. Com relação a métrica de frames por segundo, a Nvidia Tegra também apresentou o melhor desempenho, no cenário geral, devido a sua GPU superior aos demais dispositivos.

Antochi et al. (2004) propuseram uma aplicação de *benchmark* para processamento de imagens 3D de mapas, tanto de área abertas quanto de ambientes fechados, o GraalBench. Os autores buscam apresentar uma melhor forma de avaliar dispositivos móveis em relação a capacidade de processamento de imagens 3D. A carga de trabalho é composta por jogos 3D para exploração de mapas de áreas fechadas e aplicações de mapa de cidades. Os experimentos mostraram que o conjunto de aplicações selecionado conseguiu exercitar com intensidade distinta a GPU do dispositivo, o que os autores consideram como uma vantagem por possibilitar uma maior cobertura de teste. No entanto, os pesquisadores não apresentaram comparação do seu *benchmark* com outras soluções já existentes para comprovar essa melhor cobertura.

Todos esses trabalhos mostram que detalhes da arquitetura do dispositivo podem influenciar significativamente o desempenho de uma aplicação móvel bem como diferentes aspectos da experiência do usuário (tempo de bateria, tempo de resposta), e que decisões de projeto, na construção do aplicativo móvel, devem estar relacionadas ao dispositivo onde o aplicativo será executado para que o usuário obtenha uma boa experiência de uso. As informações resultantes dessas soluções atendem, porém, ao fabricante do dispositivo ou ao usuário final, mas não respondem as principais questões do desenvolvedor. Como os resultados são sempre baseados em um conjunto relativamente pequeno de dispositivos e aplicações (em comparação com o número total de modelos de aparelhos e tipos de aplicações possíveis), o desenvolvedor ainda não tem, a partir desses resultados, uma ideia clara sobre

como será o desempenho geral de sua aplicação em uma dada plataforma. De fato, alguns dos resultados podem ser considerados conflitantes, como por exemplo, um autor sinaliza que o uso de uma biblioteca específica do C é o ideal enquanto outro trabalho não apresenta grande diferença de desempenho entre códigos implementados em Java e em linguagem nativa.

2.1.3 Qualidade de serviço

De acordo com Ferguson; Huston (1998), o conceito de QoS é bastante debatido, uma vez que os termos qualidade e serviço são muito vagos e ambíguos, gerando uma variedade de definições. De fato, QoS é utilizado em várias áreas com significados relacionados, mas pouco distintos. Esse termo é usado no campo de comunicação há bastante tempo e significa o desempenho geral de uma rede, principalmente da perspectiva do usuário final. De modo mais recente, QoS tem sido adotado pela área de redes *on-chip* designando a capacidade de uma rede intra-chip de fornecer um esperado *throughput*, latência ou largura de banda na comunicação entre núcleos embarcados. Para sistemas embarcados, distribuídos e de tempo real, QoS implica no melhor uso dos recursos do sistema para satisfazer requisitos como tempo, confiabilidade e segurança.

Para sistemas móveis, o termo QoS é geralmente associado a qualidade de serviços de telecomunicações em relação ao desempenho, mas também confiabilidade e usabilidade. Por isso, o termo está mais relacionado à infraestrutura de rede do que as aplicações e dispositivos.

Em geral os trabalhos de QoS em sistemas móveis mantêm a relação com aspectos de conectividade e comunicação. As abordagens apresentadas a seguir têm por foco compartilhar recursos pela rede ou migrar tarefas entre servidores e dispositivos locais, pois, assim, a qualidade de serviço é utilizada para avaliar o benefício de executar uma tarefa localmente ou migrá-la para um servidor na nuvem. De maneira mais genérica, a qualidade de serviço também é utilizada para monitorar alguns requisitos não funcionais de componentes e selecionar aqueles que atendem aos requisitos necessários para compor uma aplicação móvel.

Em seu trabalho, Wac (2005) aponta o aumento de pesquisas que abordam o uso de níveis de QoS para determinar a escolha de redes para transmissão de dados no contexto de aplicações móveis. O estudo de caso é contextualizado com um paciente crônico com epilepsia cujos dados são monitorados por uma central de serviço de saúde, transmitidos via rede sem fio e, com base na QoS das redes disponíveis, é selecionada a melhor rede e também qual conjunto de dados pode ser transmitido. Assim, caso a rede indicada tenha ainda uma baixa qualidade, o volume de dados a serem transmitidos é reduzido para que a transmissão seja efetiva. A proposta de Wac (2005) é que a cada transmissão de dados, informações de QoS da rede e de localização sejam armazenados para que, futuramente, essas informações sejam usadas para predizer a rede ideal a ser selecionada.

Wac et al. (2011) realizaram uma pesquisa com um conjunto de usuários de *smartphones* para relacionar a QoE (do inglês, *Quality of Experience*) do usuário e a QoS da rede de dados. O estudo contou com vinte e oito voluntários que permitiram que seus dados de uso do dispositivo (tipos de rede de dados, dados do acelerômetro e aplicações usadas) fossem coletados, além de avaliarem as aplicações e participarem de entrevistas semanais com o grupo de pesquisa. Os resultados apontaram que as redes 4G não foram usadas pelos usuários devido à inexistência de cobertura ou pelo alto consumo de bateria decorrente do acesso, mas quanto às 3G e WiFi, os usuários declararam-se satisfeitos com os serviços obtidos através

dessas tecnologias. A pesquisa reportou que fatores externos podem afetar a nota que um usuário avalia uma aplicação móvel, entre eles estão a existência de um computador pessoal onde se possa realizar a mesma atividade, a experiência do usuário com a aplicação móvel ou até mesmo o ambiente no qual ele se encontra. Outra questão apontada é a expectativa do usuário em ter um desempenho semelhante entre uma aplicação móvel e àquela executada em um computador pessoal. Dessa forma, os autores não conseguiram estabelecer uma forte relação entre QoE e QoS, pois muitos fatores subjetivos afetaram as avaliações feitas pelos usuários.

Goncalves et al. (2010) propõem um *framework* para o desenvolvimento de aplicações Android sensíveis aos níveis de QoS das redes, o RT-MOBFR. O intuito é fornecer um conjunto de classes que possibilitem a migração de tarefas entre dispositivos, que são nós de uma rede, para economizar energia e melhorar a qualidade dos serviços fornecidos. A ferramenta é composta por módulos gerenciadores responsáveis por encontrar dispositivos vizinhos, descobrir seu estado e gerenciar a instalação, execução e remoção de aplicativos. Nessa solução, cada nó da rede informa regularmente seu estado atual e os serviços instalados, de forma que caso um serviço esteja com baixa qualidade o gerenciador procura um nó que possa auxiliar na execução desse serviço. Ao encontrar um nó disponível, o gerenciador instala o serviço que será então acionado via *Intent* do Android. O trabalho, no entanto, não realizou experimentos para que fossem demonstrados os ganhos da utilização de *framework.*

Em Ye et al. (2010, 2011), os autores propõem um *framework* baseado em QoS e gerenciamento de energia, utilizando serviços nas nuvens. O objetivo é coletar informações através dos perfis do serviço e os dados de estado do dispositivo para decidir entre uma execução local ou na nuvem. A ideia do *framework* é utilizar a nuvem sempre que os requisitos de qualidade de serviço sejam atendidos, caso contrário a execução é realizada no dispositivo. A decisão é realizada pelos módulos do *framework* que analisam os perfis dos serviços e calculam o custo da comunicação e transferência da execução para nuvem. Através de estudo de caso foi possível economizar até 66% o consumo de energia de um Netbook com Intel Atom N270 1.60GHz e 1GB de memória RAM.

Em Ma et al. (2006), foi apresentada uma metodologia para o desenvolvimento de aplicações embarcadas baseadas em componentes que utilizam contratos de QoS para selecionar os componentes. Os autores afirmam que a sensibilidade a níveis de QoS é importante, principalmente, para aplicações com restrições de tempo de execução e ocupação de memória. Eles assumem a existência de um grande repositório de componentes e que tais componentes são parametrizáveis. Assim, a quantidade de combinações entre valores possíveis para os parâmetros e os componentes seria extremamente grande, inviabilizando a escolha de componentes por força bruta. Nesse cenário, os estudiosos utilizaram uma abordagem baseada em otimização de Pareto e algoritmo evolutivo em um estudo de caso, onde cada componente obrigatoriamente deveria atender os requisitos não funcionais que determinavam a qualidade de serviço. A abordagem proposta apresentou um conjunto de componentes candidatos, algumas ordens de grandeza, menor do que se esse conjunto fosse escolhido aleatoriamente.

De acordo com Gatti et al. (2011), requisitos não funcionais podem expressar a qualidade que se espera de um sistema e, portanto, é importante acompanhar tais requisitos em todo o desenvolvimento de uma aplicação. Com base nessa motivação, desenvolveu-se uma extensão para uma ferramenta de projeto, DiaSuite, com o intuito de gerenciar e fornecer rastreabilidade de um requisito não funcional em todo o processo de desenvolvimento de *software* para aviões. A extensão desenvolvida por Gatti permite monitorar a qualidade de

serviço, aqui relacionada ao tempo de execução, desde a análise de requisitos até a entrega do sistema embarcado. No entanto, não foram apresentados experimentos que mostrassem o ganho real em utilizar a ferramenta com a extensão comparado ao uso da ferramenta sem a extensão.

A qualidade de serviço mesmo em sistemas móveis está ainda fortemente ligada ao desempenho da rede que será utilizada pela aplicação. Ainda que alguns trabalhos já apresentem o QoS mais voltados a aplicação, é frequente associar essa qualidade ao custo de uma transmissão na rede ou migração de serviço entre nodos. Dessa forma, os resultados das pesquisas nessa área ainda não fornecem muitas informações ao desenvolvedor para auxiliá-lo de forma a obter melhor qualidade de suas aplicações e, consequentemente, garantir uma experiência diferenciada para o usuário final da aplicação.

2.2 Ferramentas para avaliação de desempenho em sistemas móveis

A difusão dos dispositivos móveis permitiu que a população avaliasse o desempenho das aplicações móveis, muitas vezes, comparando o desempenho de rendimento de um *desktop* a um *smartphone*. Assim, os desenvolvedores necessitam de suporte para avaliar o desempenho das suas aplicações.

A plataforma Android[5] disponibiliza algumas ferramentas de teste para aplicação, traçar o caminho de execução e analisar seu desempenho. Além dessas funcionalidades disponibilizadas pelo pacote padrão do Android, é possível encontrar algumas outras ferramentas que auxiliam na avaliação do desempenho de uma aplicação em desenvolvimento. Algumas dessas ferramentas foram utilizadas como auxílio em uma primeira abordagem dessa dissertação e serão apresentadas brevemente nas seções a seguir.

2.2.1 Emulador Android

O SDK[6] do Android possui um emulador de dispositivo móvel que dispõe de funcionalidades que possibilitam ao desenvolvedor visualizar sua aplicação em execução quando um dispositivo real não está acessível.

O Emulador não fornece nenhum dado sobre a aplicação em execução, mas pode ser utilizado com outras ferramentas para coletar alguns dados, tendo por principal característica a versatilidade. Como não está fixa em nenhuma plataforma real de dispositivo, é possível fazer algumas combinações de componentes de *hardware*. Por exemplo, é possível escolher três arquiteturas de processador (ARM, Intel x86 e MIPS), tamanhos variados para a memória principal, para o *heap* da máquina virtual, para armazenamento principal, armazenamento externo, além de utilizar qualquer versão do sistema operacional disponível.

Essa ferramenta é de grande utilidade para visualização e simulação puramente funcional de uma aplicação. No entanto, em relação ao desempenho da aplicação, não há muita precisão. Quando combinada com outras ferramentas para coletar dados, não é possível obter uma estimativa ajustada da utilização dos recursos de um dispositivo, além de não ser exata a emulação do desempenho dos processadores do emulador.

---

[5] http://developer.android.com/about/index.html

[6] http://developer.android.com/sdk/index.html

Em alguns experimentos iniciais com a emulação de diferentes arquiteturas de processamento, a imagem de processador Intel x86 ofereceu um desempenho superior (algumas ordens de grandeza) ao processador ARM. Entretanto, essa diferença não é justa por ter sido obtida devido ao custo de se integrar o processador do emulador ao computador hospedeiro. Como o hospedeiro utiliza um processador Intel, o emulador com essa imagem tem um desempenho superior, o que não necessariamente é o cenário obtido se a comparação fosse realizada entre dispositivos reais.

O emulador teria uma vantagem na avaliação de desempenho de aplicações, pois seria possível estruturar diversas especificações de *hardware* e, assim, avaliar quais aspectos mais influenciam o desempenho de uma aplicação. Porém, devido à sua imprecisão, não é possível confirmar se o desempenho verificado no emulador seria repetido em um dispositivo real de mesma configuração. Outro fator em desfavor do emulador é a ausência de uma emulação do consumo de potência necessário para executar as aplicações. Focando principalmente em obter, ainda que indiretamente, dados sobre o consumo de potência do dispositivo, foi utilizado um emulador baseado na versão padrão, mas com algumas alterações, o AndroProf, apresentado a seguir.

2.2.2 AndroProf

O AndroProf desenvolvido por Luiz Sartor et al. (2013) é baseado no emulador padrão do Android que, por sua vez, utiliza o QEMU, definido por Bellard (2005), ferramenta de código aberto para virtualização e emulação de *hardware*. O AndroProf tem como intuito incrementar o emulador fornecido pelo Android SDK para que seja possível coletar informações sobre os *basic blocks* de uma aplicação. Ao coletar essas informações, o AndroProf visa possibilitar a estimativa de custo por ciclo e também custo de potência das aplicações executadas no emulador. Todas as possíveis configurações existentes no emulador padrão do Android estão disponíveis além da coleta de *basic blocks*.

O conjunto dessa ferramenta é composto, ainda, por uma classe Android que deve ser estendida pela *Activity* principal da aplicação, pois insere intervenções de código para que os dados sejam corretamente identificados após a execução da aplicação. Ainda, é disponibilizada uma aplicação Java para que seja realizada a importação dos dados gerados pelo AndroProf para classificação e análise.

Com os dados carregados na aplicação Java, o usuário da ferramenta pode então definir custo em ciclos e em potência para cada tipo de instrução de uma determinada arquitetura e, assim, obter o custo total e a potência dissipada para executar a aplicação avaliada. Nesse sentido, o AndroProf assume que uma única execução da aplicação é suficiente para a coleta das informações de interesse (ou poucas execuções).

Essa ferramenta foi utilizada em testes iniciais para coletar dados do Navegador Tint Browser[7]. Entretanto, por ser uma aplicação voltada para internet, a necessidade de repetições de um mesmo teste consecutivamente fez com que o AndroProf gerasse um arquivo de log que crescia rapidamente e atingia o valor máximo de tamanho de arquivo, ocasionando uma parada de funcionamento da aplicação.

[7] https://play.google.com/store/apps/details?=org.tint

2.2.3 Genymotion

O *Genymotion*[8] é uma opção interessante para desenvolvedores que necessitam de acesso a uma variedade maior de dispositivos para testar suas aplicações. Apesar de ser comercial, possui versões para teste assim como versões acadêmicas.

Essa ferramenta é baseada na utilização de uma máquina virtual rodando o sistema operacional Android. Para gerir essas máquinas virtuais, o *Genymotion* usa como base o VirtualBox[9], sendo necessária a instalação no ambiente de desenvolvimento. Essa virtualização possibilita que o *Genymotion* seja executado nas três principais plataformas para *desktop* (*Windows*, *Linux* e *Mac*). Ainda visando a portabilidade, o Genymotion disponibiliza *plug-ins* tanto para o ambiente de desenvolvimento Eclipse quando para o Android Studio.

O objetivo do Genymotion é fornecer imagens de dispositivos Android, ou seja, são disponibilizadas algumas imagens que buscam representar dispositivos reais Android e sua especificação de *hardware* é pré-definida. No entanto, a ferramenta permite a criação de imagens personalizadas possibilitando encontrar diversas imagens dos dispositivos mais novos e com maior visibilidade do público consumidor.

Uma das vantagens fornecidas é a economia de tempo para sua utilização. De acordo com o site da ferramenta, um dispositivo virtual *Genymotion* pode ser três vezes mais rápido do que um real quando é realizado o *deployment* de uma aplicação. Outra vantagem é a possibilidade de simular alguns sensores que não estão presentes no emulador padrão, como GPS (do inglês, *Global Positioning System*) e acelerômetro. Essa ferramenta conta também com um módulo que permite inserir variações na bateria, possibilitando assim observar a ação da aplicação em diversos níveis. Entretanto, esse módulo não se baseia em algum modelo de gasto de bateria, as variações devem ser inseridas manualmente pelo utilizador do Genymotion.

A plataforma que o Genymotion utiliza é baseada em processadores Intel x86, ou seja, todas as imagens de dispositivos terão uma CPU dessa arquitetura tornando assim o emulador mais eficiente. Em contrapartida esta configuração dificulta que o desempenho observado em uma imagem criada com o Genymotion seja semelhante àquele percebido no dispositivo real representado.

2.2.4 Android Device Monitor

O Android Device Monitor[10] é uma aplicação composta de várias funcionalidades que o próprio SDK do Android fornece ao desenvolvedor para analisar e rastrear a execução de uma aplicação Android. Essas funcionalidades podem estar disponibilizadas tanto em uma aplicação independente (o próprio *Device Monitor*) como também fazem parte do *plug-in* desenvolvido para o ambiente Eclipse. A seguir são apresentadas algumas das funcionalidades que compõem o *Device Monitor*.

[8] https://www.genymotion.com/

[9] https://www.virtualbox.org/

[10] http://developer.android.com/tools/help/monitor.html

2.2.4.1 Dalvik Debug Monitor Server

Essa ferramenta fornece ao desenvolvedor um grande volume de informações sobre um dispositivo real ou virtual que esteja executando uma aplicação Android. Através dessa ferramenta, é possível obter dados de utilização da rede, ocupação do *heap*, alocação de memória e *threads*, além de emular operações realizadas com o telefone, como chamadas de voz e operações com SMS. Todas essas informações permitem ao desenvolvedor conhecer o comportamento de sua aplicação e entender quais partes do código acarretam ao sistema alguma sobrecarga, seja na forma de um alto consumo de memória ou muita utilização da rede.

2.2.4.2 Tracer for OpenGL ES

Essa ferramenta[11], como o nome indica, foi desenvolvida para monitorar o OpenGL para dispositivos embarcados. Quando uma aplicação é voltada para a utilização de processamento e/ou geração de imagens, é utilizado o OpenGL para auxiliar na manipulação dos gráficos. O *Tracer* permite ao desenvolvedor acompanhar, quadro-a-quadro, o comportamento da aplicação e como cada comando está sendo executado.

2.2.4.3 Hierarchy viewer

O usuário de uma aplicação móvel deseja que seu dispositivo execute, com desempenho satisfatório, suas aplicações preferidas e, também, espera ter uma experiência agradável com a interface gráfica da aplicação. A interface é a comunicação entre os processamentos realizados pela aplicação e o usuário e, cada vez mais, é elaborada e agradável. Essa evolução faz com que seja necessário avaliar seu desempenho e impacto no desempenho total da aplicação.

No desenvolvimento de uma aplicação Android, a construção da interface é realizada através de arquivos .xml e pode conter uma hierarquia de *Views.* Através do *Hierarchy viewer*[12] é possível depurar cada nível dessa hierarquia e otimizá-la para que o tempo de resposta da aplicação fique menor, sem que isso acarrete perda de usabilidade para o usuário.

2.2.4.4 Traceview

De acordo com a definição apresentada no site para desenvolvedores Android[13], essa é uma ferramenta visual para logs de execução de uma aplicação.

Para utilizar essa ferramenta é necessário que algumas intervenções sejam realizadas no código da aplicação para delimitar o espaço de monitoramento que gerará o arquivo de log. Dessa forma, é necessário acrescentar a linha de código *Debug.startMethodTracing("nomeDoArquivodeTrace");* antes do ponto inicial a ser monitorado e *Debug.stopMethodTracing();* após o ponto final do código a ser monitorado.

[11] http://developer.android.com/tools/help/gltracer.html

[12] http://developer.android.com/tools/help/hierarchy-viewer.html

[13] http://developer.android.com/tools/help/traceview.html

Assim, todos os caminhos de execução compreendidos entre as duas chamadas serão acompanhados e comporá um arquivo *.trace*.

O processamento desse arquivo pode fornecer um gráfico de chamada dos métodos da aplicação, facilitando ao desenvolvedor conhecer o fluxo de uma aplicação que ele não tenha desenvolvido. Ainda com o arquivo de .trace podem ser obtidos dados de tempo de execução de cada método.

2.2.5 Littleeye

O *Littleeye*[14] é uma ferramenta para análise de desempenho de aplicações Android. Essa ferramenta pertencia a uma empresa independente que foi adquirida pelo *Facebook* em 2014.

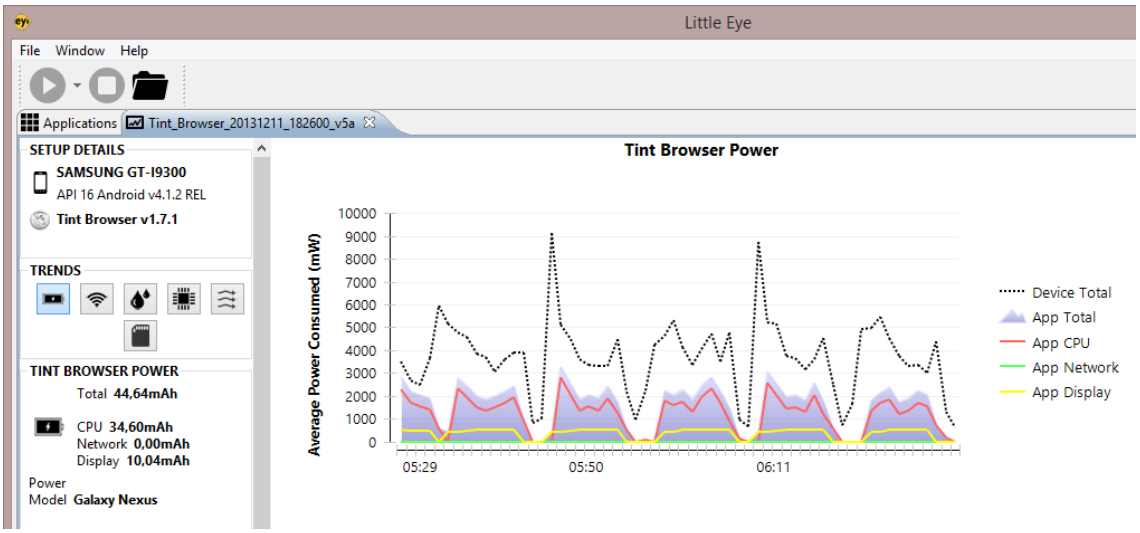
O objetivo dessa ferramenta é monitorar a execução de uma aplicação Android e coletar dados relacionados à CPU, utilização de rede, utilização do sistema de armazenamento, consumo de potência e ocupação de memória. Busca-se fornecer praticidade ao desenvolvedor que deseja avaliar uma aplicação, pois basta conectar o dispositivo a um computador com o *Littleeye* e selecionar na tela inicial dessa ferramenta qual aplicação do dispositivo deseja monitorar.

O *Littleeye* oferece, ao fim do monitoramento, um relatório com todos os dados coletados da aplicação selecionada e, assim, o desenvolvedor pode ter uma visão geral de sua aplicação. É fornecido ainda, em tempo real, gráficos das métricas monitoradas e *screenshots* do dispositivo. Dessa forma, o utilizador da ferramenta pode, manualmente, observar um determinado estado do dispositivo e o consumo dos recursos nesse momento.

Para quase todas as métricas coletadas pelo *Littleeye,* os dados são obtidos do dispositivo real. No entanto, para a métrica de consumo de potência, o *Littleeye* possui alguns modelos preditivos de consumo e os utiliza para gerar o gráfico de consumo de potência. A variedade de modelos não é muito grande, o que leva à utilização de modelos diferentes dos dispositivos reais utilizados. Assim, essa projeção do consumo de potência perde precisão devido à diferença nos modelos. A Figura 2.1 mostra um trecho do monitoramento da aplicação *Tint Browser* em um dispositivo Galaxy S3 e o modelo preditivo utilizado é do Galaxy Nexus. Esses modelos, apesar de serem de um mesmo fabricante, possuem especificações de *hardware* bastante distintas.

Figura 2.1 – Execução do *Littleeye* com gráfico de potência consumida.

[14] http://www.littleeye.co/



Fonte: Próprio autor.

O *Littleeye* é uma ferramenta prática para o desenvolvedor que deseja monitorar uma determinada aplicação sem ter qualquer contato com o código fonte da mesma. Todavia, a métrica de consumo de potência pode não ser precisa por utilizar, para diferentes dispositivos, o mesmo modelo preditivo para gerar os dados.

2.2.6 NeoLoad

A empresa NEOTYS[15] fornece algumas ferramentas para coleta de métricas em dispositivos móveis. Mesmo sendo comerciais, possuem versão de testes com limitação das coletas de dados. Entre essas ferramentas está o *NeoLoad*[16].

O *NeoLoad* possibilita coletar as métricas de uma determinada aplicação quando executada em dispositivos reais previamente selecionados. Uma das justificativas para a criação dessa ferramenta é que as aplicações móveis não são executadas em um ambiente 100% controlado e sem concorrência, mas em um com muitos outros dispositivos e cada um deles têm algumas aplicações em execução. Isso torna necessário então que as aplicações sejam testadas nos ambientes em que realmente serão executadas pelo usuário final.

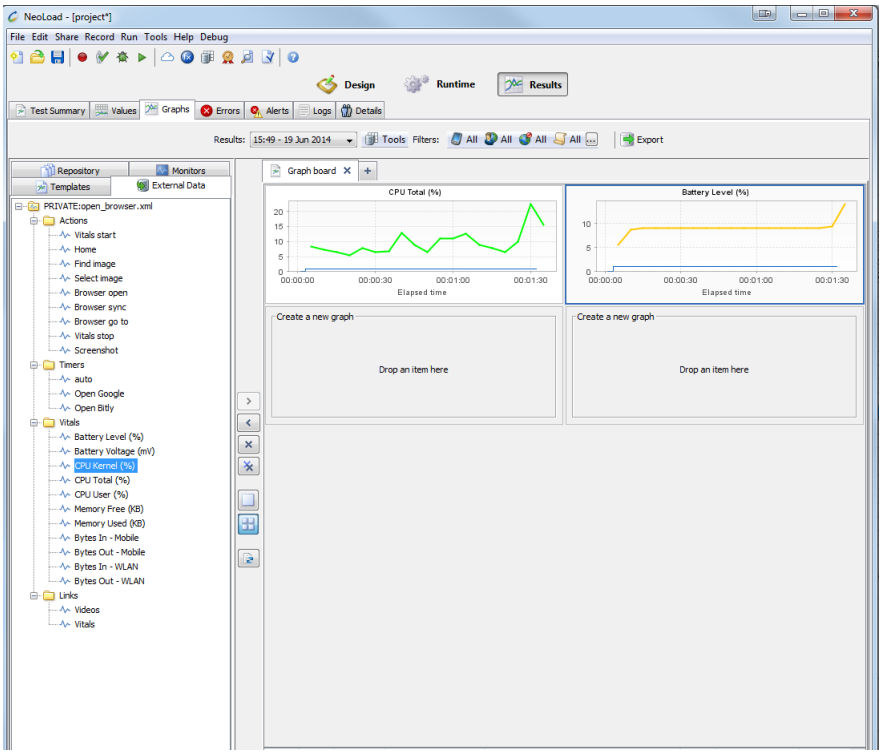
O *NeoLoad* permite que o interessado crie cenários de teste, selecione dispositivos reais onde tais cenários serão executados e, então, podem ser criados outros dispositivos virtuais que estarão no ambiente de teste, assim como podem determinar diferentes condições de rede durante a execução do cenário. Após o início da execução, o usuário pode monitorar as transações de rede entre a aplicação e um servidor, como também monitorar o consumo de CPU, memória ou bateria do dispositivo.

A Figura 2.2 mostra a coleta de dados de CPU e bateria do dispositivo real para um cenário de teste.

Figura 2.2 – Aparência da ferramenta *NeoLoad*

[15] http://www.neotys.com/introduction/real-device-testing.html

[16] http://www.neotys.com/product/overview-neoload.html



Fonte: Site da NEOTYS

Como apresentado nesse capítulo, existem algumas ferramentas para avaliação do desempenho de uma aplicação Android. No entanto, é preciso que o desenvolvedor codifique sua aplicação e, utilizando algumas (provavelmente mais de uma) dessas ferramentas, avalie o seu desempenho. Essa abordagem faz com que o desenvolvedor tenha o retrabalho de modificar sua aplicação e reavaliá-la caso o desempenho não seja satisfatório. Esse processo, além de demorado, pode ter um custo alto, pois implica em mudanças no projeto da aplicação depois que essa foi implementada. Assim, seria útil ao desenvolvedor estimar esse desempenho antes de realmente codificar toda sua aplicação. As Tabelas 2.1 e 2.2 apresentam um resumo das características das ferramentas apresentadas nessa seção.

Tabela 2.1 – Características dos emuladores Android

	CPU	*Memória*	*Energia*	*Heap*	*ADB*	*Sensores*
Emulador	Intel,	Tamanho	Não	Tamanho	Sim	Não

SDK	Arm e Mips	ajustável		ajustável		
AndroProof	Intel, Arm e Mips	Tamanho ajustável	Estimativa	Tamanho ajustável	Sim	Não
Genymotion	Intel	Tamanho definido por *template*	Variação manual	Tamanho definido por *template*	Sim	GPS, Acelerômetro

Fonte: Próprio autor.

Tabela 2.2 – Características dos monitores de recurso Android

	CPU	*Memória*	*Energia*	*Heap*	*Interface Gráfica*	*Cenários de execução*
Android Device Monitor	Sim	Sim	Não	Sim	Sim	Não
Littleeye	Sim	Sim	Estimada	Sim	Não	Não
Neoload	Sim	Sim	Sim	Não	Não	Sim

Fonte: Próprio autor.

3 ABORDAGEM INICIAL

Diferente das atuais metodologias de avaliação para plataforma móvel que focam prioritariamente o dispositivo, o objetivo desta dissertação é avaliar como decisões de projeto do desenvolvedor afetam o desempenho de uma dada aplicação e como as diferenças entre dispositivos móveis impactam o desenvolvimento dessa. Pelo exposto no Capítulo 2 deste trabalho, não é possível identificar, nas pesquisas relacionadas, uma metodologia capaz de fazer este tipo de avaliação. Não foi possível nem mesmo identificar uma metodologia para análise de desempenho em sistemas móveis, uma vez que os trabalhos apresentam, em sua maioria, estudos de caso limitados e com diferentes objetivos. Em virtude dessa ausência de padronização, a primeira abordagem desta dissertação consiste em aplicar uma metodologia estruturada de avaliação de desempenho em plataformas móveis e verificar sua viabilidade como auxilio ao projetista na tomada de decisões.

De acordo com a metodologia proposta por Jain (1991), o processo de avaliação de desempenho não possui uma única maneira de ser realizado . No entanto, uma estrutura básica deve ser seguida visando possibilitar a reprodução dos experimentos, a confiabilidade dos resultados e, também, a otimização no tempo de realização do processo. Resumidamente, essa estrutura básica é composta i) pela definição do objetivo; ii) por um detalhamento da metodologia utilizada no processo; iii) pela execução dos experimentos; e iv) pela exibição dos resultados de maneira apropriada.

Neste trabalho, essa estrutura básica foi utilizada e adaptada para a avaliação de desempenho de uma aplicação Android, Tint Browser, por ser amplamente utilizada em dispositivos móveis para navegação de internet. A avaliação foi feita através do Emulador disponível na SDK do Android e considerando algumas possíveis configurações de dispositivos. O contexto geral dessa primeira abordagem é avaliar o desempenho de um navegador Android perante um cenário de oito possíveis configurações do emulador, variando o tamanho da memória principal, o tamanho do *heap* e a versão do sistema operacional. Através de uma carga de trabalho representativa, isto é, dos dados utilizados para exercitar a aplicação e de métricas, obter o desempenho do navegador ao exibir páginas e ao final apresentar os dados das métricas coletadas.

Dentro do contexto apresentado, o principal objetivo desta abordagem inicial é descrever a aplicação de uma metodologia padrão, para avaliação de desempenho, em um sistema móvel baseado na plataforma Android, listando vantagens e limitações. O objetivo secundário deste experimento é descrever como uma avaliação de desempenho estruturada pode ajudar o desenvolvedor na implementação de uma aplicação móvel.

A seção 3.1 apresenta uma visão geral da plataforma Android, além de dados de mercado que mostram o domínio dessa plataforma no âmbito de dispositivos móveis motivando sua utilização nesses experimentos.

3.1 Android

A plataforma Android é atualmente o sistema móvel com maior base instalada em dispositivos em todo mundo, de acordo com o *site* da própria plataforma *Android Developers*. Conforme a Tabela 3.1 o *Android* obteve quase 85% do mercado de *smartphones* no mundo no segundo quadrimestre do ano de 2014.

Tabela 3.1 – Tabela de mercado de *smartphones*.

Período	*Android*	*iOS*	*Windows Phone*	*BlackBerry OS*	*Outros*
Q3 2014	84.4%	11.7%	2.9%	0.5%	0.6%
Q3 2013	81.2%	12.8%	3.6%	1.7%	0.6%
Q3 2012	74.9%	14.4%	2.0%	4.1%	4.5%
Q3 2011	57.4%	13.8%	1.2%	9.6%	18.0%

Fonte: IDC[17], 2014 Q3

O Android é um sistema operacional móvel que disponibiliza, para os desenvolvedores, um numeroso conjunto de ferramentas para desenvolvimento de aplicações. Atualmente, a loja de aplicativos tem uma média de 1,5 bilhões de *downloads* por mês. Toda essa difusão faz do Android uma plataforma-alvo para os desenvolvedores de aplicações móveis e, por isso, foi escolhida para a implementação desse *framework.*

As aplicações Android podem ser desenvolvidas na linguagem Java e incrementada com muitas bibliotecas fornecidas dentro do SDK do Android, sendo que existem dois conceitos, *Activity* e *Intent*, trazidos do SDK do Android para a linguagem JAVA.

De acordo com a definição disponível no site da plataforma, uma *Activity* é uma atividade simples e focada nas ações do usuário. Comumente, uma *Activity* é criada para interagir com o usuário final e, normalmente, está associada a uma tela que é mostrada por uma aplicação que pode oferecer campos a serem preenchidos pelo usuário com os dados necessários e, posteriormente, processá-los.

De acordo com o site do Android, uma *Intent* é estrutura de dados passiva que contém uma descrição abstrata de uma operação a ser realizada. Uma *Intent* pode ser utilizada para acionar código entre diferentes aplicações e, também, lançar *Activities* transmitindo para essa *Activity* parâmetros que serão utilizados durante a execução.

3.2 Avaliação de desempenho para aplicações móveis

Nessa avaliação foi realizada uma adaptação da proposta de Jain (1991) para aplicações de dispositivos móveis Android. A seguir, são detalhados os passos da estrutura básica dessa proposta.

3.2.1 Objetivo

O objetivo desse estudo é avaliar o desempenho de uma aplicação móvel, navegador de páginas Android.

[17] http://www.idc.com/prodserv/smartphone-os-market-share.jsp

3.2.2 Metodologia

- Definir o sistema sob avaliação e o componente a ser estudado:

  - exibição de páginas *web* no dispositivo e o componente estudado é a implementação Android de um navegador de internet, o Tint Browser.

- Definir os serviços avaliados:

  - apresentação da página *web* no aplicativo.

- Definir métricas:

  - Tempo de resposta: tempo total para exibir a página;

  - Tempo de CPU: tempo que o processo de exibição de página utiliza o CPU;

  - Uso de memória: volume de memória ocupada para a exibição de uma página.

Apesar de o emulador possuir uma representação da bateria, tal representação não inclui uma simulação de gasto de bateria. Por essa razão, a métrica de consumo de bateria não pode ser considerada neste momento.

- Definir parâmetros:

  - CPU Intel x86 no emulador;

  - Única versão da aplicação Tint Browser;

  - Servidor de páginas local.

- Definir fatores:

  - Tamanho de memória: 384MB e 1024MB;

  - Tamanho do *heap*: 32MB e 64MB;

  - Versão do Android: 4.0.3 e 4.2.2.

- Definir carga de trabalho:

O espaço de busca para avaliar a exibição de páginas *web* é extenso, pois há milhões de páginas acessadas e outras novas são criadas diariamente. Mesmo se avaliarmos as mais acessadas, ainda será um número muito grande. Visando tornar essa carga menor, mas sem perder a representatividade, foram usados métodos para agrupar páginas de acordo com sua semelhança em termos de necessidade de recursos. Assim, com base em um *dataset*[18] das páginas mais acessadas no ano de 2011[19], utilizamos a processo de *clustering* para determinar grupos de páginas com requisitos de ocupação memória e tempo de CPU similares. O processo de *clustering* é uma técnica de *data mining*, Berry; Linoff (1997), para agrupar objetos com base em sua semelhança, nesse experimento foi utilizada a ferramenta R statistics[20]. O gráfico gerado ao final do processo é mostrado na Figura 3.1, de forma a evidenciar quatro grupos distinguidos por cores e formas dos pontos. Cada grupo possui um

[18] http://www.csg.uzh.ch/publications/data/mobilewebstandards.html

[19] http://www.alexa.com/topsites

[20] http://www.r-project.org

ponto utilizado como centro para definir o grupo, representados pelos centros das elipses, e qualquer dos pontos pode ser usado para representar todo o grupo.

Dessa forma, a carga de trabalho final para o experimento consiste em acessar quatro páginas *web*, cada uma representando um conjunto de complexidade semelhante:

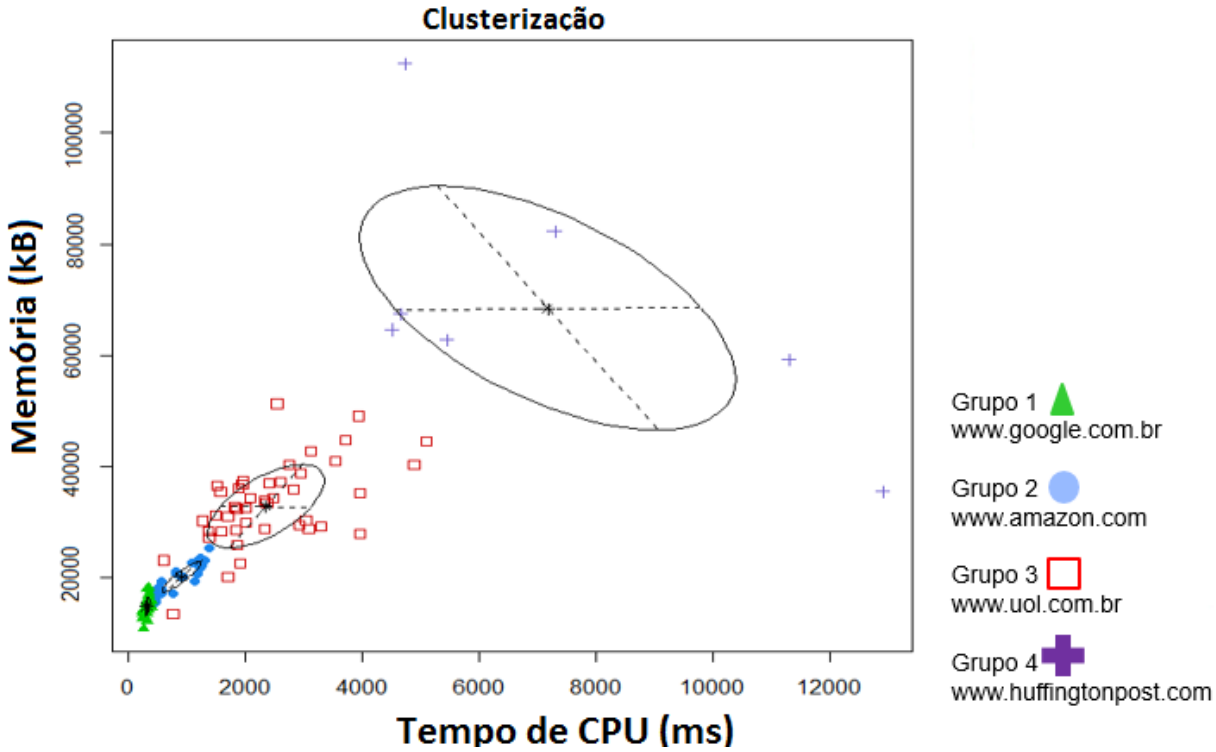
Grupo 1: www.google.com.br

Grupo 2: www.amazon.com

Grupo 3: www.uol.com.br

Grupo 4: www.huffingtonpost.com

Figura 3.1 – Processo de *clustering* de páginas web.



Fonte: Próprio autor.

Após definir a carga de trabalho, foi analisado, para cada grupo de páginas, quais fatores têm maior influência nas métricas avaliadas. Esta análise tem por objetivo reduzir o número total de experimentos realizados. Nesta pesquisa, optou-se pela técnica de projeto de experimentação chamada 2^k fatorial, Jain (1991), que seleciona dois possíveis valores para cada um dos fatores e, assim evidencia quais desses k fatores têm maior impacto nas métricas selecionadas. Aplicando esta técnica às oito possíveis configurações de emuladores, foi possível eliminar os experimentos que forneceriam resultados repetidos nas métricas. Através desta análise, identificaram-se apenas quatro configurações do emulador necessárias para a avaliação, pois os fatores versão do sistema Android, o tamanho do *heap* e a combinação desses dois fatores se mostraram como principais influências nos valores das métricas. Dessa forma, a avaliação de desempenho foi feita para as configurações de dispositivos mostradas

na Tabela 3.2. Também, foi possível identificar quais fatores são mais importantes para cada métrica:

- Tempo CPU:
  - Versão do sistema operacional, para os grupos 1, 2 e 3;
  - Tamanho do *heap*, para o grupo 4.
- Uso de memória:
  - Versão do sistema operacional, para todos os grupos;
  - Interação entre sistema operacional e tamanho do *heap* para o grupo 4.
- Tempo total:
  - Versão do sistema operacional para todos os grupos.

Tabela 3.2 – Configurações de emuladores

	Versão do Android	Tamanho do *heap*
Emulador 1	Android 4.0.3	64 MB
Emulador 2	Android 4.2.2	64 MB
Emulador 3	Android 4.0.3	32 MB
Emulador 4	Android 4.2.2	32 MB

Fonte: Próprio autor.

3.2.3 Experimentos

Os experimentos são automatizados através de um *script* que envia comandos, do tipo ADB[21] (do inglês, *Android Debug Bridge*), para acionar a aplicação *Android* e fornece o URL (do inglês, *Uniform Resource Locator*) da página a ser acessada. Após o carregamento da página, a aplicação e o emulador são fechados e todos os dados de usuários excluídos. Dessa forma, objetiva-se reduzir a influência da *cache* do navegador na exibição de páginas. O *script* possui uma estrutura que garante uma exibição de 30 vezes para cada uma das páginas, para atenuar as variações de exibição da mesma. Um índice de dispersão é usado para mostrar essa variação, o intervalo semi-interquartil, Jain (1991).

A seguir são apresentados gráficos de cada uma das métricas para os quatro grupos de páginas *web*.

3.2.4 Resultados

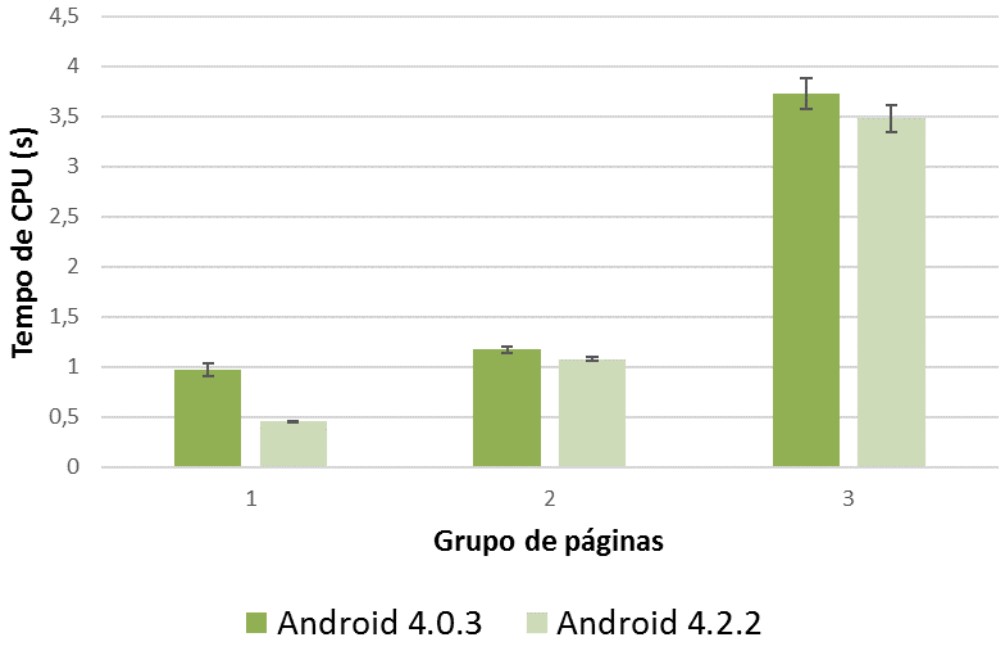
A Figura 3.2 apresenta um dos gráficos da métrica de tempo de CPU no qual as barras apresentam a média de 30 execuções e a barra de erro apresenta o índice de dispersão. Para essa métrica, o principal fator de influência é a versão do sistema operacional nos três

[21] http://developer.android.com/tools/help/adb.html

primeiros grupos de páginas, pois observamos que a versão mais nova do *Android* necessita de menos tempo de CPU, mas essa diferença de tempo é reduzida à medida que a complexidade das páginas aumenta.

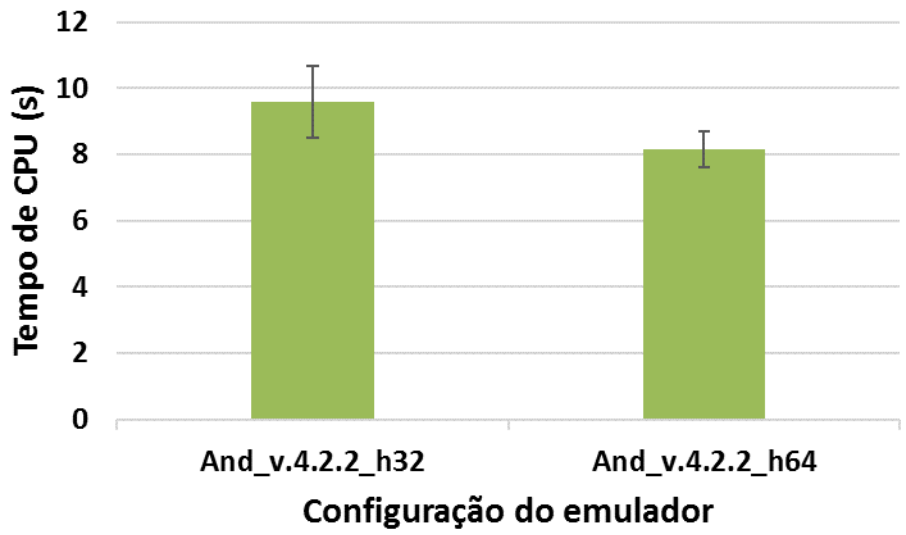
Para o grupo de páginas mais complexo, como pode ser visto na Figura 3.3, o fator de maior influência é o tamanho do *heap* da máquina virtual do Android. É possível observar, assim, que o emulador com *heap* maior necessita de um menor tempo de CPU para exibir a página.

Figura 3.2 - Gráfico da métrica de tempo de CPU para os três primeiros grupos de páginas



Fonte: Próprio autor.

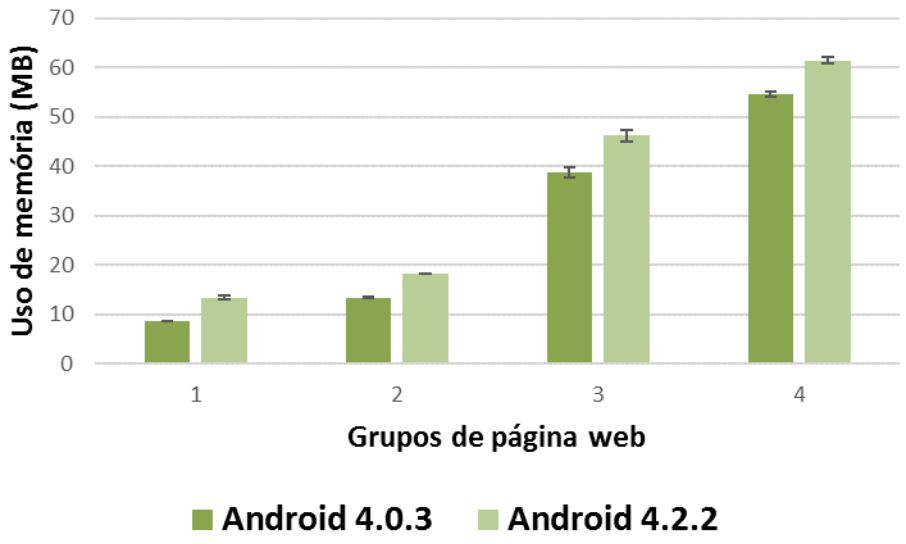
Figura 3.3 – Gráfico da métrica de tempo de CPU para o grupo 4.



Fonte: Próprio autor.

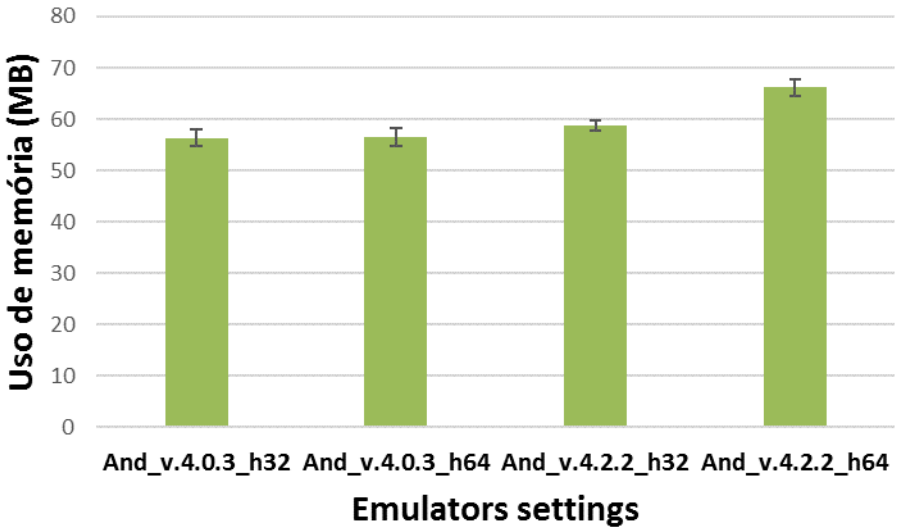
Para a métrica de ocupação de memória, todos os grupos de páginas foram influenciados pela versão do sistema operacional e o grupo 4 foi influenciado, também, pela interação da versão do sistema operacional e o tamanho do *heap*. Analisando os quatro grupos influenciados pelo sistema operacional (Figura 3.4), pode-se observar que a versão mais nova ocupa uma quantidade maior de memória do que a versão mais antiga. Quando se avalia apenas o grupo quatro, vê-se que o aumento do tamanho do *heap* não afeta a métrica na versão mais antiga do *Android*, enquanto que, na versão mais recente, o aumento do *heap* resulta em maior ocupação de memória, como mostrado na Figura 3.5.

Figura 3.4 – Gráfico de uso de memória em relação a versão do sistema operacional



Fonte: Próprio autor.

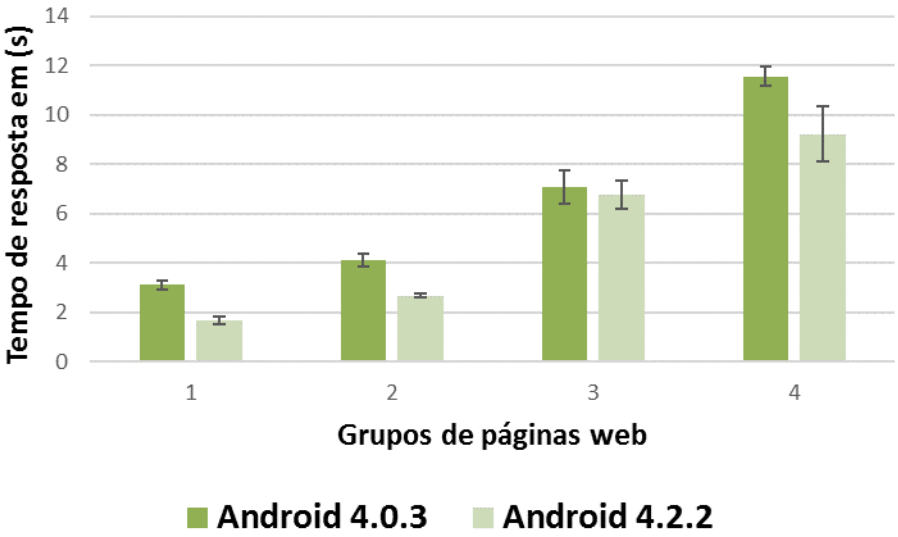
Figura 3.5 – Gráfico de uso de memória em relação a interação entre sistema operacional e *heap*



Fonte: Próprio autor.

A métrica de tempo total para exibição de uma página *web*, é, basicamente, influenciada pela versão do sistema operacional, como mostrado na Figura 3.6. Para os dois primeiros grupos de páginas, observa-se uma redução de tempo real na versão mais nova do *Android*. No entanto, nos grupos de páginas mais complexas, apesar da diferença entre o tempo total de uma versão do sistema operacional, a variação nos experimentos ofusca a redução na versão mais nova do *Android*.

Figura 3.6 – Gráfico do tempo total de execução



Fonte: Próprio autor.

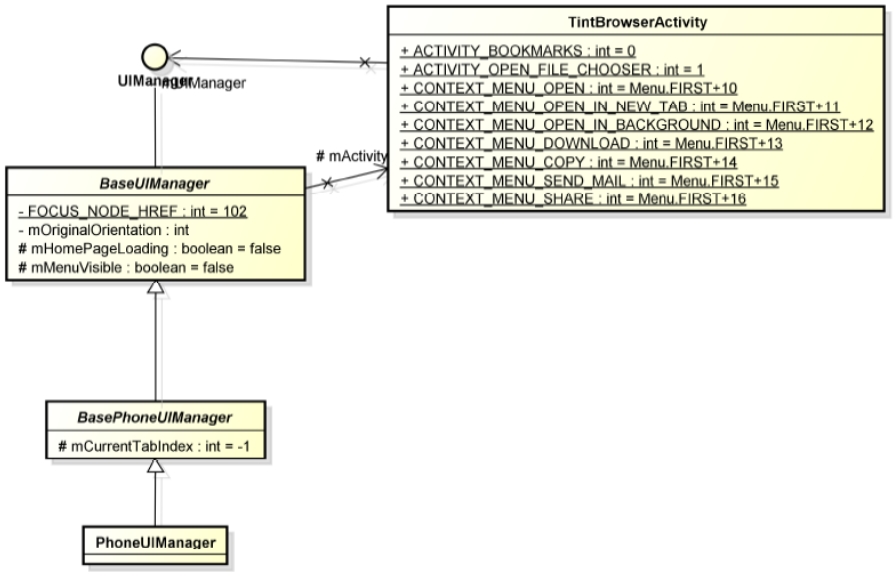
Com base nesse experimento, pode-se concluir que a aplicação da metodologia proposta por Jain (1991) para avaliar o desempenho de uma aplicação móvel é factível e ressalta informações interessantes para o projetista. Do ponto de vista da avaliação de desempenho, pudemos avaliar que nas métricas selecionadas a versão do sistema operacional teve significativa influência no comportamento da aplicação no consumo de recursos.

Com base nessa primeira avaliação de desempenho, decidiu-se avaliar se refatorações em uma aplicação móvel podem afetar o seu desempenho. Este segundo experimento está descrito a seguir.

3.3 Avaliação de impacto das refatorações

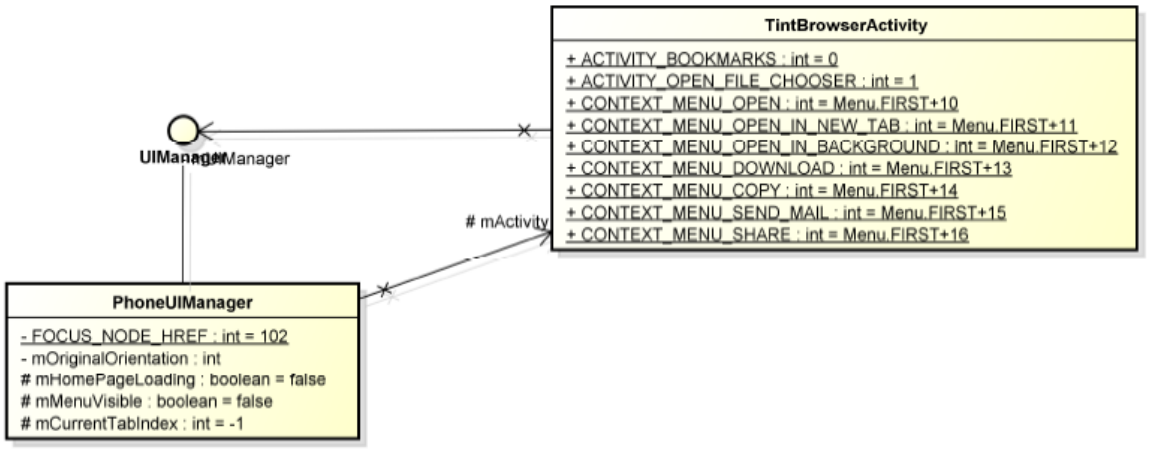
A aplicação Tint Browser, utilizada como estudo de caso, consiste em mais de 100 classes que implementam as diferentes funcionalidades do navegador, incluindo exibição de páginas, gerenciamento de *download* de arquivos e operações de gerenciamento do próprio navegador (favoritos, por exemplo). Devido ao tamanho da aplicação, decidiu-se refatorar uma parte do código relativa à principal funcionalidade do aplicativo, ou seja, a exibição de páginas. Após avaliar o código dessa aplicação, foram identificadas as classes envolvidas na execução da operação de exibição de página *web* e essas foram modificadas para gerar versões diferentes da aplicação. Um diagrama de classes do código utilizado como caso base é apresentado na Figura 3.7 (contendo apenas as classes envolvidas na funcionalidade escolhida). As refatorações executadas no código base tiveram duas inclinações: na primeira instância, o intuito é tornar o código da aplicação menos coeso, juntando todos os métodos em uma única classe. O diagrama de classes dessa refatoração é mostrado na Figura 3.8. Em seguida, buscou-se a vertente contrária, tornando a aplicação mais coesa com a distribuição das tarefas por classes, com finalidades melhor definidas. O diagrama de classes da segunda refatoração é apresentado na Figura 3.9.

Figura 3.7 – Diagrama de classe da versão base.



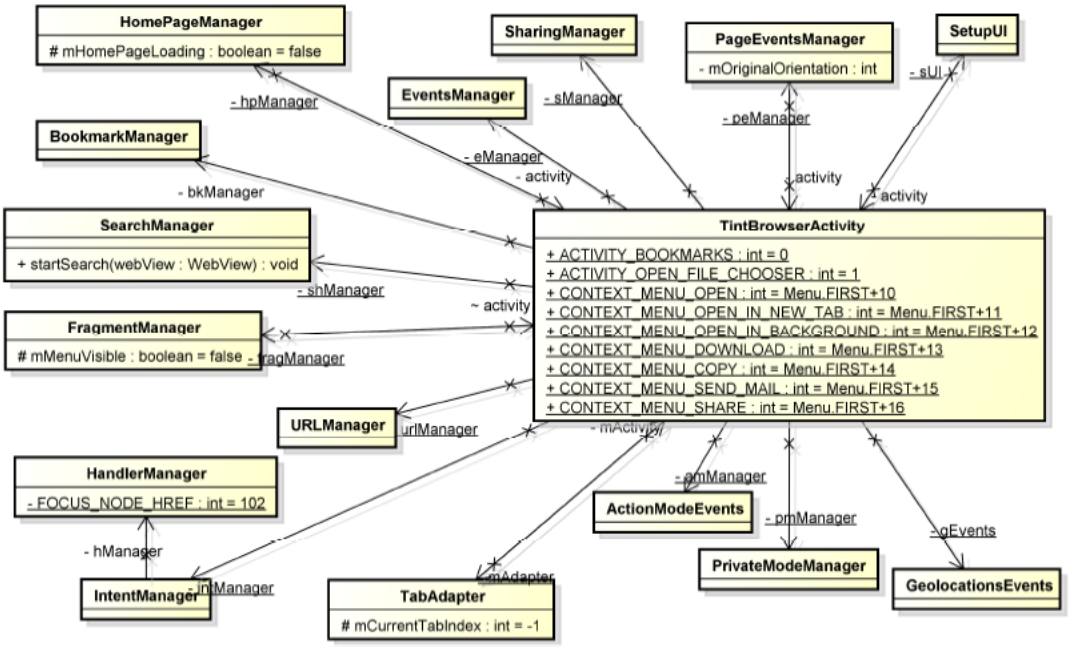
Fonte: Próprio autor.

Figura 3.8 – Diagrama de classe da versão menos coesa.



Fonte: Próprio autor.

Figura 3.9 – Diagrama de classe da versão mais coesa.



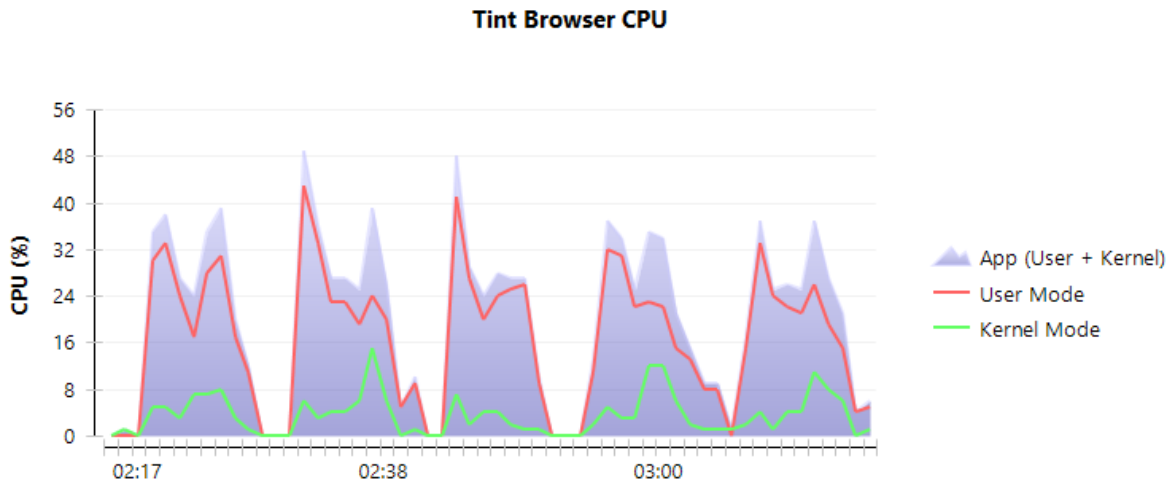
Fonte: Próprio autor.

Para a avaliação de desempenho, foi utilizado um dispositivo real, o Galaxy S3, no intuito de coletar a métrica de consumo de potência não disponível através de emuladores. O detalhamento da especificação desse dispositivo pode ser visto na

Tabela 1.2. Para essa coleta, foi utilizada a ferramenta *Littleeye*, que permite a observação tanto de ocupação da CPU e memória por uma dada aplicação como também uma estimativa do consumo de potência baseada em modelos preditivos. A carga de trabalho utilizada foi obtida de forma semelhante a do experimento anterior, utilizando, *clustering*, e resultando no acesso a quatro páginas web que representam diferentes complexidades. Foram realizadas 30 execuções do aplicativo através de um *shell script*, que envia as quatro páginas que devem ser

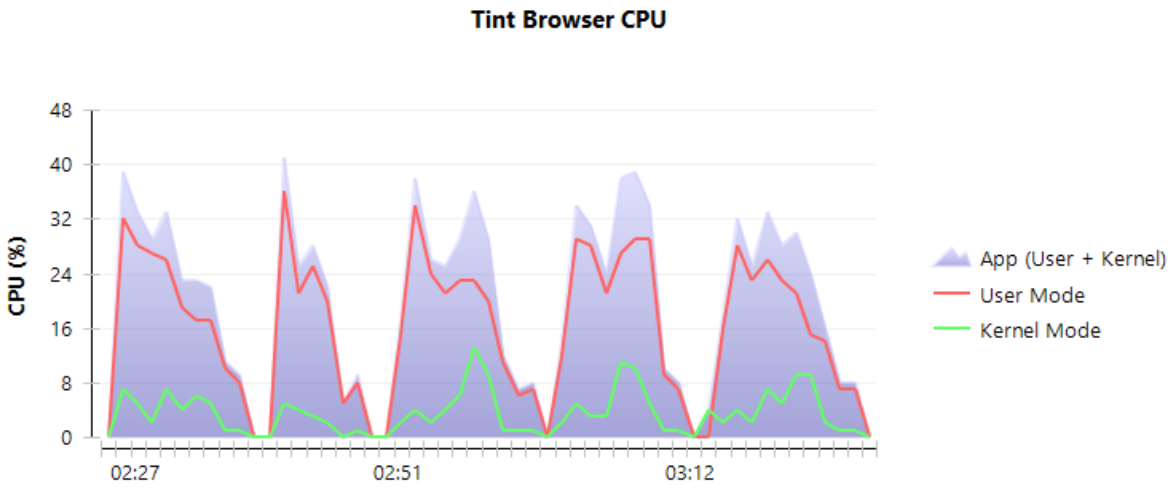
exibidas. Esse *script* é executado para cada uma das versões do Tint (base, refatoração 1 e refatoração 2, respectivamente). A métrica de ocupação de CPU é apresentada nas Figuras 14 a 16 para, respectivamente, a versão base, a versão menos coesa (refatoração 1) e para a versão mais coesa (refatoração 2).

Figura 3.10 – Métrica de ocupação de CPU da versão base



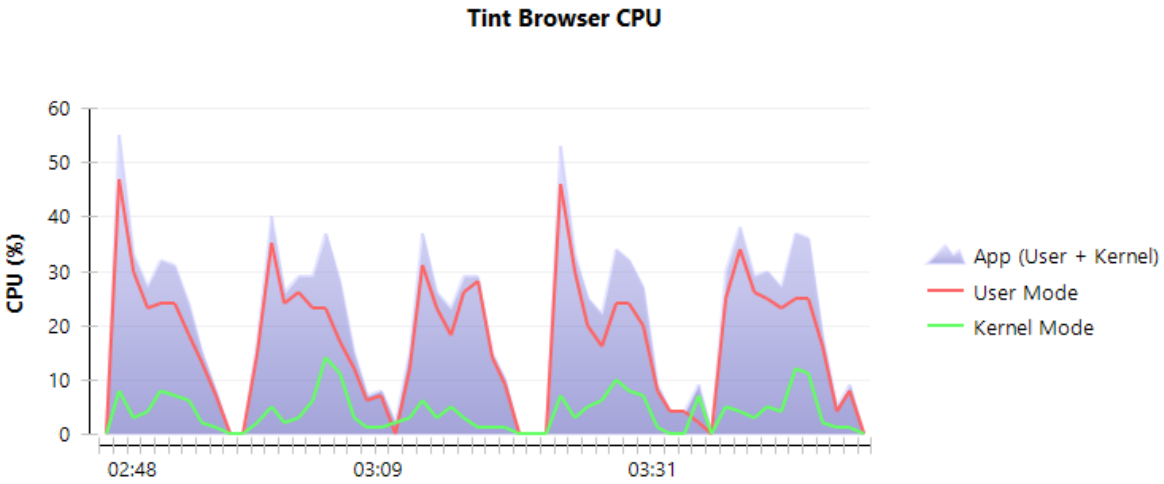
Fonte: Próprio autor

Figura 3.11 – Métrica de ocupação de CPU versão menos coesa



Fonte: Próprio autor

Figura 3.12 – Métrica de ocupação de CPU versão mais coesa

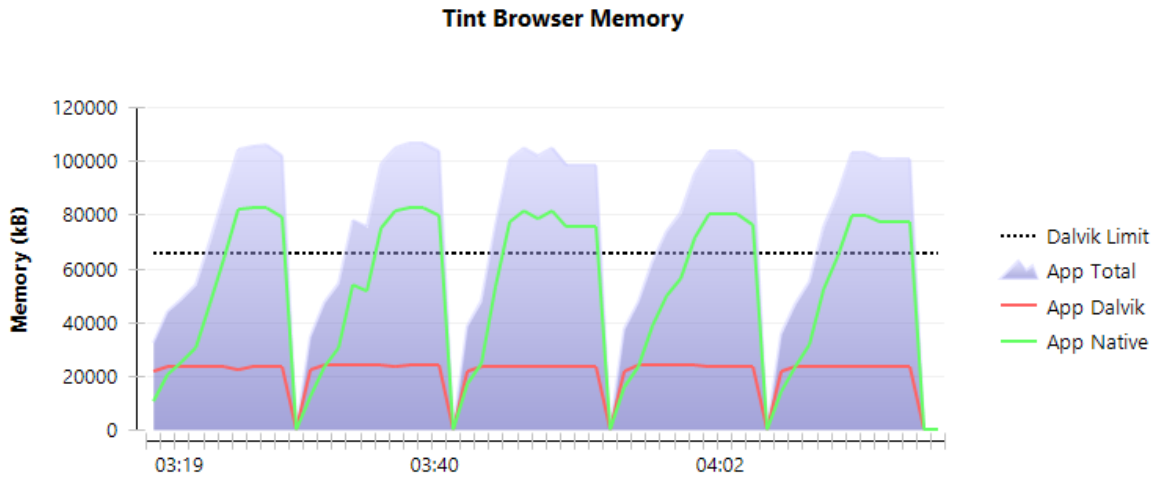


Fonte: Próprio autor

Como pode ser observado acima, existem variações entre as execuções de um mesmo experimento, ou seja, o desempenho obtido em cada uma das 30 execuções de uma versão do código é diferente. Isso pode ser observado nas diferentes áreas desenhadas nos gráficos, onde cada formação delimitada entre as ocupações de 0% representa uma execução da exibição das páginas. Por outro lado, quando comparamos o comportamento geral das três versões de código, não pode ser identificado, visualmente, um comportamento distinto entre elas, ou seja, alterar as classes não modificou o quanto a aplicação ocupa da CPU. Os gráficos fornecidos pela ferramenta têm escalas diferentes e não possibilitaram a determinação mais exata dos valores fornecidos.

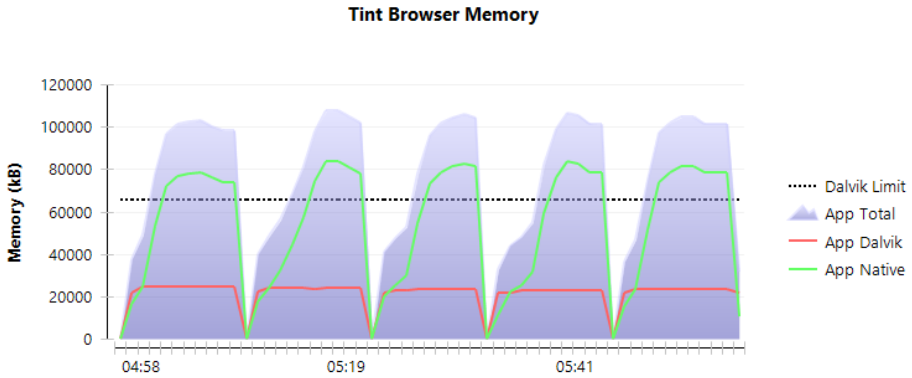
As Figuras 17 a 19 mostram a métrica de ocupação de memória das três versões de código e a mesma tendência é observada.

Figura 3.13– Métrica de ocupação de memória da versão base



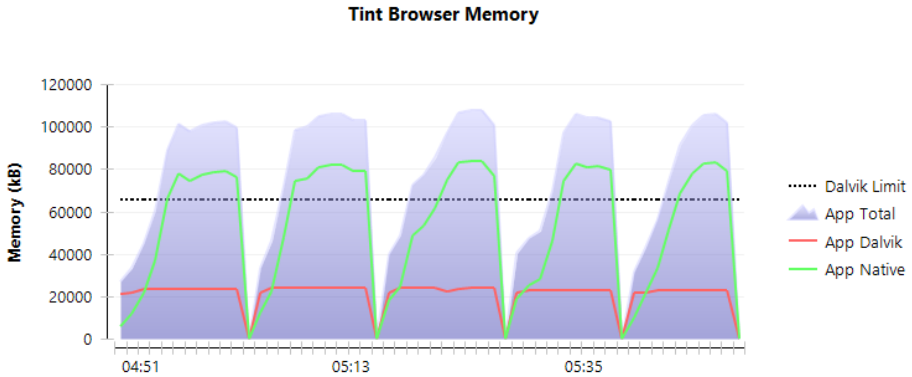
Fonte: Próprio autor

Figura 3.14 – Métrica de ocupação de memória da versão menos coesa



Fonte: Próprio autor

Figura 3.15 – Métrica de ocupação de memória da versão mais coesa

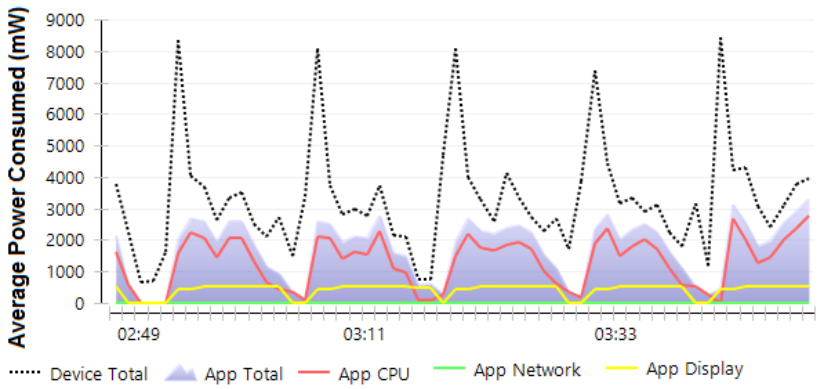


Fonte: Próprio autor

Assim como na métrica de ocupação de CPU, não é possível distinguir um impacto claro das mudanças realizadas pela refatoração na ocupação de memória exigida pela exibição das páginas. Assim como na métrica avaliada anteriormente, ocorrem algumas variações de ocupação entre execuções de uma mesma versão de código.

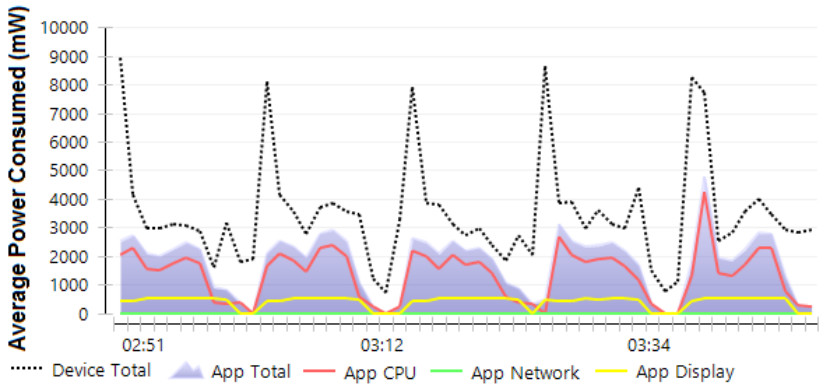
Por fim, as Figuras 20 a 22 apresentam os dados da métrica de consumo de potência para as três versões do Tint Browser.

Figura 3.16 – Métrica de consumo de potência da versão base



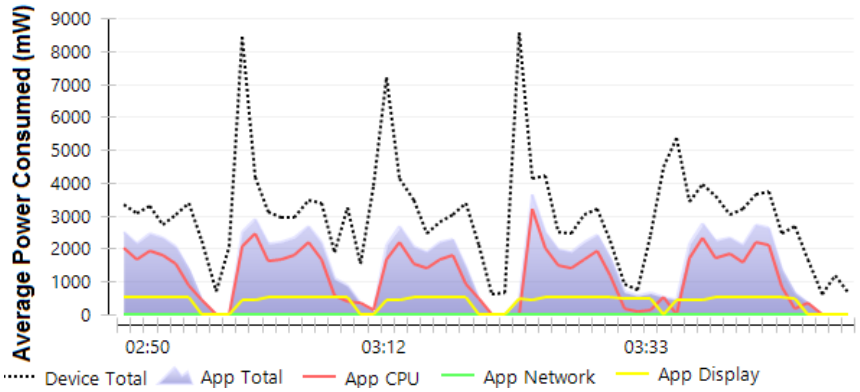
Fonte: Próprio autor

Figura 3.17 – Métrica de consumo de potência da versão menos coesa



Fonte: Próprio autor

Figura 3.18 – Métrica de consumo de potência da versão mais coesa

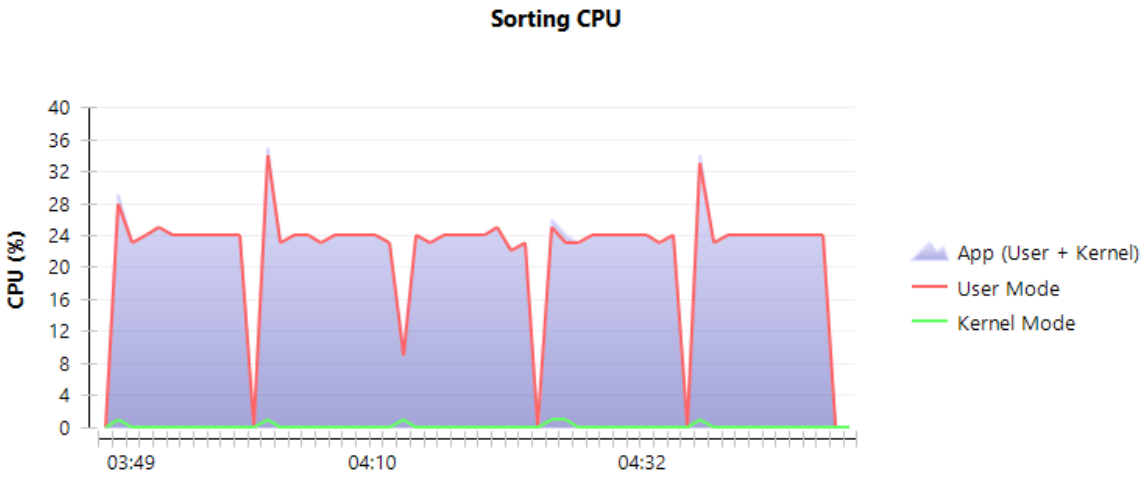


Fonte: Próprio autor

Assim como nas métricas anteriores, o consumo de potência variou entre as execuções de uma mesma versão de código. Devido a essas variações, não é possível determinar que uma das versões consuma menos potência que outras.

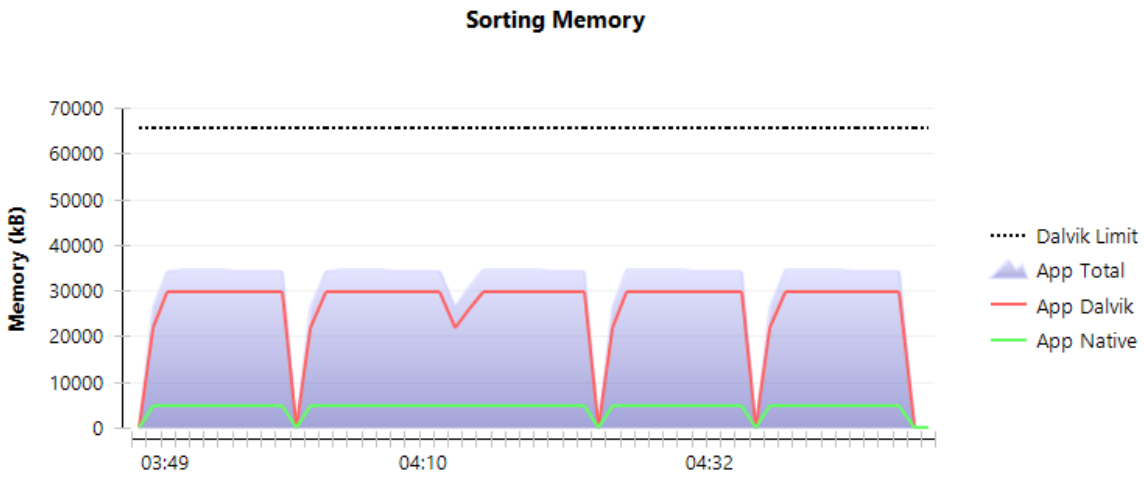
Essa indefinição dos dados e, principalmente, a constante variação dos valores em uma mesma configuração de experimento (mesma carga, mesmo dispositivo e mesma versão de código) dificulta sobremaneira uma avaliação confiável do desempenho esperado para uma aplicação. Foram investigadas algumas hipóteses para este resultado: a primeira está relacionada com a variação entre execuções de uma mesma versão do código. Para reduzir tal variação, uma série de medidas foram tomadas: além da limpeza de todos os dados da *cache* do navegador o servidor local de páginas foi substituído por páginas armazenadas como arquivos html dentro do próprio dispositivo, mas as variações se mantiveram. A segunda hipótese está relacionada ao fato de que diferentes versões da aplicação obtiveram desempenho médio bastante similares. Estima-se que as classes refatoradas representam uma parcela muito pequena da aplicação, embora lidem com uma funcionalidade essencial da mesma. Para confirmar esta hipótese, foi implementada uma pequena aplicação que consiste em uma versão do algoritmo *QuickSort* para ordenar um *array* de números gerados aleatoriamente. Então, as mesmas métricas foram coletadas para essa aplicação e estão apresentadas na Figura 3.19 (CPU), na Figura 3.20 (ocupação de memória) e na Figura 3.21 (potência consumida).

Figura 3.19 – Métrica de ocupação de CPU *QuickSort*



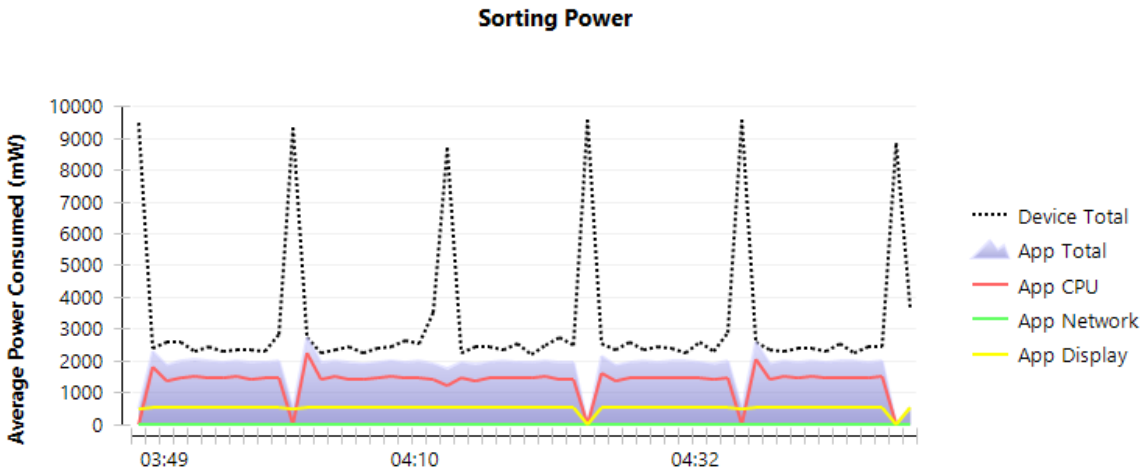
Fonte: Próprio autor

Figura 3.20 – Métrica de ocupação de memória *QuickSort*



Fonte: Próprio autor

Figura 3.21 – Métrica de consumo de potência *QuickSort*



Fonte: Próprio autor

Diferente do que foi observado nas versões de código do *Tint Browser*, as métricas da aplicação de *QuickSort* foram mais regulares entre as execuções de um mesmo experimento. No entanto, essa é uma aplicação bastante simples e sem grande dependência de alguma biblioteca, pois o algoritmo foi implementado à mão e não foi utilizada a versão disponível nas bibliotecas Java. Esses resultados foram confirmados (a mesma medida de ocupação de memória foi obtida) usando uma estratégia de coleta de métricas sem o uso do *Littleeye*, mas através de instrumentação do código.

Finalmente, comparamos a métrica de ocupação de memória entre uma aplicação que simplesmente exibe uma página *web* utilizando a biblioteca *WebKit* (também utilizada pelo *Tint Browser*) e a aplicação de ordenamento com o algoritmo *QuickSort*. A Figura 3.22 mostra essa comparação.

Figura 3.22 – Comparação da métrica de ocupação de memória

Fonte: Próprio autor

O gráfico de barra mostra a média das trinta execuções das aplicações e a barra de erro apresenta o índice de dispersão dessa média, intervalo semi-interquartil. Como pode ser observado, a aplicação baseada na biblioteca, embora extremamente simples, teve uma considerável variação entre as execuções, enquanto que a aplicação de ordenação tem

variação mínima. Tais variações podem ofuscar aquelas pequenas inseridas pelas diferentes refatorações de código. Avaliando-se detalhadamente o código do Tint Browser, observamos que a funcionalidade de visualização de páginas é implementada, de fato, pela biblioteca *WebKit* e que as classes refatoradas realizam tarefas secundárias desta funcionalidade. Esse resultado indica que i) o desempenho do Tint Browser está muito relacionado ao desempenho da biblioteca; ii) o desempenho da biblioteca não é tão previsível quanto desejado.

Os desenvolvedores de aplicações móveis têm uma variedade de bibliotecas para agilizar o desenvolvimento e fornecer mais funcionalidades a suas aplicações. Entretanto, tais bibliotecas podem influenciar de forma decisiva o desempenho da aplicação final em um determinado dispositivo. Sendo assim, as aplicações fortemente baseadas em bibliotecas, como o *Tint Browser* utilizado nessa avaliação, podem não ser claramente beneficiadas quando o desenvolvedor realiza algumas modificações na estrutura do seu código. Em contrapartida, o desenvolvedor pode não ser penalizado por seguir estilos ou padrões de programação que tornem seu código mais legível, modularizado e mais facilmente manutenível, pois, como pode ser visto no trabalho, tornar as classes do *Tint Browser* mais coesas não decaiu seu desempenho nas métricas avaliadas.

Com base nos resultados obtidos nesses dois experimentos e também pela avaliação dos trabalhos relacionados à avaliação de desempenho de plataformas móveis, as seguintes conclusões podem ser extraídas:

- O sistema operacional e as bibliotecas possuem uma considerável influência no desempenho das aplicações móveis;

- ao contrário de outras aplicações embarcadas, aplicações móveis não possuem um espaço tão grande para exploração de diferentes alternativas de decomposição em módulos. De fato, pelo menos para a plataforma *Android*, a estrutura geral de uma aplicação é definida pela plataforma e a lógica envolvida nos módulos definidos pelo usuário tende a ser simples, enquanto grande parte da funcionalidade está de fato implementada em bibliotecas disponíveis no SDK;

- a suposição inicial desta pesquisa de que estratégias de decomposição (em alto nível) do sistema poderiam impactar o desempenho final de uma aplicação mostrou-se inválida ou, pelo menos, imprecisa devido à dominância das bibliotecas na implementação.

Por outro lado, o desempenho de uma biblioteca em diferentes configurações de dispositivos pode variar consideravelmente e impactar negativamente o desempenho da aplicação final. Há, porém, diferentes alternativas de bibliotecas que podem ser usadas pelo desenvolvedor para a mesma funcionalidade. Sendo assim, muitas variáveis compõem o projeto de aplicação móvel e o desenvolvedor possui questões que demandam mais esclarecimento no início do projeto de desenvolvimento:

- Minha aplicação utiliza eficientemente a bateria?

- A biblioteca vai impactar o desempenho da aplicação?

- Minha aplicação é afetada por diferentes tipos de conexão de redes?

- Um dispositivo com *hardware* básico executará minha aplicação de forma agradável ao usuário final?

O desenvolvedor *mobile* deseja saber, de forma rápida e simples, como as diferentes bibliotecas ou estruturas de código se comportam em diferentes configurações de *hardware*. Com essa informação, diferentes alternativas de bibliotecas, estruturas de dados e recursos do *hardware* podem ser avaliadas para prever o comportamento não funcional da aplicação em

uma variedade de dispositivos. Conhecer previamente o desempenho de uma determinada decisão de projeto pode alterar a implementação do aplicativo visando melhorar sua adequação a um dispositivo alvo.

Essas necessidades do desenvolvedor e a falta de ferramentas existentes que possibilitem estimar o desempenho de uma aplicação móvel levaram ao desenvolvimento de uma abordagem focada em desenvolver um *framework* para gerar programas de testes de desempenho com base nas necessidades de uma específica e, assim, estimar os requisitos não funcionais de uma aplicação móvel. Essa abordagem é detalhada no próximo capítulo.

4 *BENCHMARKS* ORIENTADOS À APLICAÇÃO PARA SISTEMAS MÓVEIS

Conforme discutido nos Capítulos 1 e 2, *benchmarks* genéricos são uma maneira simples e eficaz para comparar dispositivos móveis do ponto de vista do usuário final do dispositivo. Por exemplo, como apresentado na Tabela 1.1 (página 15), o dispositivo Galaxy S3 pode ser considerado o melhor investimento, pois apresenta a melhor pontuação nos três *benchmarks* utilizados. Porém, pode-se observar ainda que a classificação dos outros aparelhos varia com o *benchmark*: Xperia U é classificado como melhor do que o Atrix em dois *benchmarks* (*Quadrant* e *Vellamo),* enquanto o *Antutu* classifica o Atrix como melhor que o Xperia U. Ainda assim, para o usuário final, esta informação é suficiente para a tomada de decisão, pois o raciocínio é, normalmente, de se optar pelo melhor aparelho dentro de uma faixa de preço, ou seja, há menos variáveis envolvidas na decisão.

Porém, do ponto de vista do desenvolvedor, essa avaliação é pouco útil ou, no mínimo, insuficiente. De fato, a única informação consistente, a partir dos *benchmarks,* é que o Galaxy S3 parece ser o melhor dispositivo do momento. Como essa informação ajuda o desenvolvedor? Ele pode escolher fazer sua aplicação voltada apenas para este dispositivo. Porém, se este é um dispositivo novo no mercado, poucos usuários poderão usar a aplicação e o desenvolvedor perde alcance em volume de vendas. Sendo um dispositivo com mais recursos, seu preço é maior e, novamente, o número de usuários é reduzido. Por outro lado, se o desempenho da aplicação feita para o S3 não for adequado em outros dispositivos mais acessíveis ou que já estão no mercado há algum tempo, o desenvolvedor perde o mercado já disponível além da confiança do consumidor nos seus produtos. Por fim, a variação na classificação dos demais dispositivos não permite ao desenvolvedor estabelecer uma estratégia alternativa ou um conjunto razoável de dispositivos para dar suporte.

Outra questão relevante dessas aplicações de *benchmark* genéricas é a sua relação com as reais necessidades de uma aplicação específica. Em um *benchmark* genérico o conjunto de testes é predefinido e não parametrizável, ou seja, as operações realizadas nos dispositivos bem como a carga de trabalho são fixas. No entanto, essas características podem ser distintas da aplicação para a qual se deseja saber o desempenho. O resultado da execução do *benchmark* não explicita, na maioria dos casos ou de forma detalhada, porque um aparelho teve melhor ou pior desempenho do que outro. A discrepância pode ter sido causada por uma operação, por exemplo, que não será usada na nova aplicação ou, do contrário, o bom desempenho de um aparelho pode se dever ao fato de que o programa de teste usou uma carga de trabalho mais leve do que a aplicação em mente. Em outras palavras, os testes executados por um *benchmark* genérico podem ser muito distintos dos requisitos de processamento de uma aplicação que reúna apenas um subconjunto das funcionalidades testadas.

É possível que o desenvolvedor tenha interesse em um dispositivo específico, como é o caso de uma aplicação móvel para sistemas corporativos. Nesse contexto, é comum que o empregador (cliente do desenvolvedor da aplicação móvel) forneça aparelhos a seus funcionários como ferramenta de trabalho, situação recorrente em organizações que realizam pesquisa de campo, vendas no atacado. Neste caso, ou o cliente já possui os aparelhos ou avalia qual o de melhor custo-benefício para o funcionário. A partir dos *benchmarks* genéricos, o desenvolvedor apenas verifica qual a posição de um determinado aparelho no *ranking* de dispositivos e se sua plataforma-alvo não estiver bem classificada, não há nenhuma informação adicional que o ajude a melhorar a experiência do usuário final naquele dispositivo.

Com o objetivo de auxiliar o desenvolvedor a verificar se as possíveis operações da sua aplicação terão um desempenho adequado em determinado dispositivo, e ainda auxiliar interessados na escolha do dispositivo de melhor custo benefício para execução de uma determinada aplicação ainda não desenvolvida, esta dissertação propõe um *framework* para estimar requisitos não funcionais, normalmente relacionados à noção de desempenho, de aplicações móveis.

Esse *framework* se baseia na junção da facilidade e praticidade de avaliação através de *benchmarks* com a especialização de um processo de avaliação de desempenho bem estruturado. O método proposto consiste na geração semiautomática de *benchmarks* orientados à aplicação, aqui também chamados de programas de teste. A partir desse *framework*, o desenvolvedor pode gerar pequenas aplicações selecionando diversas operações atômicas previstas para a aplicação real e configurando corretamente a carga de trabalho desejada. Essas aplicações são executadas nos dispositivos alvos para avaliar o desempenho.

Para se construir um *benchmark* orientado à aplicação são necessárias algumas peças que juntas comporão a aplicação final de teste, são essas:

- Conjunto de operações independentes, passíveis de composição e parametrizáveis, com granularidade menor que de uma aplicação completa;

- Estratégia de seleção de operações e carga de trabalho;

- Mecanismos de coleta de métricas de interesse;

- Estrutura para geração do programa de teste.

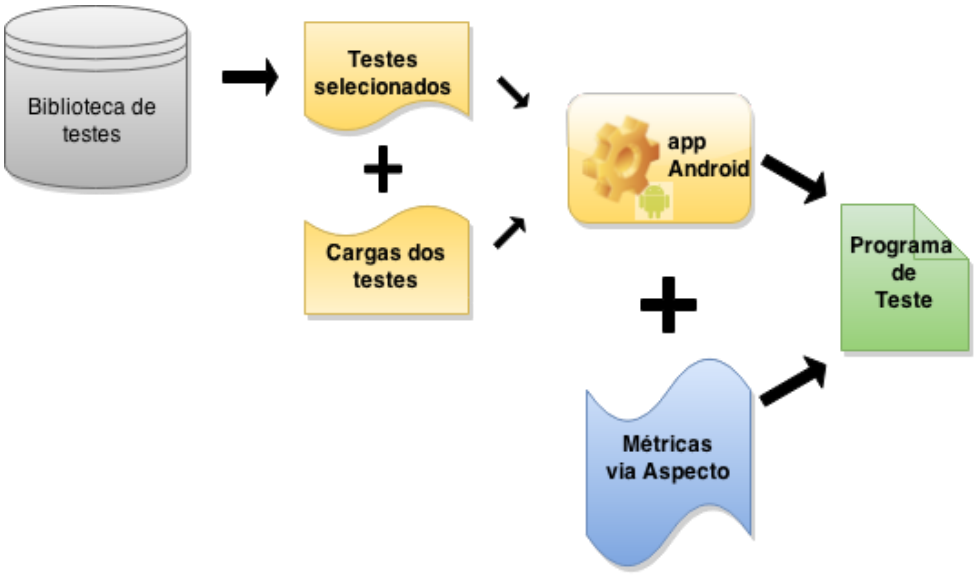
Com a utilização dessas peças, o usuário do *framework* (desenvolvedor) pode selecionar quais operações sua aplicação terá, determinar a carga de cada uma das operações, escolher quais métricas serão avaliadas e, assim, gerar sua aplicação de teste orientada às necessidades daquela a ser desenvolvida.

As subseções a seguir detalham a estrutura geral do *framework* proposto, que foi desenvolvido de forma a facilitar seu incremento.

4.1 Estrutura

O intuito do *framework* proposto é possibilitar a criação semiautomática de aplicações de teste para estimar requisitos não funcionais de aplicações móveis. As de teste são formadas por pequenos trechos de código (operações) que exercitam determinadas funcionalidades do dispositivo.

Figura 4.1 – Esquema de criação de um programa de teste



Fonte: Próprio autor.

O conjunto de trechos de código desse *framework* é agrupado no que chamamos de biblioteca de testes, composta por categorias que subdividem as operações com relação aos aspectos dos dispositivos que são exercitados de forma principal por uma operação. Note que, certamente, mais de um componente de *hardware* (não apenas a CPU) é usado quando se deseja fazer, por exemplo, uma operação matemática. No entanto, essa operação específica usa de forma mais intensiva a CPU e, portanto, é mais adequada para a avaliação deste recurso. Dessa forma, as categorias disponíveis na biblioteca de testes indicam os recursos de *hardware* passíveis de avaliação em um dispositivo. Dentro de cada categoria, subcategorias indicam que aquele recurso pode ser usado de diferentes formas, como será detalhado abaixo. Por fim, as operações que melhor avaliam um determinado uso de um aspecto específico do *hardware* são inseridas na respectiva categoria.

Para montar a biblioteca de testes, foi realizado um estudo de trabalhos desenvolvidos para avaliação de desempenho e aplicações de *benchmark* de código aberto. A partir desse estudo, foram selecionadas diversas operações que estimulam determinados componentes dos dispositivos. Além disso, foram incluídas algumas operações comuns em aplicações móveis, mas que muitas vezes não são testadas em *benchmarks* genéricos, como envio de e-mails, consumo de *web services*, *download* de arquivos, etc.

Inicialmente a biblioteca é subdivida nas seis categorias abaixo:

- **CPU**: composta por operações que focam em exercitar características do processador do dispositivo como operações de inteiros e pontos flutuantes.

- **Memória**: possui operações de manipulação da memória real e também do *heap* da máquina virtual.

- **Rede**: compreende operações que avaliam os mecanismos do dispositivo para comunicar-se com a *internet*, enviando ou recebendo dados.

- **GPU**: composta por operações para testar o desenho de gráficos 2D e 3D.

- **Armazenamento**: compreende operações de manipulação de dados através do sistema de arquivos do dispositivo ou acionando o sistema de banco nativo.

- **Sensores**: compreende operações que exercitam os sensores atualmente disponíveis nos dispositivos móveis, categoria que contém apenas a operação com GPS.

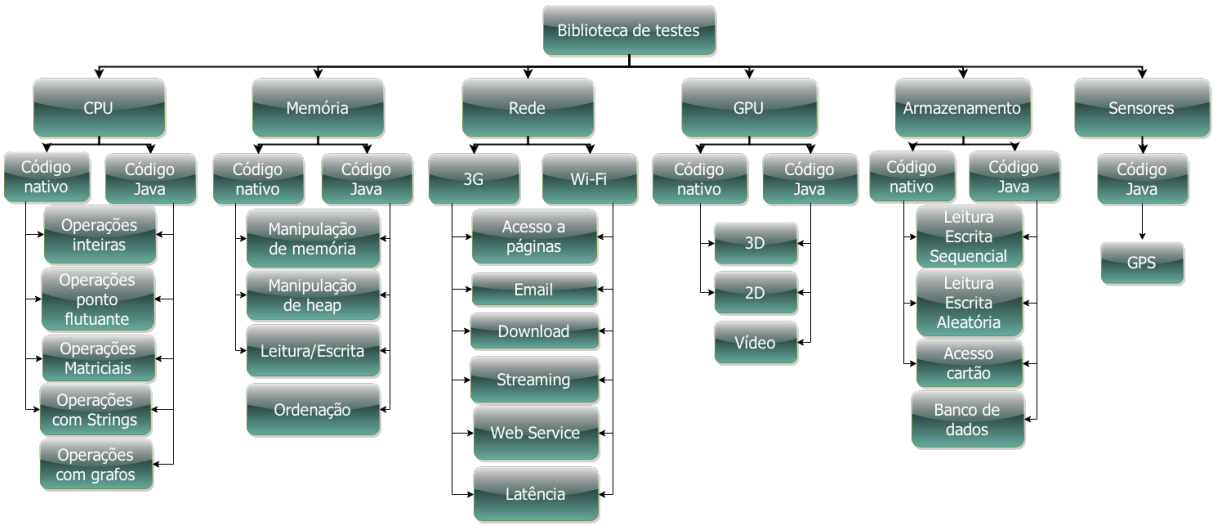
É importante ressaltar que a classificação proposta não é exaustiva e está fortemente baseada nos diversos trabalhos e *benchmarks* que disponibilizam trechos ou descrições de código usado em avaliação de desempenho.

Um aspecto comum em trabalhos de avaliação de desempenho é a comparação entre o código executado sobre uma máquina virtual e o código C/C++, por exemplo, Lin et al. (2011). Essa possibilidade advém do fato de que muitos desenvolvedores já possuem códigos prontos, normalmente buscando melhor desempenho em C/C++ e gostariam de reutilizá-los em novas aplicações móveis (aspecto previsto nesse *framework* através de subcategorias). Assim, as categorias CPU, Memória e Armazenamento possuem algumas operações que foram implementadas tanto na linguagem Java, quanto na linguagem C++, chamado de código nativo. Essa dupla implementação possibilita que, ao criar uma aplicação de teste, uma mesma operação seja avaliada quando executada sobre a máquina virtual ou não.

O acesso à *internet* em dispositivos móveis pode ser realizado através de redes de dados de uma operação de telefonia ou através de uma rede sem fio (*WiFi*), podendo influenciar o desempenho da operação realizada como, também, o consumo de bateria do dispositivo. O código das aplicações não necessita, a princípio, de modificação para acessar uma rede ou outra, uma vez que isso pode ser considerado transparente para a aplicação. Porém, o uso de uma ou outra possibilidade pode interferir no desempenho da aplicação. Assim, na definição da aplicação de testes, o usuário deve selecionar o tipo de rede que será utilizada nas operações de acesso à *internet*.

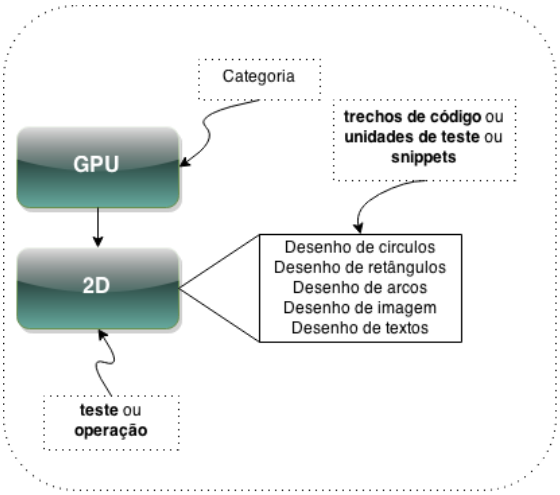
A Figura 4.2 mostra uma representação da biblioteca de testes proposta que conta, atualmente, com 68 unidades de teste distribuídas entre 11 subcategorias e 23 operações, também chamadas de testes. A estrutura sobre a qual a biblioteca foi construída possibilita a fácil inserção de novas categorias e operações que, certamente, serão necessárias à medida que novos elementos de *hardware* são incluídos nos dispositivos. A Figura 4.3 mostra uma definição dos elementos que compõem a biblioteca.

Figura 4.2 – Estrutura da biblioteca de testes



Fonte: Próprio autor.

Figura 4.3 – Definição de elementos da biblioteca



Fonte: Próprio autor.

4.2 Biblioteca de testes

A biblioteca de testes é formada por trechos de código (unidades de teste) que são executados visando exercitar determinados elementos do dispositivo onde a aplicação está instalada. Cada operação pode necessitar de parâmetros distintos, mas, de modo geral, o principal parâmetro é denominado *level*, utilizado para graduar a intensidade do teste a ser realizado. A construção modular da biblioteca permite que novos testes sejam facilmente inseridos visando torná-los, cada vez mais, próximos do comportamento de uma aplicação real. As subseções a seguir apresentam detalhes de cada um dos testes disponíveis atualmente na biblioteca.

Nessa versão inicial, o *framework* é composto por 33 pacotes de código que organizam aproximadamente 150 classes, sendo *edu.performance.test* o pacote mais geral e que engloba todos os demais. Além disso, há classes abstratas que são utilizadas como base para os testes, de forma que a maioria dos sub pacotes agrupam as operações de teste. No entanto, existem dois pacotes especiais: o *.util* que armazena classes que são utilizadas como auxiliares no processo de construção da aplicação de teste e o *.domain* que armazena algumas classes de domínio para criar uma interface gráfica para seleção dos testes.

De forma mais detalhada, temos a seguinte relação entre os pacotes e as categorias da biblioteca:

- CPU:
    - floatoperation e nativo.floatoperation;
    - integeroperation e nativo.integeroperation;
    - stringoperation e nativo.stringoperation;
    - graphoperation.
- Memória:
    - memoryoperation e nativo.memoryoperation;
    - sortingoperation.
- Rede:

- weboperation;
- webserviceoperation;
- weblatencyoperation.
- streamingoperation;

- GPU:
  - graphicoperation;
  - graphicoperation.twod;
  - graphicoperation.threed;
  - graphicoperation.threed.gltron.
  - screenoperation;
- Armazenamento:
  - filerandomoperation;
  - filesequentialoperation;
  - nativo.fileoperation;
  - databaseoperation;
  - databaseoperation.dominio;
  - databaseoperation.util.
- Sensores:
  - locationoperation.

As subseções a seguir detalham as operações disponíveis na biblioteca do *framework*.

4.2.1 Teste de banco de dados

O teste de operações de banco de dados está no pacote *edu.performance.test.databaseoperation*, composto por mais dois sub pacotes. O primeiro é denominado *domain* e engloba as classes que representam as entidades manipuladas no banco de dados. O outro, é o *util*, composto por duas classes para carregar, a partir de arquivos xml, os dados utilizados no teste. Atualmente os arquivos lidos são fixos (denominados "Tarefas.xml" e "listas.xml").

Esse pacote engloba quatro unidades de teste de banco de dados, a saber: inserção, busca, atualização e exclusão de dados. Nesse teste, o parâmetro *level* é do tipo número inteiro e define a quantidade de dados que serão manipulados nas operações do banco. Na atual versão, os dados manipulados são de dois tipos Listas (simples e contém apenas os atributos *id* e nome da lista) e Tarefas (mais complexo e possui vários atributos incluindo relações com outras entidades dentro do banco).

Esse conjunto de testes possui uma métrica específica para operações de banco, o *throughput*, obtida pela divisão do valor do parâmetro *level* pelo tempo gasto para realizar a operação do banco de dados.

4.2.2 Teste de *download* de arquivos

O teste de *download* de arquivos está no pacote *edu.performance.test.downloadoperation*. Como indica o nome, é realizado o *download* de um arquivo a partir de um site da *internet* e seu armazenamento feito em uma pasta local do dispositivo. Nessa versão inicial, a operação possui um *array* com endereços de três arquivos de tamanhos distintos (1MB, 10MB e 50MB), armazenados no site arquivos.oi.com.br. Caso o usuário deseje alterar os arquivos, basta fornecer novos endereços. O parâmetro *level* é do tipo inteiro e determina o índice do *array* que será acessado e, consequentemente, o tamanho do arquivo a ser transferido. Esse teste necessita que o parâmetro booleano *networktest* tenha o valor verdadeiro, pois é necessário que se tenha acesso à *internet* para realizar a transferência de arquivo.

4.2.3 Teste de manipulação de arquivos

O teste de manipulação de arquivos está no pacote *edu.performance.test.fileoperation*. São realizadas quatro unidades de teste: leitura e escrita de arquivos de modo sequencial e leitura escrita de arquivos de modo aleatório. O parâmetro *level_filename* fornece o endereço do arquivo de texto que será manipulado e o tamanho do arquivo é usado para graduar a intensidade do teste.

Essa operação possui um *array* de posições que é utilizado pelas unidades de teste de leitura e escrita aleatória para determinar onde essas operações serão realizadas. O parâmetro *level* é do tipo inteiro e contém o índice do *array* de posições. O parâmetro *stretch* é utilizado nas operações de escrita e contém um trecho de texto que deve ser inserida no arquivo utilizado no teste. A construção atual da biblioteca disponibiliza três arquivos de texto com tamanhos diferentes, *small*, *medium* e *large*. Esse último possui a obra *The Adventures of Sherlock Holmes*[22] de Sir Arthur Conan Doyle em inglês e tem cerca de 100.000 linhas.

4.2.4 Teste de operações de ponto flutuante

O teste de operações com ponto flutuante está no pacote *edu.performance.test.floatoperation*. Os trechos de código executados nesse teste são cálculos com números ponto flutuante e o parâmetro *level* é usado como dado de entrada para cada uma das unidades de teste. Os cálculos realizados são: cálculo da raiz quadrada, utilizando o método de Newton; cálculo do seno de um ângulo utilizando a biblioteca Math; cálculo de Pi utilizando o método MonteCarlo; resolução de um polinômio cúbico; conversões entre graus e radianos e o *benchmark* Linpack que resolve um sistema linear nXn do tipo Ax = b, onde o n é fornecido pelo parâmetro *level*. É importante ressaltar que esse parâmetro está atualmente limitado em 1000.

Os trechos de código de resolução de polinômio cubico, raiz quadrada e conversão de ângulos foram inspiradas no código do *benchmark* para CPU, Mibench Guthaus et al. (2001).

[22] http://www.gutenberg.org/ebooks/1661

4.2.5 Teste de gráficos em 2D

Esse pacote é composto por cinco unidades de teste para desenhar formas em 2D sobre superfícies que estendem a classe *android.view.SurfaceView* e implementam a interface *android.view.SurfaceHolder*. Essas unidades de teste são baseadas em uma aplicação de *benchmark* de código aberto denominada 0xBench[23]. Os trechos de código para exercitar o desenho de gráficos 2D estão no pacote *edu.performance.test.twod*.

4.2.5.1 Desenho de arcos

Essa unidade de teste desenha arcos coloridos na tela do dispositivo que está sendo testado, através do método *drawArc* da biblioteca *android.graphics.Canvas*. O parâmetro *level* é do tipo inteiro e determina diretamente a quantidade de formas desenhadas de forma que seu valor é utilizado para calcular o raio dos arcos. Assim, quanto maior o valor de *level* maior a quantidade de arcos e menor o raio de cada arco. As cores dos arcos atualmente são escolhidas aleatoriamente.

4.2.5.2 Desenho de círculos

Essa unidade de teste desenha círculos coloridos na tela do dispositivo testado através do método *drawCircle* da biblioteca *android.graphics.Canvas*. O parâmetro *level* é do tipo inteiro e determina a quantidade de círculos concêntricos desenhados a cada chamada do método de desenho. Quanto maior a quantidade de círculos maior a exigência para a unidade de processamento para desenhar as formas. Atualmente, as cores dos círculos são escolhidas aleatoriamente.

4.2.5.3 Desenho de imagens

Essa unidade de teste carrega uma imagem e desenha suas réplicas movimentando na vertical da tela do dispositivo através do método *drawImage* da biblioteca *android.graphics.Canvas*. O parâmetro *level* é do tipo inteiro e determina a quantidade de imagens desenhadas, já o valor é utilizado para determinar as dimensões da imagem. Assim, quanto maior o valor de *level* maior a quantidade de réplicas desenhadas e menor o tamanho dessas réplicas.

4.2.5.4 Desenho de retângulos

Essa unidade de teste desenha retângulos coloridos na tela do dispositivo. A operação gera uma lista de retângulos com cores, dimensões e posicionamentos aleatórios, sendo cada um

[23] https://code.google.com/p/0xbench/

deles uma instância da classe *Rectangle*. Os retângulos são então desenhados através do método *drawRect*, da biblioteca *android.graphics.Canvas*. O parâmetro *level* é do tipo inteiro e determina o tamanho máximo da lista de retângulos desenhados. Quando o tamanho máximo é atingido, a lista é limpa e começa a geração de outros novos.

4.2.5.5 Desenho de textos

Essa unidade de teste desenha palavras em posições e cores aleatórias da tela do dispositivo. Essa operação possui um array de *strings*, atualmente com 10 posições. O parâmetro *level* é do tipo inteiro e determina quantas palavras diferentes serão desenhadas através do método *drawText* da biblioteca *android.graphics.Canvas*.

4.2.6 Testes de gráficos 3D

Esse conjunto de unidades de teste é formado majoritariamente pelo desenho de cubos utilizando OpenGL 1.2 e OpenGL 2. Essas operações estão localizadas no pacote *edu.performance.test.threed*.

4.2.6.1 Desenho de cubos coloridos com OpenGL 2

Essa unidade de teste consiste em desenhar cubos giratórios na tela do dispositivo, tendo suas faces coloridas e determinadas via código da operação. O parâmetro *level* é do tipo inteiro e determina a quantidade de cubos que são desenhados na tela. Nesse método, é utilizada a versão 2 do OpenGL e são usados Vertex e Shaders para definir as cores e as dimensões dos cubos.

4.2.6.2 Desenho de cubos com texturas utilizando OpenGL 1.2

Essa unidade de teste consiste em carregar uma textura e desenhar cubos com ela em suas faces. É necessário que a textura esteja dentro da aplicação de teste para que possa ser fornecido o identificador dessa textura para a operação. O parâmetro *level* determina a quantidade de cubos que serão desenhados e também ajusta a profundidade do ambiente de modo a permitir a visualização desses. Para essa operação, é utilizada a versão 1.2 do OpenGL.

4.2.6.3 Desenho de cubos com textura utilizando OpenGL 2

Essa unidade de teste consiste em carregar uma textura e desenhar cubos com ela em suas faces. É necessário que a textura esteja dentro da aplicação de teste, para que assim possa ser fornecido o identificador dessa textura para a operação. O parâmetro *level* determina a quantidade de cubos que serão desenhados e também ajusta a profundidade do ambiente, de modo a permitir a visualização dos cubos. Para essa operação, é utilizada a versão 1.2 do OpenGL.

4.2.6.4 Desenho de motos do Jogo GLTron

Essa unidade de teste foi adicionada à biblioteca durante os testes de validação. Como não foi possível, com os testes existentes, estimar as métricas do jogo GLTron, foi criado um teste que utiliza algumas das classes desse jogo. Essa operação consiste em carregar um modelo de objeto (motocicleta) e desenhá-lo em várias posições de um plano. O parâmetro *level* determina a quantidade de motos desenhadas no plano. As classes desse jogo estão no sub pacote denominado *gltron*.

4.2.7 Teste de operações com grafos

A operação com grafos está no pacote *edu.performance.test.graphoperation*. Esse teste consiste em aplicar o algoritmo para encontrar o menor caminho em um grafo proposto por Dijsktra. O argumento *level_filename* designa o caminho para o arquivo que contém o grafo que será percorrido pelo algoritmo. Nessa versão, temos três arquivos de texto que contém grafos de tamanhos distintos, a saber: tiny_g.txt; medium_g.txt e big_g.txt. Caso queira utilizar outro grafo como parâmetro, esse deve ser ponderado e direcional. Essa operação busca exercitar os dispositivos para traçar rotas, atividade comum em aplicações de GPS, e baseia-se no Mibench.

4.2.8 Teste de operações com números inteiros

A operação com inteiros está no pacote *edu.performance.test.integeroperation*. As unidades de teste executadas são cálculos com números inteiros. O parâmetro *level* é do tipo inteiro e determina os limites para o cálculo de números primos e da sequência de Fibonacci. O teste com o cálculo de números primos é realizado através da implementação do algoritmo do crivo de Erastóstenes e o com cálculo da sequência de Fibonacci é iterativo. Outro teste realizado é a implementação do algoritmo recursivo de Ackermann-Peter que consiste no recebimento de dois números inteiros e faz chamadas recursivas decrementando o valor dos parâmetros de entrada. No entanto, a exigência por recursos computacionais nesse algoritmo aumenta rapidamente mesmo com pequenos parâmetros inteiros. Nessa versão existe um controle que impede que os parâmetros de entrada para esse algoritmo sejam maiores que ‘2’.

4.2.9 Teste de GPS

Essa operação está no pacote *edu.performance.test.locationoperation* e consiste em acionar o sensor de GPS do dispositivo e coletar o posicionamento atual. Como esse dado também pode ser obtido com auxílio de uma rede de *internet*, esse teste necessita que o parâmetro *networktest* tenha valor verdadeiro.

4.2.10 Teste de envio de *e-mail*

O teste de envio de *e-mails* está no pacote *edu.performance.test.mailoperation*. Essa operação envia um *e-mail* através da biblioteca *javax.mail*, sendo que nesse teste o parâmetro *level_str*, do tipo cadeia de caracteres determina o corpo da mensagem a ser enviada. Outro parâmetro do tipo cadeia de caracteres que é obrigatório é o *destination* onde deve ser informado o endereço que receberá o *e-mail*. O parâmetro booleano *notify* determina se deve ser mostrada uma notificação na barra de notificações do *Android* sobre o processo de envio. O parâmetro opcional *level_filename* determina um caminho para um arquivo que será enviado como anexo. Sendo um teste da categoria Rede, o envio de *e-mail* necessita que o parâmetro booleano *networktest* tenha valor verdadeiro.

4.2.11 Teste de manipulações de memória

O teste de manipulação de memória está no pacote *edu.performance.test.memoryoperation.* As unidades de teste consistem em solicitar memória para estruturas de dados e manipulá-las. São executados os seguintes trechos de código: criação de um array e um *ArrayList* de objetos e preenchidas com instâncias da classe *Object*. Posteriormente, são criadas estruturas de dados semelhantes, mas vazias, e então é realizada cópia do array e do *ArrayList* de objetos para as novas estruturas de dados. O parâmetro *level* é do tipo inteiro e determina o tamanho das estruturas de dados que são criadas e, consequentemente, afeta a área necessária para armazená-las.

4.2.12 Teste de manipulação de arquivos em C++

O teste de manipulação de arquivos em C++ está no pacote *edu.performance.test.nativo.fileoperation* e é semelhante ao pacote de teste de manipulação de arquivos Android, mas a manipulação é realizada por um código escrito em C++ e acionado através das chamadas de métodos nativos que estão no arquivo file.cpp. Para acessá-los são utilizadas as JNI.

4.2.13 Teste de operações de ponto flutuante em C++

O teste com ponto flutuante em C++ está no pacote *edu.performance.test.nativo.floatoperation* sendo semelhante ao pacote de teste de operações de ponto flutuante em JAVA, porém nesse pacote os algoritmos são implementados em C++ e acionado através das chamadas de métodos nativos que estão no arquivo float.cpp. Para acessá-los são utilizadas as JNI.

4.2.14 Teste de operações em números inteiros em C++

O teste com números inteiros em C++ está no pacote *edu.performance.test.nativo.integeroperation*. Tal teste é semelhante ao pacote de teste de operações com números inteiros em JAVA, mas nesse pacote os algoritmos são implementados em C++ e acionado através das chamadas de métodos nativos que estão no arquivo integer.cpp. Para acessá-los são utilizadas as JNI.

4.2.15 Teste de operações na memória em C/C++

O teste de operações na memória está no pacote *edu.performance.test.nativo.memoryoperation.* Nesse teste são realizadas operações de alocação, cópia de *arrays* e liberação da memória alocada para o array. São utilizados métodos *malloc*, *memcpy* e *free*, respectivamente que estão codificados no arquivo memory.cpp e para acessá-los são utilizadas as JNI.

4.2.16 Teste de operações com *strings*

O teste com *strings* está no pacote *edu.performance.test.stringoperation*. Esse teste consiste em localizar sub cadeias dentro de um texto, concatenar cadeias e substituir caracteres. Exige três parâmetros adicionais, o *searchable* é do tipo cadeia de caracteres e contém o endereço do arquivo que é utilizado para carregar o texto onde será buscada a sub cadeia e também onde serão substituídos os caracteres, o *snippets* é do tipo *array* de cadeia de caracteres e possui sub cadeias que serão buscadas no texto e também serão utilizadas para concatenação. Por padrão, é fornecido um *array* de cadeias com nove delas de tamanhos diferentes e que podem estar ou não dentro do arquivo. O parâmetro *level* é do tipo inteiro e determina o índice do *array* de cadeias a serem buscadas.

4.2.17 Teste de operações com *strings* em C++

O teste com *strings* está no pacote *edu.performance.test.nativo.stringoperation* e é semelhante ao pacote de teste de operações com *strings* em JAVA. No entanto, nesse pacote os algoritmos são implementados em C++ e acionado através das chamadas de métodos nativos que estão no arquivo string.cpp e para acessá-los são utilizadas as JNI.

4.2.18 Teste de ordenação

O teste de ordenação de *arrays* está no pacote *edu.performance.test.orderingoperation*. Esse teste consiste na implementação de dois algoritmos de ordenação o *QuickSort* e *MergeSort*. É utilizado um *array* do tipo People gerado pseudo-aleatoriamente através de

semente pré-definida, e então ordenado baseado no atributo id utilizando um dos algoritmos. O parâmetro *level* é do tipo inteiro e determina o tamanho do *array* que será gerado.

4.2.19 Teste de *streaming* de vídeo

Esse teste está no pacote *edu.performance.test.streamingoperation* e consiste na exibição de um vídeo fornecido via uma URL da *internet*. É realizado com três classes de exibição de telas, *SurfaceView*, *GLSurfaceView* e *TextureView*, sendo que apenas essa última permite a coleta da métrica de *frames* por segundo e está disponível apenas a partir da versão 4.0 do *Android*. O parâmetro *level* é do tipo cadeia de caracteres e contém o endereço do vídeo que será exibido.

4.2.20 Teste exibição de vídeo

Esse teste consiste em utilizar a classe *StreamingTextureActivity* com um arquivo local e, assim, contabilizar a taxa de *frames* por segundo dessa exibição. O parâmetro *level_filename* é do tipo cadeia de caracteres e contém o endereço do arquivo de vídeo a ser exibido.

4.2.21 Teste de acesso a páginas *web*

O teste de acesso a páginas está no pacote *edu.performance.test.webviewoperation*. Consiste em acessar páginas *web* utilizando a biblioteca *WebKit*. Por padrão, essa biblioteca possui quatro páginas de *internet* como carga de trabalho, definida por meio de *clustering* na pesquisa realizada por Fernandes et al. (2014). Essa operação possui um *array* de cadeia de caracteres com o endereço dessas quatro páginas. O parâmetro *level* é do tipo inteiro e determina o índice do *array* que indica qual a página mais complexa que será acessada, mas todas as páginas mais simples também serão acessadas.

4.2.22 Teste de latência de página

O teste de latência das páginas está no pacote *edu.performance.test.weblatencyoperation*. Baseia-se na medição da latência de páginas *web*. Por padrão, utiliza-se a mesma carga de trabalho do teste de acesso a páginas *web*, assim como o parâmetro *level* também indica a página mais complexa que deve ser testada a partir de uma posição no *array* de páginas *web*.

4.2.23 Teste de consumo de *web service*

O teste de consumo de serviço *web* está no pacote *edu.performance.test.webserviceoperation*, tendo por finalidade acessar um endereço de serviço *web* com o protocolo SOAP que transmite dados através de arquivos xml. Após

realizar o acesso ao serviço, é feito um armazenamento do resultado recebido em um objeto do tipo *Document* para posterior acesso. O parâmetro *level_website* é do tipo cadeia de caracteres e contém o endereço do serviço *web*.

4.2.24 Teste exibição de telas

Esse teste, composto pela exibição de telas com objetos de interação como campos de texto, botões, *checkboxs,* está no pacote *edu.performance.test.screenoperation*. O parâmetro *level* é do tipo inteiro e seleciona entre três possíveis telas pré-definidas, com quantidade e tipos de objetos de interação distintos. O parâmetro *level_str* possibilita que seja selecionada a cor do fundo da tela que, por padrão, é a cor preta.

Cada um desses testes foi codificado sem a inclusão de diretivas de código para coleta das métricas do *framework*. Dessa maneira, é estabelecida uma modularidade entre as operações e coleta de métricas, possibilitando o reuso do código dos testes no desenvolvimento de aplicações finais. Para a coleta de métrica foi utilizada Programação Orientada a Aspectos (POA), apresentada na próxima seção.

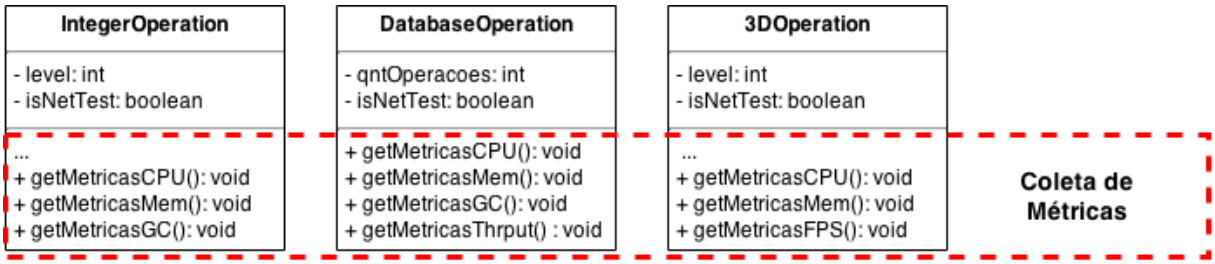
4.3 Programação Orientada a aspectos

A Programação Orientada a Aspectos (POA) tem a proposta de possibilitar a separação de interesses dentro do projeto de um *software*. Essa necessidade de separação de interesses visa uma maior modularização do código da aplicação, uma vez que os mecanismos de modularidade hierárquica da programação orientada a objetos não possibilitam modularizar todos os diferentes interesses dos sistemas atuais, Kiczales et al. (2001).

Com a dinamicidade do mercado de *software*, o foco das empresas na reutilização é uma prática constante para aumentar a lucratividade na produção de *software* e redução do *time-to-market*. Outra preocupação constante é o custo da manutenção do *software*. Assim, quanto mais simples e limpo um código, mais fácil é seu desenvolvimento e manutenção, Sommerville (2010). A POA é um dos padrões desenvolvidos na promoção do reuso de *software*, tornando o código das aplicações mais conciso e aumentando assim seu potencial para ser reutilizado.

A programação orientada a aspectos traz consigo o conceito de *crosscutting concerns* que pode ser compreendido como interesses transversais ao propósito da aplicação. Ainda de acordo com Kiczales et al. (1997), a POA fornece os mecanismos de linguagem para a estrutura de *crosscutting* assim como a Programação Orientada a Objetos (POO) provê o aparato necessário para o conceito de encapsulamento. A Figura 4.4 mostra um exemplo onde o interesse transversal pode ser evidenciado. Nesse cenário temos classes distintas para realização de determinadas operações com números inteiros, base de dados ou processamento de imagens 3D. No entanto, cada uma dessas classes realiza a coleta das métricas das operações realizadas, ou seja, a coleta de métricas é de interesse transversal a todas as classes.

Figura 4.4 – Exemplo de interesses transversais.



Fonte: Próprio autor, baseado em Kiczales et al. (2001).

De acordo com Kiczales et al. (1997), o objetivo da POA é unir duas vantagens ter um código eficiente e fácil de entender. De maneira geral, a programação orientada a aspectos propõe definir pontos de interesse na estrutura de código da aplicação e incluir nesses pontos algum trecho que represente um interesse transversal. Por exemplo, podemos definir como ponto de interesse todas as chamadas de um método do tipo *set*()*. Assim, podemos enviar uma mensagem ao terminal informando que o atributo foi alterado. O código referente à construção da mensagem de terminal é definido no aspecto e não no código do método *set()*.

Seguindo o paradigma de POA, algumas extensões de linguagem foram propostas. A seguir é abordada a extensão para linguagem Java, *AspectJ*.

4.3.1 AspectJ

De acordo com Kiczales et al. (2001), aspecto pode ser definido como a unidade modular para implementação do conceito de *crosscutting*. O processo de inclusão ao código da aplicação é denominado costura (*aspect weaving*), realizado pela ferramenta *weaver*. A declaração de um aspecto é composta de: *joinputs*, que são os pontos de interesse; *pointcuts*, que são a declaração dos *joinpoints*; e, *advices*, que são os trechos de código que implementam o interesse transversal. Além dessas estruturas relativas à POA, um aspecto possui também todas as declarações cabíveis a uma classe da orientação a objetos, como métodos e atributos.

O *AspectJ* foi apresentado por Kiczales et al. (2001) como uma extensão para a linguagem JAVA, preocupando-se em manter a compatibilidade e a usabilidade para que o desenvolvedor JAVA tenha facilidade em compreender a utilização do *AspectJ* e possa incorporá-lo às suas aplicações. Uma aplicação *AspectJ* precisa ser completamente compatível com as máquinas virtuais JAVA, assim como uma aplicação pura de Java pode ser uma aplicação legal *AspectJ*.

O código *AspectJ* é baseado na definição de aspectos, estruturas semelhantes às classes da linguagem JAVA de forma que as três principais estruturas que atribuirão a uma aplicação pura JAVA o caráter de orientação a aspectos, são:

- *Join points*: pontos claramente definidos da execução de uma aplicação;

- *Pointcuts*: estruturas para referenciar grupos de *join points* e valores desses pontos da execução;

- *Advices*: blocos de código semelhantes aos métodos que explicitam o comportamento que será adicionado nos *join points*.

Um *join point* pode ser considerado um nodo no grafo de execução da aplicação, como por exemplo a chamada de um método, a chamada de um construtor, a inicialização de uma

classe, execução de um método, entre outros. Esses pontos podem ser definidos através de uma representação textual: os *pointcuts*.

O *AspectJ* possui diversos designadores de *pointcuts*, como *execution* (para a execução de métodos e construtores), *call* (para a chamada de métodos e construtores), *within* (para determinar se o *pointcut* está dentro do corpo de uma determinada classe), *cflow* (para definir o fluxo de execução desse *pointcut*), entre outros que possibilitam referenciar diversos pontos da execução. O código a seguir exemplifica a definição de um *pointcut* que referencia o seguinte *joinpoint* "todas as chamadas dos métodos começados com *set* que estão no pacote *edu.performance.test*".

```
@Pointcut("call(* edu.performance.test.set*(..))")
    public void Sets() {


    }
```

Com relação aos *Advices* são suportados três tipos (*before*, *after*, *around)* que permitem a inserção de código antes, depois ou em substituição do *join point*, respectivamente. Os *advices* do tipo *before* e *after* são costurados ao código Java em tempo de compilação. Desse modo, o *overhead* incluído é de uma chamada de métodos, mas sempre estáticos ou constantes, permitindo que a JVM a interprete como métodos *inline*. Com essa medida, o *AspectJ* busca reduzir significativamente o *overhead* de sua utilização. Os *advices around* possuem um *overhead* um pouco mais significativo, pois não geram métodos que podem ser transformados em métodos *inline* pela JVM.

A utilização da programação orientada a aspectos desperta nos desenvolvedores a dúvida sobre possível sobrecarga nas aplicações pela separação de interesses e *costura* dos aspectos nos pontos de interesse da aplicação. Essa incerteza tem motivado alguns trabalhos que visam investigar esse impacto no desempenho da aplicação mediante o uso de POA.

Em Chawla; Orso (2004) é proposto um *framework* para coleta de informações dinâmicas de aplicações Java. A ideia é focada em permitir a instrumentação de código para obter informações do programa em execução e reduzir o *overhead* sobre essa instrumentação quando comparado ao uso de *AspectJ*. Para essa finalidade, os autores reduzem as informações coletadas apenas àquelas solicitadas pelo usuário diferentemente do *AspectJ* que busca um conjunto grande de informações pré-definidas do ponto do código observado. Nesse trabalho, os estudiosos realizaram um estudo de caso onde o *overhead* imposto pelo *framework* é de 2 a 3 vezes menor do que usando *AspectJ*. No entanto, a pesquisa apresentada é limitada e não se pode garantir que seja generalizável. Outro ponto relevante é a estrutura do *AspectJ* que possibilita a inserção de qualquer tipo de métrica nos pontos de interesse, uma vez que exista alguma biblioteca que forneça tal métrica. Já o *framework* proposto não deixa claro como poderia ser feita essa integração com outras bibliotecas que podem oferecer informações adicionais em relação ao código avaliado.

Em Alexandersson; Öhman (2010) os autores buscam avaliar a sobrecarga incluída em um sistema de frenagem de veículo ao se implementar tolerância a falhas através da programação orientada a aspectos. O estudo de caso utilizou a linguagem *AspectC++* e os experimentos iniciais apontaram uma sobrecarga que inviabilizaria a utilização da tolerância implementada. Ao se investigar o *weaver* da linguagem, foi possível promover otimizações que reduziram consideravelmente o impacto imposto pela POA. Os resultados finais obtidos mostram que a

não utilização de opções de otimização no compilador GCC reduziu o desempenho da aplicação mais do que a utilização de aspectos que fornecem uma melhor modularidade a aplicação.

Em Dufour et al. (2004) é feito um estudo do comportamento de aplicações desenvolvidas com orientação a aspectos. O trabalho propõe modificações no interpretador da linguagem *AspectJ* para que sejam fornecidos detalhes sobre o impacto do código orientado a aspectos na aplicação base. As alterações permitem, além de mensurar o tempo gasto em aplicar os aspectos ao programa base, quantificar e classificar as instruções de *bytecode* geradas pelos aspectos. Também foi apresentado um estudo de caso com oito aplicações JAVA que têm diferentes usos das ferramentas fornecidas pelo *AspectJ*. Dentre essas, três aplicações o argumento de que o uso de aspectos fornece uma modularidade não disponível com a orientação objeto da linguagem Java sem afetar significativamente seu desempenho. No entanto, nas outras aplicações foi percebida uma grande carga adicionada pelas instruções relacionadas à orientação a aspectos. Nesses casos, foram propostas algumas alterações tanto na utilização da linguagem como modificações para otimizar a geração de código pelo interpretador. Foi possível, nesse trabalho, apontar ao utilizador da linguagem que o uso indiscriminado do *around*, uma definição muito genérica dos *pointcuts*, e a utilização do *cflow* podem afetar significativamente o desempenho da aplicação. Os autores reiteram que se deve ajustar a definição dos *pointcuts* para que esses representem de forma precisa o local que necessita de adição dos *advices*. É sugerido, ainda, utilizar *after returning*, sempre que possível, pois afeta menos o desempenho, ao invés do *around*.

Em Debusmann; Geihs (2003) é proposta a utilização de uma abordagem com programação orientada a aspectos para validar o cumprimento de SLA (do inglês, *Service Level Agreement*) em soluções baseadas em componentes. O trabalho avaliou o impacto da utilização de aspectos para realizar o monitoramento de requisitos não funcionais e comparou com o alcance do monitoramento realizado por código manualmente inserido no componente. O estudo de caso foi realizado com um servidor de páginas que recebe 1000 requisições de um determinado cliente, espaçadas em 100 ms. O acompanhamento consiste em determinar o tempo necessário para resposta à requisição. Os resultados mostraram que o tempo de resposta ao cliente, no pior caso, é 7% maior com a instrumentação manual e 8% maior com a utilização de aspectos. Essa pequena diferença apoia a utilização de uma abordagem orientada a aspectos, pois possibilita a total independência entre o código monitoramento e código do serviço a ser executado, a reutilização do código de monitoramento e uma fácil manutenção e aprimoramento no código de monitoramento sem aumentar significativamente o custo de observar o cumprimento dos SLAs.

Em Okanović; Vidaković; et al. (2013) temos uma opção ao uso de AspectJ para monitoramento adaptativo de aplicações empresariais distribuídas. Em trabalhos anteriores, os autores desenvolveram o Dprof, Okanović; Van Hoorn; et al. (2013) para esse monitoramento adaptativo utilizando AspectJ. No entanto, estudiosos justificam que o AspectJ introduziu uma sobrecarga grande em algumas situações (não esclarecidas) e só trabalha em nível dos métodos. Assim, eles desenvolveram um novo *framework*, DiSL, para substituir o uso do AspectJ no Dprof e, assim, minimizar o impacto introduzido e ter maior poder de inserção no código da aplicação. O DiSL é bastante semelhante ao AspectJ, mas não possui a opção de *advice* do tipo *around*, porém para substituí-la é utilizada a combinação de *advices before* e *after*. Outra característica é a possibilidade de realizar instrumentações dentro de métodos, diferentemente do AspectJ. O estudo de caso comparativo mediu o tempo adicional do monitoramento com Dprof e AspectJ, além daquele feito com o Dprof e DISL. Nessa comparação, foi percebida uma redução de apenas 1,2 % na carga adicional introduzida. A

pesquisa, no entanto, foi realizado com *around* do AspectJ, mostrado em Dufour et al. (2004) como mais dispendioso do que *before/after* do próprio AspectJ.

Em Hassan (2014) foi realizado um estudo para avaliar o impacto do uso de programação orientada a aspectos em aplicações *Android*. Foram retirados trechos de código da aplicação referentes ao tratamento de exceções e de *logging* passando-os para um Aspecto. Os autores avaliaram métricas de qualidade de *software*. A métrica LOC (do inglês, *Lines of Code*) foi reduzida em 8% devido à diminuição de códigos repetidos para exceção e log ao se usar POA. Também, perceberam um aumento no acoplamento da aplicação devido à criação de interdependências entre o aspecto e as classes da aplicação. No entanto, não tratam da diferença de desempenho entre as versões de código.

Em Liu et al. (2011) os estudiosos promoveram um estudo de caso para avaliar o impacto da programação orientada a aspectos em aplicações concorrentes. A pesquisa parte de duas hipóteses: 1) o uso de POA aumenta a sobrecarga do sistema; e, 2) o aumento no número de *joinpoints* piora o desempenho da aplicação orientada a aspectos. No estudo realizado eles analisaram as aplicações *Leader/Followers*, *Producer/Consumer* e *Half-Sync/Half-Async*, que tiveram os seguintes parâmetros variados: quantidade de *threads*, tamanho da fila, número de *joinpoints* e características das aplicações (com alta exigência de CPU ou exigentes de uma combinação entre CPU e armazenamento). As métricas avaliadas foram o tempo médio de execução e o impacto da utilização POA na aplicação. Os resultados apontaram que a programação orientada a aspectos não afetou negativamente o desempenho das aplicações e que o aumento no número de *joinpoints* não é um fator que afetou significantemente a aplicação.

A partir dos trabalhos já desenvolvidos para avaliar o custo e os benefícios de POA nas aplicações, o *framework* utiliza *advices* do tipo *before* e *after* que permitem fornecer a modularidade à aplicação sem afetar significativamente o seu desempenho. A seção a seguir apresenta as métricas que são coletadas no *framework* e como essa coleta é realizada através de POA.

4.4 Métricas

Para avaliar as aplicações de teste geradas são necessárias métricas, requisitos não funcionais da aplicação, como o tempo para realização das tarefas ou a quantidade de recurso que uma determinada operação consome durante a sua execução.

A coleta de métricas é um processo transversal a todas as operações que podem compor a biblioteca de testes, ou seja, todas elas terão métricas coletadas para posterior avaliação. Seguindo o paradigma de orientação a objetos, poderíamos implementar a coleta de métricas através de herança ou poderíamos criar uma classe estática, responsável apenas pela coleta a qual seria chamada a cada vez que fosse necessário coletar uma métrica. No entanto, ao utilizarmos essas abordagens estaríamos criando um relacionamento desnecessário entre o código da operação e o código de coletas. Escolhendo, por exemplo, a herança, poderíamos ter um método obrigatoriamente chamado antes e depois de cada execução de operação e assim coletar as métricas. Uma desvantagem dessa abordagem é a criação de vínculo entre a coleta de métricas e o código que efetivamente realiza a operação. Outra possível abordagem seria criar uma classe ou conjunto de classes apenas para coletar métricas e acioná-la essa classe quando fosse desejado que as métricas fossem coletadas. Essas duas abordagens

poderiam ser implementadas nesse *framework*. No entanto, elas possuem algumas desvantagens como o aumento do acoplamento entre as classes da aplicação, que pode levar a uma maior dificuldade na evolução e manutenção do *framework*.

Mediante a natureza transversal da coleta de métricas, a utilização do paradigma de orientação a aspectos apresentou-se viável, pois seria possível separar completamente o código referente às operações realizadas no *framework* do código responsável por coletar as métricas. Essa divisão possibilita, por exemplo, que uma nova métrica seja inserida no *framework* sem qualquer necessidade de alteração no código das operações já existentes.

Na seção 4.3 foi apresentada uma visão geral da programação orientada a aspectos. No desenvolvimento do *framework* foi utilizada a linguagem *AspectJ,* Kiczales et al. (2001), desenvolvida com base em Java, suportando assim todos os seus mecanismos e acrescentando a orientação a aspectos em uma formatação similar a orientação a objetos facilitando sua utilização.

Ao utilizar essa linguagem, foram definidos os *Pointcuts*, descrições de pontos da aplicação que poderão sofrer alguma intervenção do através de um aspecto. Vejamos por exemplo o Pointcut definido a seguir.

```
@Pointcut("call(* edu.performance.test.*.*.testTJM*(..))")
    public void TestMethodsTimeAndMemoryJava() {


    }
```

Esse Pointcut, denominado *TestMethodsTimeAndMemoryJava*, define como ponto de execução da aplicação, todas as chamadas de qualquer método que tenha seu nome iniciado por *testTJM* e que estão no segundo nível de sub pacotes do pacote *edu.performance.test*. Através desse *Pointcut,* muitas intervenções podem ser realizadas nesse ponto de execução, inclusive a total substituição do código dos métodos.

Com os pontos de interesse definidos, podemos estabelecer os *jointpoints*, que são descritos através *pointcuts*, e determinar se as intervenções serão de caráter incremental ao código ou substitutivo. No primeiro caso, usaremos *jointpoints* associados a *Advices* do tipo *before* e *after*, que possibilitam incrementos de código antes ou depois do ponto de interesse, respectivamente. No segundo caso é utilizado *advice* do tipo *around.* A seguir é apresentado um esboço de como é inserida a coleta de métricas no *Pointcut TestMethodsTimeAndMemoryJava*.

Figura 4.5 – Esboço do arquivo de aspecto para coleta de métricas.

*Metrics.aj

```
1  package edu.performance.test.util;
2
3  import org.aspectj.lang.JoinPoint;
4  import org.aspectj.lang.annotation.After;
5  import org.aspectj.lang.annotation.Before;
6
7  public aspect Metrics {
8
9      @Before("TestMethodsTimeAndMemoryJava()")
10             public void logBeforeTM(JoinPoint joinPoint) {
11                 //Realiza preparação para a coletadas métricas:
12                 //inicia contadores e instancia objetos.
13             }
14
15             @After("(TestMethodsTimeAndMemoryJava()")
16             public void logAfterTM(JoinPoint joinPoint) {
17         //Coleta a métricas, limpa os contadores e armazena os dados.
18         }
19
20
21 }
```

Fonte: Próprio autor.

O *Jointpoint*, denominado *logBeforeTM*, permite a definição de algum código a ser inserido antes da ação delimitada pelo *Pointcut*, indicado entre as aspas duplas. Esse código inserido é chamado de *Advice*. No caso do *framework* proposto, é realizada a instanciação de objetos que fornecem as métricas e inicialização dos contadores de outras métricas. Após a execução desse *Advice*, o método monitorado é executado. Após a execução do método, o *logAfterTM* é acionado e as métricas definidas no *Advice* são coletadas e guardadas.

Como mencionado anteriormente, o código da operação é independente do de coleta das métricas. Por uma questão de organização foi definido um padrão para o início do nome dos métodos a serem monitorados. Assim, pode-se criar um *pointcut* que represente o maior número de pontos de interesse possível. No entanto, em algumas operações, principalmente as relacionadas a operações de GPU, foram necessários outros *pointcuts* devido às peculiaridades dessas operações.

Desse modo, caso seja necessário coletar uma nova métrica, basta alterar o código *advice* e, assim, haverá uma nova análise sem necessidade de mudar qualquer código na classe que implementa a operação. Isto acontece porque o código do *advice* será costurado ao da aplicação no momento de construção da mesma.

O apêndice contém o código completo e comentado do aspecto utilizado para coletar as métricas nessa primeira implementação do *framework.*

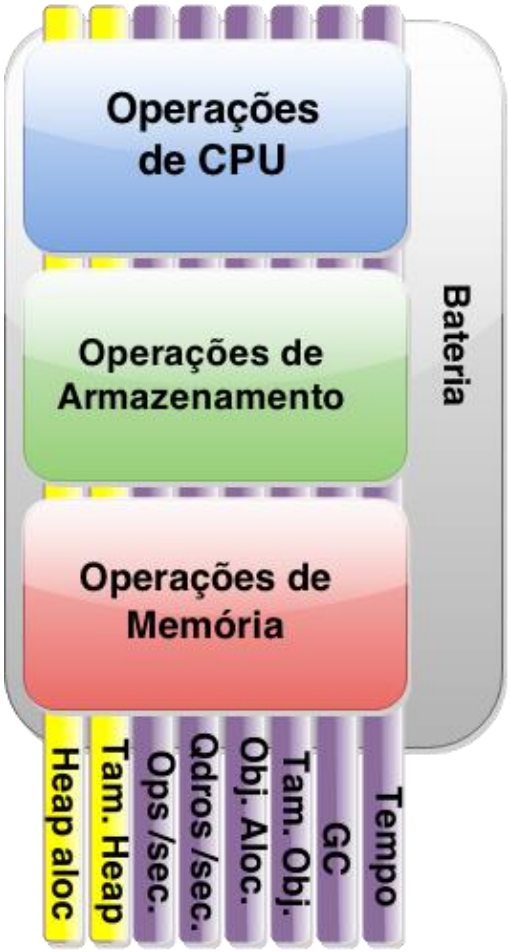
4.4.1 Métricas coletadas

Nessa primeira implementação do *framework,* foram selecionadas nove métricas, sendo cinco delas fornecidas pela classe *Debug*[24] (API do Android), uma através do monitoramento do estado da bateria e as demais relacionadas ao tempo e a quantidade de operações realizadas num intervalo de tempo. A Figura 4.6 mostra uma representação de como é a relação das

[24] http://developer.android.com/reference/android/os/Debug.html

métricas e as operações da biblioteca. Isto é, a coleta das métricas é transversal a todas as operações. Exceto pela métrica de bateria, as métricas são coletadas via POA.

Figura 4.6 – Exemplo de estruturação de uma aplicação gerada pelo *framework*.



Fonte: Próprio autor

A seguir são apresentadas e detalhadas as métricas que atualmente são coletadas no *framework*.

1) **Tempo de execução**: é baseada no tempo total gasto para realizar uma dada operação disponível na biblioteca de testes. A coleta dessa métrica é feita através de um intervalo de chamadas do método *SystemClock.uptimeMillis()*, com precisão de milissegundos;

2) **Acionamentos do coletor de lixo**: essa métrica é coletada através da classe *Debug*[25] do SDK do Android. No *Advice*, executado antes da operação monitorada, é realizada a chamada dos métodos *Debug.resetAllCounts*() e *Debug.startAllocCounting()*, sendo que o primeiro limpa todos os contadores fornecidos por essa classe e o segundo inicia o monitoramento realizado pela classe *Debug*. Então, no *Advice* executado após a execução, é finalizado o monitoramento da classe *Debug* através do método *stopAllocCounting()*. Neste momento, são chamados dois métodos, o *Debug.getGlobalGcInvocationCount()*, que fornece a quantidade de vezes que o coletor foi acionado nas *threads* do sistema operacional ao longo do período

[25] http://developer.android.com/reference/android/os/Debug.html

de monitoramento, e o *Debug.getThreadGcInvocationCount()*, que indica o número de chamadas do coletor feitas apenas pela *thread* local;

3) **Tamanho de objetos**: essa métrica, tal assim como a de coletor de lixo, é fornecida pela classe Debug e tem seus dados coletados durante o monitoramento realizado por essa classe. Os valores dessa métrica são obtidos através do método *Debug.getGlobalAllocSize()*, que fornece o tamanho total em *bytes* dos objetos alocados durante o monitoramento, e do método *Debug.getThreadAllocSize()*, que informa, também em *bytes*, o tamanho dos objetos alocados pelo *thread* local;

4) **Objetos alocados**: essa métrica é fornecida pela classe *Debug* e apresenta a quantidade de objetos alocados, de forma global pelo método *Debug.getGlobalAllocCount()* e relacionados à *thread* local pelo método *Debug.getThreadAllocCount();*

5) **Consumo de bateria**: essa métrica é coletada através da utilização de *BroadcastReceivers* da linguagem *Android*. Esse serviço possibilita que uma determinada *Activity Android* seja informada de mudanças ocorridas na bateria do dispositivo e, assim, coletar informações da bateria quando essa mudança ocorre. Ainda, são possíveis duas análises: quantas vezes uma dada aplicação de teste foi executada até que a bateria varie por uma porcentagem pré-definida e a possibilidade que se determine um intervalo de tempo para que a aplicação execute em *loop* infinito e, ao final do intervalo de tempo, é coletada variação da bateria.

6) **Quadros por segundo**: essa métrica está disponível apenas para as operações relativas ao uso da GPU. Assim, quando uma operação dessa categoria é selecionada para aplicação de teste, será contabilizada a quantidade de quadros desenhados em intervalos de 5 segundos;

7) **Operações por segundo**: essa métrica está disponível e, atualmente, relacionada apenas à operação de banco de dados. É derivada da métrica de tempo total de execução, ou seja, quando o usuário passa uma quantidade de operações de banco para serem realizadas é calculado o tempo total gasto e feito a média com número de operações realizadas.

8) **Tamanho do *heap***: essa métrica é fornecida pela classe *Debug* e está disponível apenas para operações com código nativo, pois apresenta o tamanho do *heap* nativo. No *advice* adicionado antes dos métodos nativos é feita a chamada do método *startNativeTracing()* e no *advice* do tipo *after* das operações com código nativo é chamado método *stopNativeTracing()*. Nesse último *advice* é chamado o método *getNativeHeapSize()* que retorna o tamanho atual do *heap* nativo.

9) ***Heap* alocado**: essa métrica é bastante semelhante ao tamanho do *heap*. Sua coleta é delimitada pelas mesmas chamadas de método e está disponível apenas para código nativo. A diferença principal é que essa métrica é definida como o montante de memória que está alocada no *heap*. O método que retorna esse dado é *getNativeHeapAllocatedSize()*.

4.5 Construção da aplicação de teste

A versão atual do *framework* foi implementada na linguagem JAVA para *Android*, de forma que essa plataforma se baseia na utilização de *Activities* e *Intents*. Buscando controlar melhor a execução de listas de testes, cada um deles faz parte de uma *Activity* e é executado

em uma *thread* específica. Para coleta da métrica de consumo de bateria é utilizada uma *Activity* separada (*BatteryMetric*) que solicita, do sistema operacional, notificações sobre o estado da bateria. Assim, quando o usuário deseja essa métrica, a lista de testes é executada a partir dessa *Activity*.

A construção atual do *framework* baseia-se na utilização de arquivos .xml que contêm os testes que serão realizados. Com essa estrutura, o *framework* está preparado para receber uma interface gráfica que permita ao usuário selecionar os testes que espera fazer. A classe *TestManager* é responsável por ler o arquivo .xml de testes e gerar uma lista desses formada por *Intents*, sendo que cada *Intent* corresponde a um teste. A Figura 4.7 mostra um exemplo de arquivo .xml com dois testes, um da operação banco de dados e outro da operação de *download*.

Figura 4.7 – Exemplo do arquivo xml de testes.

MetricsByAspects.aj | listadetestes_alguns.xml

```
<?xml version="1.0" encoding="utf-8"?>
<testList
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="testListxsd.xsd">
    <test>
        <activityName>edu.performance.test.database.DatabaseOperationActivity</activityName>
        <extraInt name="MAXTIMEMS">400000</extraInt>
        <extraBool name="THELASTTEST">false</extraBool>
        <extraBool name="BATTERYTEST">false</extraBool>
        <extraInt name="LEVEL_INT">50</extraInt>
        <extraStr name="STATUS">Testing Database skills..</extraStr>
        <extraBool name="NETWORKTEST">false</extraBool>
    </test>
    <test>
        <activityName>edu.performance.test.downloadoperation.DownloadOperationActivity</activityName>
        <extraInt name="MAXTIMEMS">400000</extraInt>
        <extraBool name="THELASTTEST"> false</extraBool>
        <extraBool name="BATTERYTEST"> false</extraBool>
        <extraInt name="LEVEL_INT">0</extraInt>
        <extraStr name="FILELOCATION">/data/data/edu.performance.test/app_performanceDir</extraStr>
        <extraStr name="STATUS">Testing Download skills..</extraStr>
        <extraBool name="NETWORKTEST">true</extraBool>
    </test>

</testList>
```

Fonte: Próprio autor.

O *framework* possui uma *Activity* central denominada *Library*, classe responsável pelas pré-configurações do teste. Inicialmente, faz-se o carregamento da biblioteca de testes desenvolvidos em C/C++ e, posteriormente, são escritos no sistema de armazenamento todos os arquivos de texto que serão usados no teste. Essa classe, também, é responsável por solicitar o controle ao sistema operacional, de modo que o celular não entrará em modo *sleep* durante os testes.

Com base nas escolhas do usuário, pode ser acionada a classe estática *TestManager* ou lançar a *Activity* denominada *BatteryMetric*. Quando o usuário não deseja obter métricas de consumo de bateria, a classe *TestManager* gera a lista de testes e o método *executeTest(Intent i)* é acionado na *Library*. Esse método verifica se o teste necessita de acesso à rede de *internet* e, se confirmado, o sensor é acionado, caso contrário esse é desligado. No cenário do dispositivo não contar com sensor de *WiFi*, será solicitado que o usuário ligue manualmente a rede de dados da operadora ou ainda cancele a execução da aplicação de teste.

Como cada teste está englobado por um *Intent*, o controle de execução é feito pelo método sobrescrito *onActivityResult*. Nesse método, caso o teste não tenha ocorrido normalmente, uma mensagem de erro é adicionada ao arquivo de texto correspondente e, em seguida, é

verificado se o teste finalizado é o último da lista de testes. Em caso afirmativo, os arquivos copiados são excluídos dos dispositivos e é realizado o fechamento da aplicação. Caso não seja o último teste, é acionado novamente o método *executeTest* com o próximo a ser realizado.

Duas classes compõem um teste a ser realizado: uma responsável pelo controle de execução do teste (estende a classe abstrata *PerformanceTestActivity)* e a outra comporta os trechos de código (unidades de teste) executados pelo teste (estendem a classe abstrata e parametrizável *PerformanceTest)*.

Como detalhado anteriormente, cada teste é encapsulado dentro de uma *Intent*, contendo a classe (*Activity*) da operação que será realizada, além dos parâmetros que configuram o teste. A *PerformanceTestActivity* possui o método abstrato *execute()* que deve ser implementado nas *Activities* das operações de teste. Devido a características da linguagem *Android*, nos testes relacionados à interface gráfica, como os da categoria GPU, a implementação desse método é vazia. Nos demais, esse método contém a chamada do método *execute0* da classe abstrata. Outra funcionalidade, é validar que os parâmetros mínimos obrigatórios de cada teste tenham sido fornecidos. São eles:

- THELASTTEST: tipo booleano que determina se o teste é o último da lista;

- MAXTIME: tipo inteiro que determina o tempo máximo (em ms) que um teste pode durar. Caso esse tempo seja ultrapassado, o teste é finalizado com erro de *Timeout*;

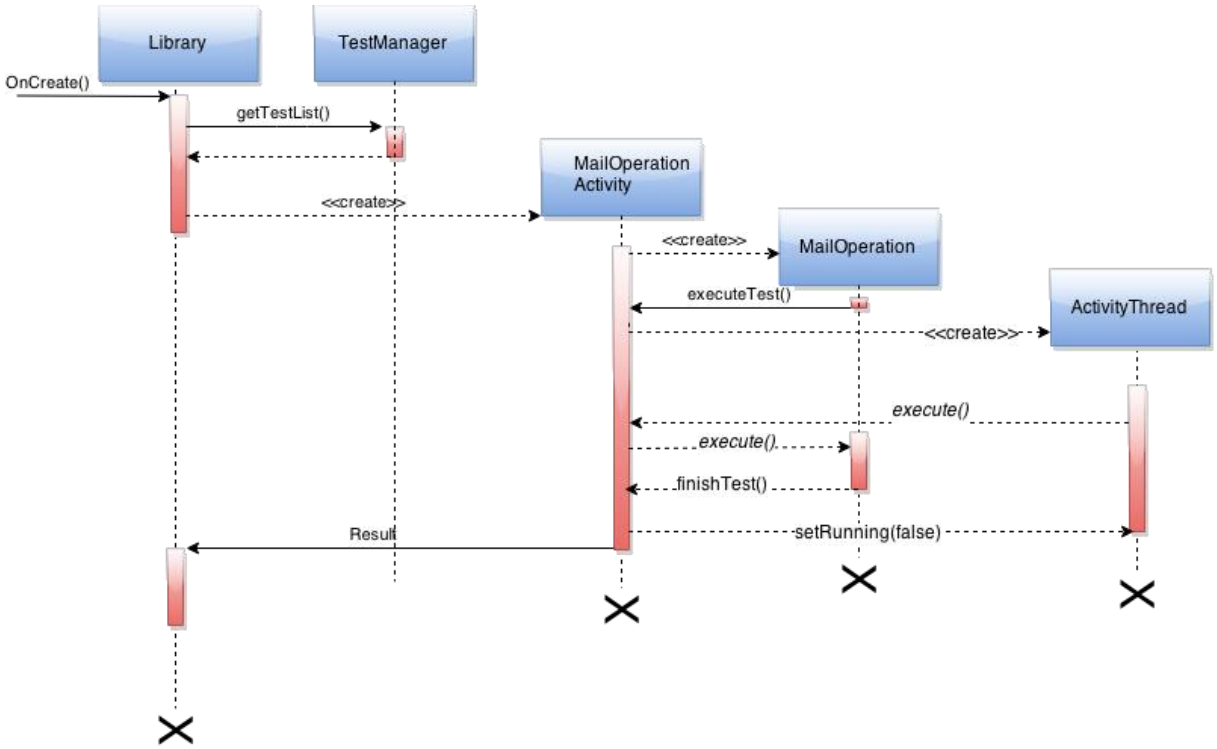
- STATUS: tipo *string* que armazena a mensagem a ser mostrada na tela enquanto o teste está sendo realizado;

- BATTERYTEST: tipo booleano que pode sobrepor o parâmetro MAXTIME e possibilitar que um teste dure tempo indeterminado. Por padrão, esse parâmetro tem valor falso, e deve ser verdadeiro apenas quando se deseja fazer um teste que dure um tempo maior que o representado pelo maior inteiro disponível.

Cada teste é executado em uma *thread* para possibilitar a coleta de métricas únicas daquele teste. Ao ser iniciada, a *thread* chama o método *execute()* da classe *PerformanceTestActivity* que, através do polimorfismo, acionará o método correto implementado pela subclasse. A Figura 4.8 apresenta um diagrama de sequência que mostra a execução de uma operação que, nesse exemplo, foi a de envio de *e-mail*.

A classe *PerformanceTest* possui o atributo parametrizável *level*, utilizando o conceito de *Generics* da linguagem JAVA, para graduar a intensidade da carga de trabalho realizada pelo teste. Assim, é possível utilizar diversos tipos de *level*, por exemplo, a operação de banco de dados possui um *level* do tipo inteiro que determina o número de operações realizadas, já a operação de leitura de arquivos tem *level* do tipo *string* com o caminho do arquivo a ser lido. Quando é criada uma subclasse de *PerformanceTest*, é definido o tipo do nível. Essa classe também possui o método abstrato *execute()* que, quando implementado pelas subclasses, contém as unidades de teste daquela operação.

Figura 4.8 – Diagrama de sequência para operação de envio de *e-mail*.



Fonte: Próprio autor.

Quando a métrica de bateria é interessante ao usuário, a classe *Library* instancia a classe *BatteryMetric* e essa, através da *TestManager*, obtém a lista de testes que serão realizados. Entre os parâmetros de um teste com avaliação de consumo de bateria, podemos citar: BYTIME, TIME_TEST e TESTVARIATION. O primeiro determina se o teste de bateria será controlado pelo tempo, ou seja, o conjunto de testes será executado durante um dado intervalo de tempo definido em TIME_TEST. Quando o BYTIME é falso, deve ser fornecida a variação de bateria necessária para finalizar o teste, ou seja, a lista de testes será executada até que a bateria reduza seu percentual de carga ao mínimo o valor da carga inicial menos o valor definido em TESTVARIATION. Por exemplo, caso o usuário deseja que o teste seja executado até que a bateria reduza seu percentual em 5%, o parâmetro TESTVARIATION tem valor ‘5’.

O desenvolvedor de aplicações móveis tem uma ferramenta que possibilita a seleção dos testes, das cargas e das métricas a serem coletadas. Assim é possível criar um *benchmark* orientado a aplicação a ser desenvolvida.

É de grande importância para o desenvolvedor que esse *benchmark* orientado apresente um comportamento alinhado à aplicação final e que ao utilizá-lo para ordenar dispositivos reais aquele apontado como mais indicado pelo *benchmark* seja o mesmo da aplicação real. Para avaliar se as operações fornecidas atualmente no *framework* fornecem um alinhamento adequado às aplicações reais, o próximo capítulo apresenta a configuração de um ambiente experimental construído para validar o *framework*.

5 EXPERIMENTOS

5.1 Experimentos para validação

A estratégia de validação para o *framework* proposto é:

1) cinco especificações de aplicações reais e distintas foram utilizadas para identificar as operações executadas e guiar a criação semiautomática de cinco *benchmarks* orientados à aplicação;
2) os *benchmarks* foram executados em quatro dispositivos distintos;
3) os códigos-fonte de cada uma das aplicações reais foram instrumentados para coletar métricas de desempenho durante a execução;
4) as aplicações reais foram executadas nos mesmos quatro dispositivos e comparados seus reais desempenhos com o desempenho estimado pelos *benchmarks*.

Algumas aplicações *Android* de código aberto e com diferentes requisitos funcionais foram selecionadas para análise.

A necessidade das aplicações serem de código aberto é justificada, principalmente, pela necessidade de instrumentação do código para coleta das métricas de execução da aplicação real. Em um contexto real, o desenvolvedor tem em mente o tipo de estrutura de dados que pretende usar, bem como o conjunto principal de operações que serão usadas pela aplicação. Esse tipo de informação foi extraído das aplicações existentes através do seu código fonte.

Após avaliar as aplicações selecionadas, foram definidas funcionalidades básicas dessas aplicações e que poderiam ser avaliadas pelas métricas do *framework* proposto. Esse processo é oneroso, pois a estrutura utilizada para desenvolver as aplicações é distinta uma da outra e, muitas vezes, não possibilita a coleta das métricas.

Devido à grande variabilidade das aplicações selecionadas para validar o *framework*, simular sua execução não é possível de uma forma única. Com o objetivo de exercitar corretamente as funcionalidades das aplicações e possibilitar que os exercícios fossem passíveis de repetição, a simulação de cada aplicação exigiu uma abordagem distinta. Os testes foram executados em quatro dispositivos reais de especificações diferentes envolvendo os componentes de *hardware* e versões de sistema operacional. Podemos afirmar que a gama de dispositivo utilizada representa tanto dispositivos com baixo poder de processamento como dispositivos com especificações mais robustas. O detalhamento das especificações dos dispositivos é fornecido pela

Tabela 1.2.

As subseções seguintes detalham os programas de testes gerados para cada aplicação e os procedimentos utilizados para simular a execução das aplicações reais e, assim, coletar as métricas para comparação. Assim como na aplicação de testes, a coleta de métricas é realizada através de programação orientada a aspectos. Para isso, adicionamos o mesmo aspecto utilizado no *framework* e adequamos no *Pointcut* o nome dos métodos onde são coletadas as métricas.

5.1.1 Lembrar Beta[26]

Essa aplicação consiste em um cliente móvel para o sistema de gerenciamento de lista de tarefas, *Remember the Milk*[27], pois é uma típica aplicação orientada a dados que busca informações do servidor através de um *webservice*. Tais informações (tarefas e listas de tarefas) são armazenadas no banco de dados local do dispositivo e estão disponíveis para manipulação pelo usuário, mesmo sem acesso à rede de *internet*. O usuário pode, então, manipular seus dados e a aplicação fica responsável por sincronizar os dados entre o dispositivo e a nuvem. O perfil dessa aplicação, que foi utilizado para geração da aplicação de testes, consiste em uma utilização intermediária de operações com *webservice* e um uso intensivo do banco de dados local, tanto para a primeira busca de informações junto ao servidor como também para salvar as alterações realizadas pelo usuário. Nessa aplicação, as métricas que foram utilizadas para comparação entre a estimativa e a aplicação real foram tempo de execução e consumo de bateria.

O *benchmark* orientado gerado utilizou duas operações disponíveis no *framework*, as operações de banco de dados com parâmetro *level* igual a '200' e operações de *webservice* com parâmetro *level* igual a "http://mcupdate.tumblr.com/api/read?num=50". Essas operações são gerenciadas através de *Activities* do Android. Dessa forma, o ciclo consiste em executar a operação de *web service* e executar operações de banco de dados. Ao fim desse ciclo, é verificado se a bateria já decaiu o limite atribuído de 11%. Em caso positivo, o programa finaliza. Caso contrário, o ciclo é retomado e o processo se repete até que ocorra a alteração desejada na bateria.

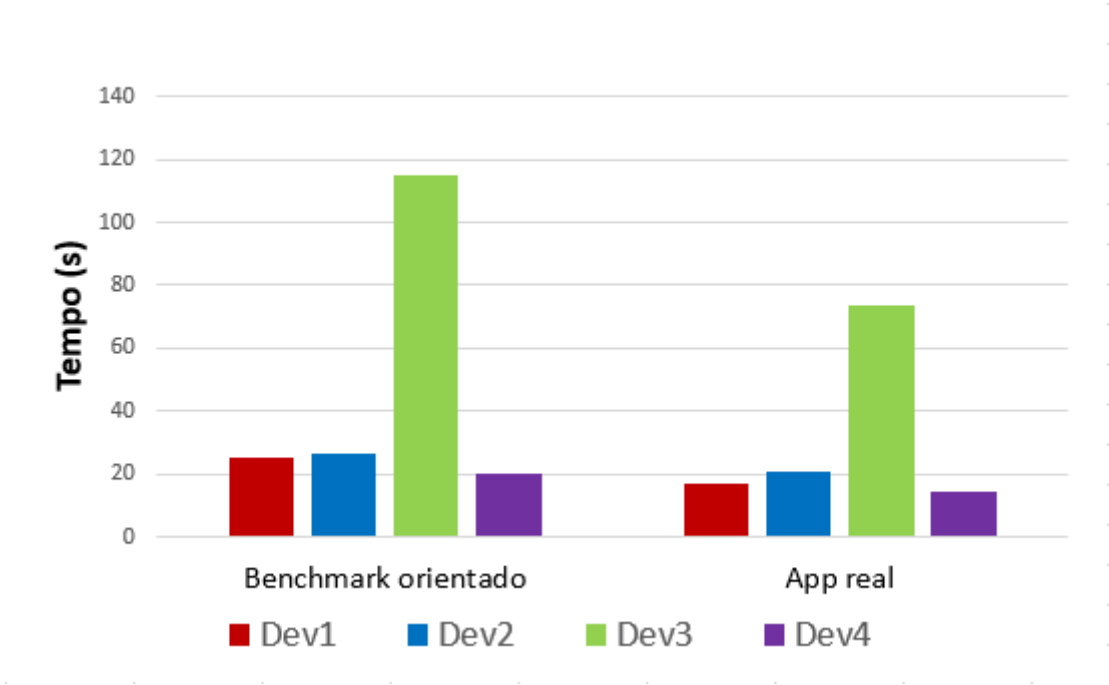
Para simular o uso da verdadeira implementação dessa aplicação, foi desenvolvida uma aplicação *Android* que aciona a aplicação real. Essa, por sua vez, realizará a requisição de dados do serviço *web* a serem inseridos no banco de dados local e, após todas as inserções, são feitas alterações em todas as tarefas. Posteriormente, todas as tarefas são excluídas e a aplicação fechada, formando um ciclo de execução. Esse processo é repetido até que a bateria reduza seu percentual em 11% e são calculadas quantas vezes o ciclo foi executado. O resultado é mostrado como uma média do número de execuções necessárias para variar 1% o valor de carga da bateria. Para a métrica de tempo de execução, o aspecto coletou os dados normalmente.

A Figura 5.1 apresenta a comparação entre os tempos de execução do *benchmark* orientado e da aplicação real nos quatro dispositivos avaliados. Nesse gráfico, quanto menor a barra mais eficiente é o dispositivo para executar a aplicação.

Figura 5.1 – Gráfico de comparação de tempo de execução do Lembrar

[26] https://play.google.com/store/apps/details?id=br.thg.lmb&hl=pt_BR.

[27] https://www.rememberthemilk.com/



Fonte: próprio autor.

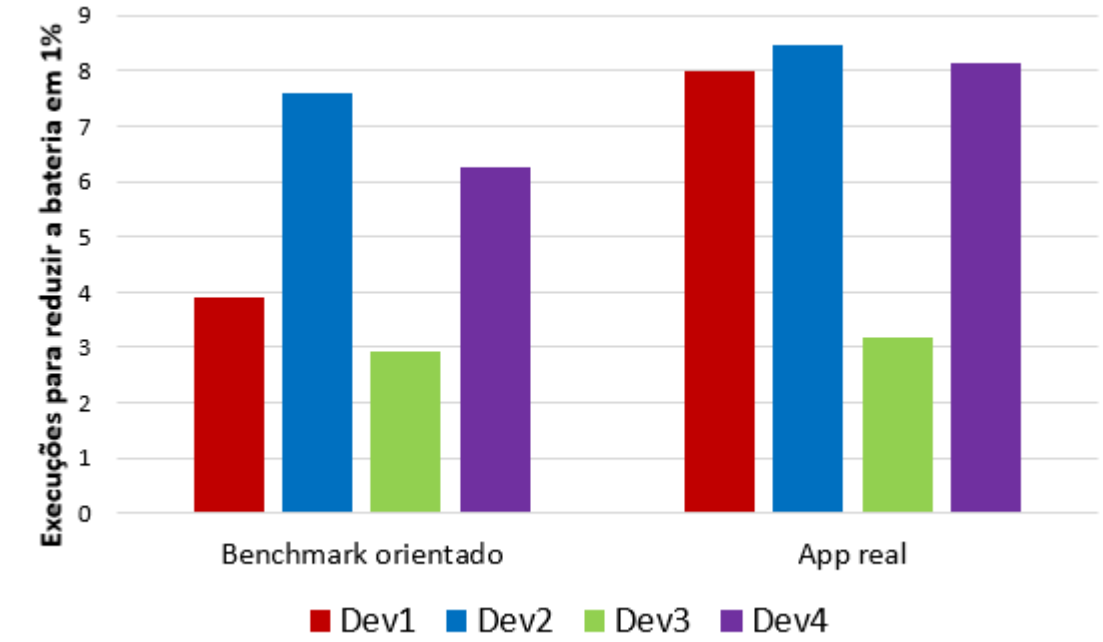
É importante salientar que o *framework* não busca ter o mesmo valor das métricas da aplicação real, pois a comparação visa observar se o *framework* possibilita uma melhor classificação dos dispositivos com a mesma tendência da aplicação real.

De acordo com o *benchmark* orientado, os dispositivos Dev1 e Dev2 têm um desempenho semelhante, mas com uma pequena vantagem do Dev1. O Dev4 tem uma performance relativamente superior, necessitando menos tempo do que todos os demais. Além disso, o Dev3 tem o pior desempenho entre os dispositivos avaliados. Quando analisamos o gráfico mais à direita da aplicação real, observamos a mesma tendência do *benchmark* orientado.

Como exposto anteriormente, o *framework* proposto disponibiliza várias métricas que podem melhor detalhar o desempenho da aplicação de teste. No caso dessa aplicação, foi possível determinar que o fator responsável pelo desempenho pior do Dev3 são as operações de banco de dados que têm um tempo de execução muito alto. Essa informação pode gerar uma ação efetiva do desenvolvedor para melhorar o desempenho da aplicação em um determinado dispositivo.

A Figura 5.2 apresenta a comparação entre o consumo de bateria do programa de testes e a aplicação real. Nesse gráfico, quanto maior a barra mais eficiente é o dispositivo em executar a aplicação e economizar energia, pois significa que mais operações podem ser realizadas para um mesmo consumo.

Figura 5.2 – Gráfico de comparação de consumo de bateria do Lembrar



Fonte: próprio autor

De acordo com o *benchmark* orientado, o Dev2 apesar de não ser o mais rápido para executar a aplicação é o mais econômico em sua execução. O Dev4 é menos eficiente, mas supera os demais. Já o Dev1 apresentou o terceiro melhor desempenho, entretanto seria pouco mais eficiente do que o Dev3. Quando analisamos o gráfico de consumo de bateria da aplicação real, a tendência é semelhante. É possível observar que, na aplicação real, houve uma grande diferença na previsão do consumo de bateria do Dev1 justificada pelo controle de operações sem *internet*, ou seja, a cada execução de uma operação que necessita de rede a Wi-Fi é ligada e, ao fim, é desligada caso a próxima operação não necessite de internet. Como na aplicação real a rede fica ligada todo o tempo, não ocorre esse custo de acionar e desligar o sensor, gerando uma considerável economia de energia.

Se compararmos esses dados com a tabela de resultados dos *benchmarks* genéricos, veremos que o Dev1, que era classificado com última opção de modo genérico, foi classificado aqui como o segundo melhor para essa aplicação. Esse resultado reforça a premissa de que uma classificação baseada nas necessidades da aplicação pode apresentar um resultado diferente e mais próximo do comportamento da aplicação real.

5.1.2 Zirco Browser[28]

Essa aplicação consiste em um navegador *web* e a sua escolha foi pela disponibilidade de seu código, mas também por sua compatibilidade com os dispositivos disponíveis para utilização nos experimentos. Como qualquer outra aplicação similar, o Zirco Browser permite a navegação através de páginas *web* de quaisquer tipos, o acesso a *web services* e *download* de arquivos. O perfil dessa aplicação orientou para a criação de um *benchmark* composto por operações de acessos a páginas *web* de todos os tipos. Para isso, foi utilizada a carga de trabalho definida no Capítulo 3 para outra implementação da mesma funcionalidade, Fernandes et al. (2014). Além disso, o programa de teste é composto de operações de

[28] https://play.google.com/store/apps/details?id=org.zirco&hl=pt_BR.

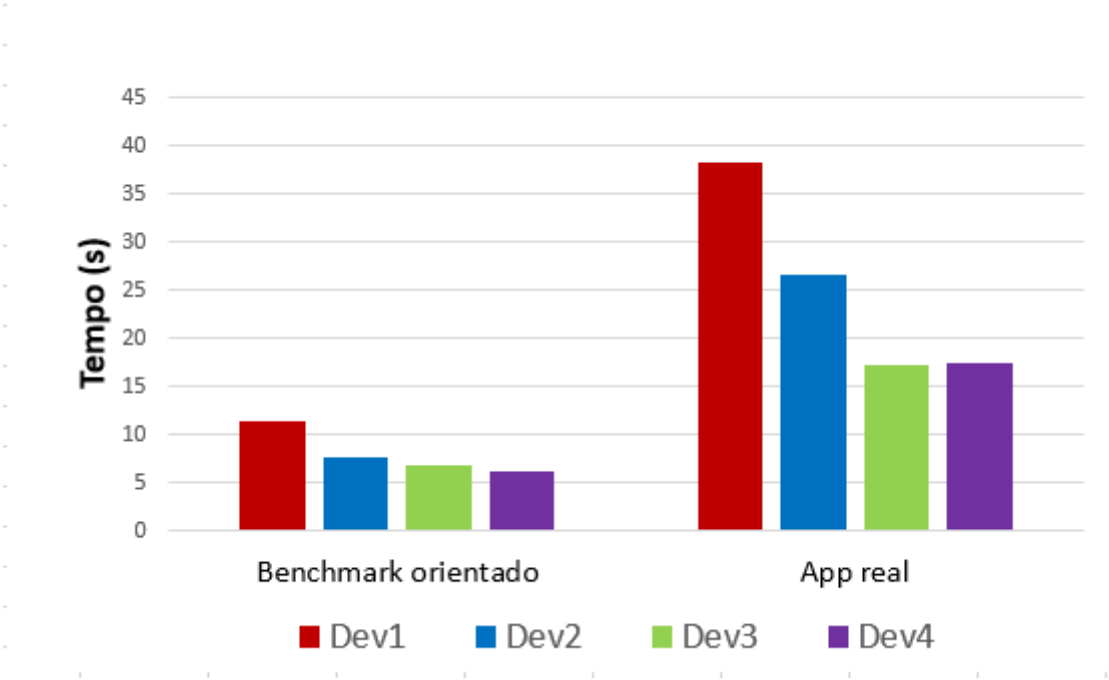
webservice e *download* de arquivos. Nessa aplicação, as métricas utilizadas para comparação foram o tempo de execução e o consumo de bateria.

O *benchmark* orientado gerado utilizou três operações da biblioteca de testes: operações de acesso a *web services* com parâmetro *level* igual a "http://mcupdate.tumblr.com/api/read", operação de acesso a páginas *web* com parâmetro *level* igual a '3' e operação de *download* de arquivos com parâmetro *level* igual a '0'. O ciclo do programa de teste consistiu em realizar a operação de acesso ao *webservice*, a operação de acesso às páginas *web* e finalizando com a operação de *download* de arquivo. Ao fim desse ciclo, é verificado se a bateria atingiu o limite de consumo. Em caso afirmativo, o programa é finalizado, mas se a situação contrária ocorre o ciclo é reiniciado e repetido até que o limite seja atingido.

Para o *Zirco Browser*, a métrica de tempo de execução foi coletada através da utilização de um *shell script* que executou comandos ADB para acionar o *browser*. Como para utilizar os comandos ADB é necessário que o dispositivo esteja conectado a uma porta USB, não é possível coletar a métrica de consumo da bateria da mesma forma. Então, um pequeno serviço *Android* foi desenvolvido para enviar chamadas da aplicação real transmitindo a ela endereços URL a serem visitados. Assim, quando a página termina de carregar ou o *download* do arquivo termina, a aplicação é fechada. Esse processo foi repetido até que a bateria reduzisse seu percentual de carga em 3% (quando usado com mais variações de percentual a aplicação parou de funcionar), sendo calculadas quantas vezes a aplicação foi executada até a variação da bateria. O resultado mostra uma média das execuções para variar em 1%.

A Figura 5.3 mostra a comparação entre o tempo de execução do *benchmark* orientado e o aplicativo real. Nesse gráfico, quanto menor a barra, mais eficiente é o dispositivo em executar a aplicação.

Figura 5.3 – Gráfico de comparação de tempo de execução do Zirco Browser



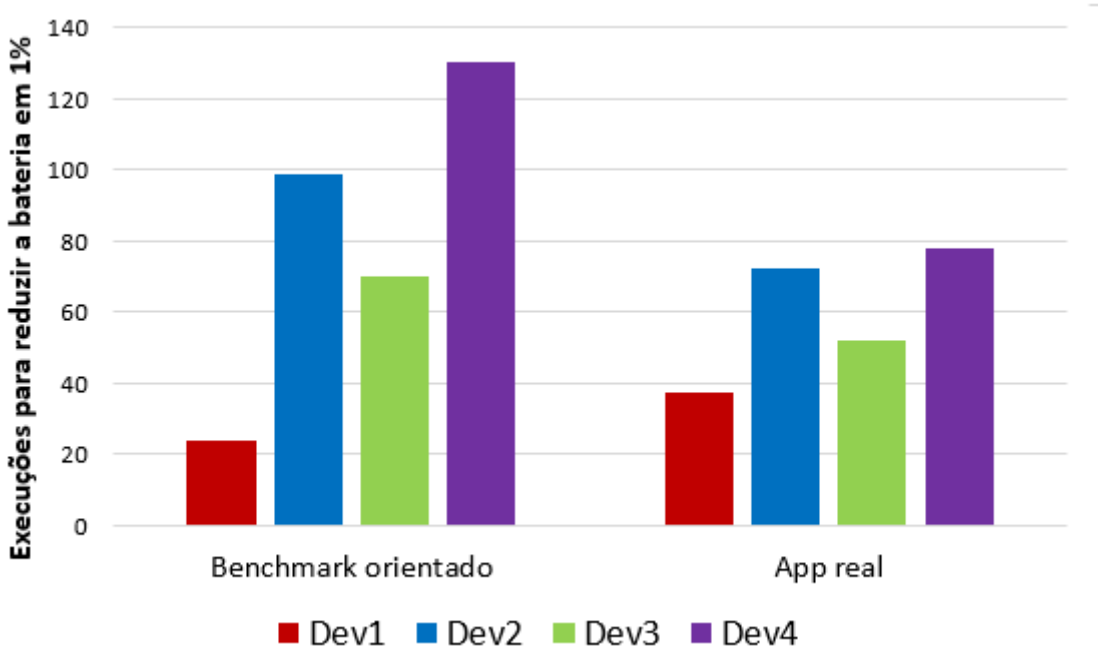
Fonte: próprio autor.

De acordo com o *benchmark* orientado, o Dev1 tem o pior desempenho para executar uma aplicação fortemente relacionada ao acesso de páginas e arquivos *web*, seguido pelo Dev2.

Ainda podemos ver uma pequena vantagem do Dev4 em relação ao Dev3. De acordo o gráfico da aplicação real, é confirmado o desempenho inferior do Dev1 seguido do Dev2. No entanto, na aplicação real o Dev3 mostrou uma pequena vantagem em relação ao Dev4, inversão que pode ser atribuída às pequenas variações inseridas pelo acesso a páginas na *internet*.

A Figura 5.4 apresenta a comparação em relação ao consumo de bateria entre o *benchmark* orientado e a aplicação real. Aqui quanto maior a barra, mais econômico é o dispositivo para executar a aplicação.

Figura 5.4 – Gráfico de comparação de consumo de bateria do Zirco



Fonte: próprio autor

De acordo com o *benchmark* orientado, o Dev1 é menos eficiente em realizar operações com acesso à rede de *internet*, seguido pelo Dev3 e Dev2, respectivamente. O Dev4 mostrou ser o mais eficiente nessas operações. Quando avaliamos o gráfico da aplicação real, a mesma tendência foi observada, apesar de uma menor distância de eficiência entre o Dev4 e o Dev2.

5.1.3 *Frozen Bubble*[29]

Essa aplicação consiste em um jogo cujo objetivo é combinar bolinhas coloridas em um grupo mínimo de três para gerar pontos. Esse jogo desenha imagens de bolinhas através do método *drawImage* da biblioteca *android.graphics.Canvas*. Dessa forma, o *benchmark* orientado que foi gerado é composto da unidade de teste de carregamento de imagem 2D e utilizamos uma das imagens desenhada no jogo real como fonte da imagem. Nessa aplicação, as métricas avaliadas foram a taxa de quadros desenhados por segundo e o consumo de bateria, através da execução repetida do jogo.

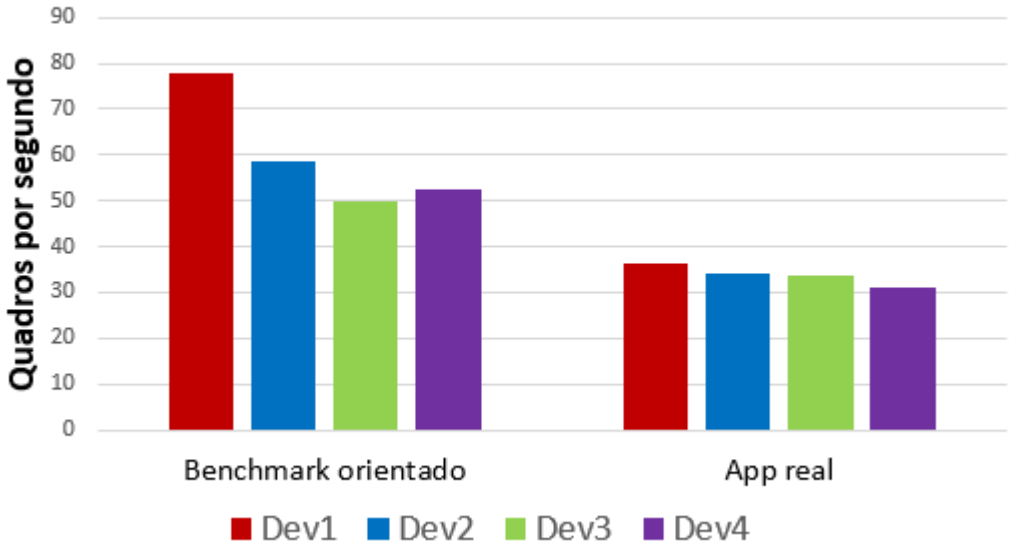
[29] https://play.google.com/store/apps/details?id=org.jfedor.frozenbubble&hl=pt_BR

O *benchmark* orientado gerado para comparar com o jogo 2D contém apenas uma unidade de teste da biblioteca, o carregamento de imagens 2D e sua movimentação na tela do dispositivo com o parâmetro *level* igual ‘10’, que corresponde a ter dez bolinhas por linha desenhadas na tela, valor semelhante a quantidade desenhada no jogo. A imagem carregada foi uma das utilizadas pelo jogo original.

Para simular a utilização da aplicação de jogo 2D, foi usada a ferramenta *Monkey*[30] do *Android* que possibilita o envio de eventos pseudoaleatórios para o dispositivo os quais continuam sendo executados mesmo com o dispositivo desconectado do computador. Assim, foram gerados 6.000 eventos de toque na tela para execução do jogo. O intervalo entre os eventos é de 500 ms o que corresponde a aproximadamente 25 minutos de interação com a aplicação possibilitando que fosse observado um consumo mínimo próximo aos 10% da bateria do dispositivo avaliado. O jogo foi alterado de forma que após uma execução completa e finalização do jogo um novo toque na tela seja suficiente para recomeçar a jogar, permitindo assim o ciclo contínuo de utilização da aplicação. Foram coletadas as métricas de taxa de quadros por segundo e também de consumo da bateria nesse período de execução dos jogos.

A Figura 5.5 apresenta a comparação da métrica de quadros por segundo entre a aplicação de teste e o jogo real. Aqui quanto maior a barra maior a taxa de quadros e melhor o desempenho do dispositivo.

Figura 5.5 – Gráfico de comparação de quadros por segundo do *Frozen Bubble*



Fonte: Próprio autor

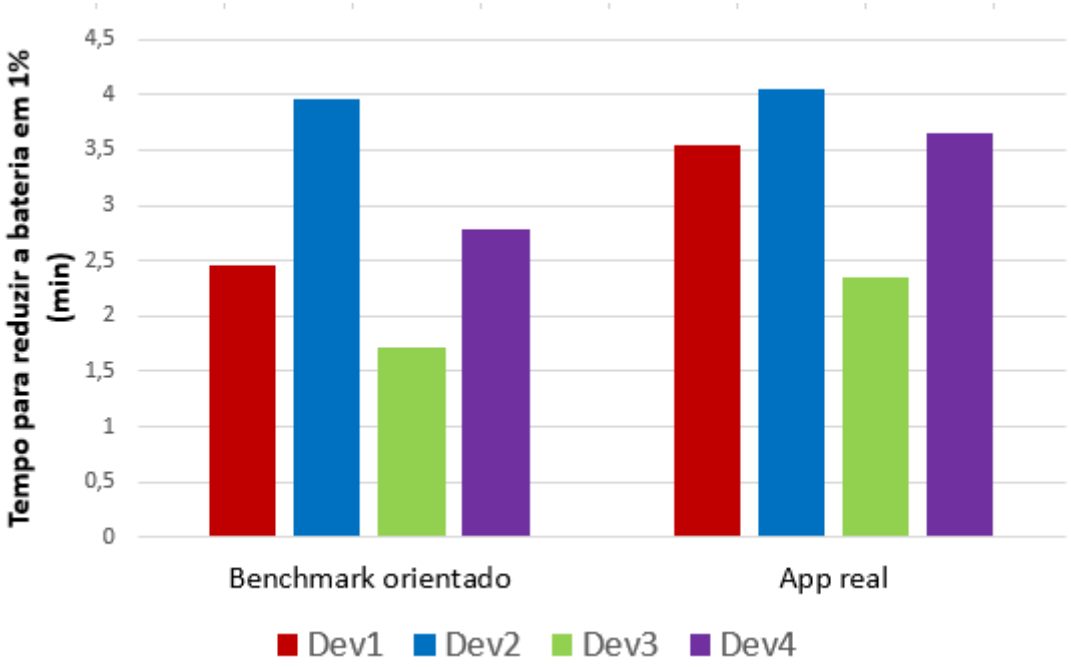
De acordo com o *benchmark* orientado, o Dev1 tem o melhor desempenho para executar a operação de carregar e renderizar a figura de uma bola, seguido pelo Dev2, Dev4 e Dev3, respectivamente. Ao avaliar a aplicação real, essa tendência se mantém, exceto em relação ao Dev3 e Dev4, onde um comportamento anômalo foi observado. Nesse experimento é importante ressaltar que o Dev1 é o dispositivo com configuração de *hardware* mais antiga e

[30] http://developer.android.com/tools/help/monkey.html

mais simples. No entanto, apresentou desempenho melhor do que os demais com melhor *hardware*. Ainda, esse comportamento se manifestou na execução do jogo real.

A Figura 5.6 apresenta a comparação da métrica de consumo de bateria entre o programa de teste e o jogo 2D. Quanto maior a barra, mais econômico é o dispositivo durante a execução da aplicação.

Figura 5.6 – Gráfico de comparação de consumo de bateria do *Frozen Bubble*



Fonte: Próprio autor

De acordo com a aplicação de teste: o Dev2 é o mais econômico para executar a operação de desenho da figura de um círculo, o Dev1 e Dev4 têm desempenho semelhante entre si e o Dev3 tem o maior gasto de bateria para a execução da operação 2D de desenho de figura circular. Ao analisar o gráfico, pode-se observar a mesma tendência do programa de teste, mesmo com uma melhora dos dispositivos menos eficientes.

5.1.4 GLTron[31]

Essa aplicação consiste em um jogo 3D no qual o jogador deve clicar na tela para direcionar uma moto através de um espaço plano. No entanto, não pode haver colisão com o rastro deixado por ele mesmo e por outras motos. Inicialmente, foi gerado um *benchmark* orientado com desenho de cubos 3D com a mesma textura do jogo real. Nessa aplicação, as métricas avaliadas foram a taxa de quadros desenhados por segundo e, também, o consumo de bateria, através da execução repetida do jogo.

Apesar de os resultados da métrica de quadros por segundo, do *benchmark* orientado, apresentar tendência semelhante ao jogo real, a métrica de consumo de bateria não apresentou comportamento semelhante. Uma hipótese para essa discrepância é que operação de teste

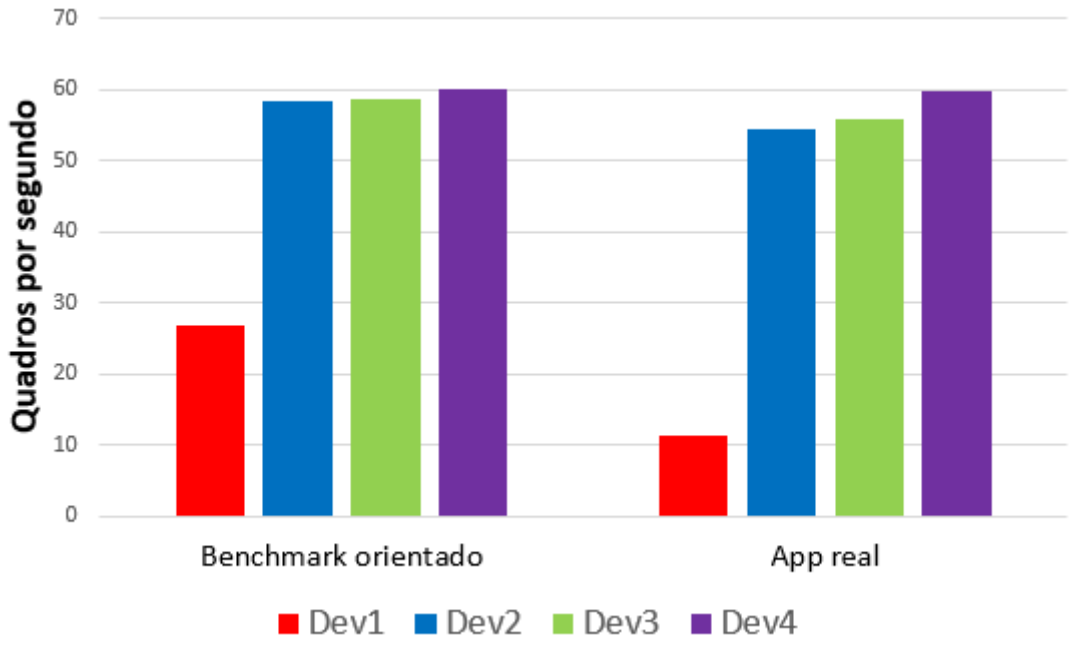
[31] https://play.google.com/store/apps/details?id=com.glTron&hl=pt_BR

disponível na biblioteca para o recurso 3D é muito simples, sendo capaz apenas de simular a textura de uma imagem, enquanto que no jogo diversas outras operações com a imagem são feitas. Mediante esse fato, optou-se por enriquecer a biblioteca com um trecho do código do próprio jogo, para confirmar a hipótese acima e visando melhor representar as necessidades da aplicação real. O trecho de código consiste em *renderizar* os modelos gráficos de quatro motos e a configuração de visão do modo de abertura do jogo. Foi, então, acrescentado a esse teste gráfico um teste de ponto flutuante com operações com cálculo de ângulos com o parâmetro *level* igual a '180'. Esse último teste visa representar a computação necessária para os movimentos que a moto faz no jogo real e que não fazem parte do trecho do jogo importado. Dessa forma, o *benchmark* orientado final consiste em duas unidades de teste da biblioteca: a unidade de teste com trecho do jogo 3D e a unidade de teste de ponto flutuante com cálculos de ângulos.

Para simular a utilização da aplicação de jogo 3D foi usada a ferramenta *Monkey* do *Android*. Foram gerados 6.000 eventos de toque na tela para execução do jogo. O intervalo entre os eventos é de 500 ms o que corresponde a cerca de 25 minutos de interação com a aplicação e possibilitando que fosse observado um consumo mínimo próximo a 10% da bateria do dispositivo. O jogo não foi alterado, pois ao finalizar uma seção, basta um novo toque na tela para recomeçar a jogar, permitindo, assim, a manutenção do ciclo de execução apenas através dos eventos de toque. Foram coletadas as métricas de taxa de quadros por segundo e também de consumo da bateria nesse período de execução dos jogos.

A Figura 5.7 apresenta a comparação da métrica de quadros por segundo entre o *benchmark* orientado e o jogo. Quanto maior a barra, maior a taxa de quadros e melhor o desempenho do dispositivo.

Figura 5.7 – Gráfico de comparação de quadro por segundo GlTron



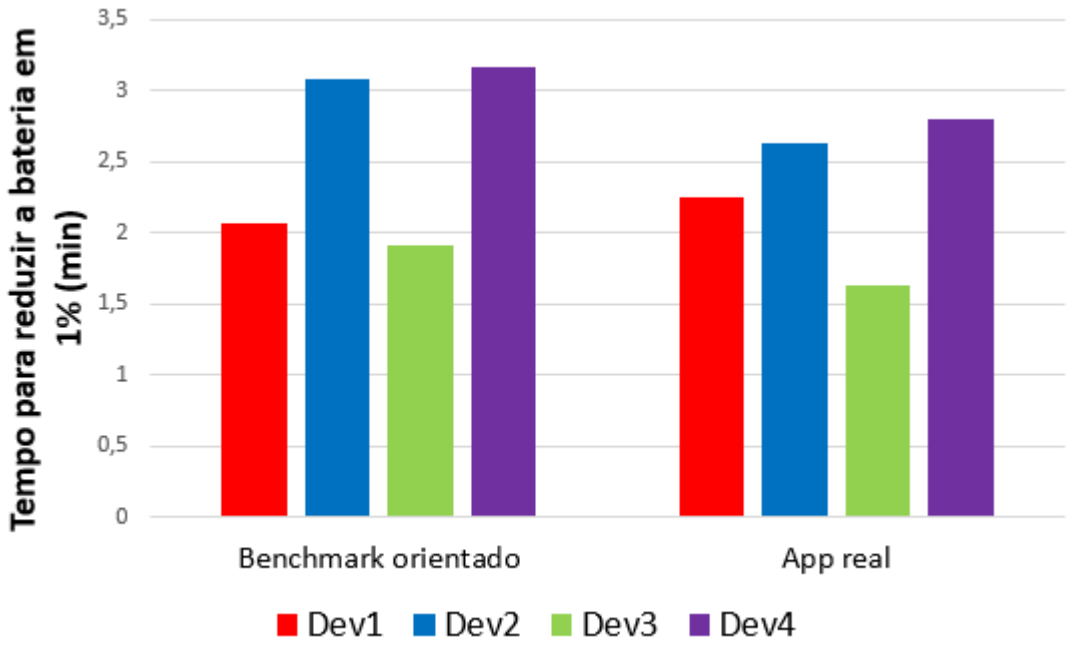
Fonte: Próprio autor

De acordo com a métrica coletada no *benchmark* orientado, é possível afirmar que o Dev1 apresenta o desempenho menos satisfatório ao realizar o jogo, sendo que os demais

dispositivos avaliados têm um desempenho semelhante com uma alguma vantagem do Dev4 em relação aos demais e uma pequena vantagem do Dev3 em relação ao Dev2. Ao analisarmos a métrica da aplicação real, vimos essa mesma tendência, apesar do desempenho do Dev1 ser inferior ao previsto na aplicação de teste. Essa diferença acontece porque o jogo tem mais processamento de gráfico que o trecho de código extraído e adicionado à biblioteca de testes.

A Figura 5.8 apresenta a comparação da métrica de consumo de bateria entre o *benchmark* orientado e o jogo GLTron. Quanto maior a barra, mais econômico é o dispositivo durante a execução da aplicação.

Figura 5.8 – Gráfico de comparação de consumo de bateria no GLTron



Fonte: Próprio autor

De acordo com a métrica prevista no *benchmark* orientado, os dispositivos Dev4 e Dev 2 tem um consumo de bateria semelhante ao executar o jogo 3D, mas com uma leve vantagem para o Dev4. O Dev1 é inferior a esses, mas apresenta o desempenho mais favorável do que o Dev3. Quando avaliamos o consumo de bateria da simulação do GLTron, observamos uma tendência semelhante à aplicação de testes. No entanto, o Dev1 apresentou um desempenho consideravelmente melhor do que o Dev3, mas ainda inferior ao Dev2.

5.1.5 K9 Mail[32]

Essa aplicação consiste em cliente de *e-mail* completo, permitindo o gerenciamento de múltiplas contas, tendo sido escolhida devido sua disponibilidade de código e suposta compatibilidade com os quatro dispositivos disponíveis. Apesar ser compatível, a princípio,

[32] https://play.google.com/store/apps/details?id=com.fsck.k9&hl=pt_BR

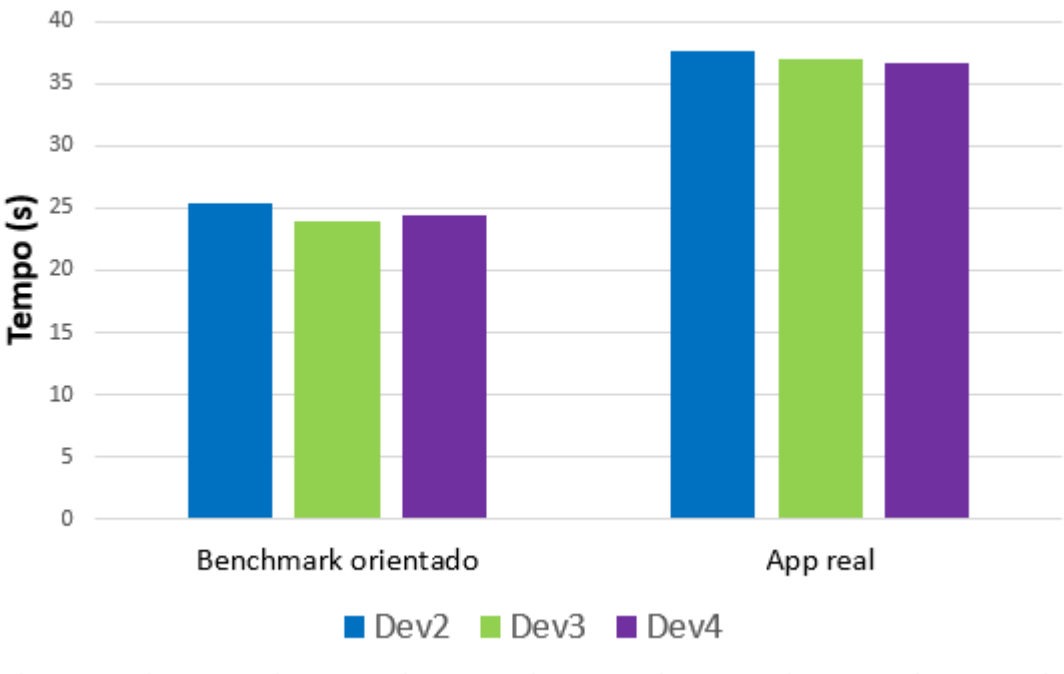
com dispositivos com *Android* 2.2, ao tentar gerar a aplicação para o Dev1 ocorreu um erro com alguns componentes da aplicação e não foi possível executá-la nesse dispositivo. Nessa aplicação, foi possível coletar valores para todas as métricas do *framework*.

O perfil dessa aplicação possui muitos serviços, mas visando a comparação entre a aplicação real e o teste disponível no *framework* foi avaliado o serviço de envio de *e-mail*. Na aplicação real, para se enviar um *e-mail* é necessário a sincronização de uma conta e, a partir dessa, realizar o envio de mensagens. O *benchmark* orientado que foi gerado continha apenas a operação de enviar *email* da categoria de Rede.

Para simular a utilização da aplicação real, foi feita a alteração no código para criar um *loop* que enviasse *e-mails* até que a bateria reduza seu percentual em 5%, sendo o *email* composto por um texto na linguagem HTML e um anexo de 1Mb. Foram feitas outras tentativas de simulação da aplicação, uma sem qualquer anexo e outra com anexo de 5MB. No entanto, a aplicação não suportou a repetição dessas operações e fechou durante o teste, o que inviabilizou sua utilização.

A Figura 5.9 mostra a comparação entre o tempo de envio de *e-mail* entre os dispositivos. O *benchmark* orientado apontou que os três dispositivos utilizaram um tempo semelhante para enviar os *e-mails*, apesar de algumas variações atribuídas à rede de envio das mensagens, sendo que houve diferenças consideráveis entre o envio de um *e-mail* e outro em um mesmo dispositivo. Na aplicação real, os três dispositivos tiveram um tempo semelhante de envio, mas essa também apresentou variações nos experimentos em um mesmo dispositivo levando às pequenas diferenças apresentadas no gráfico.

Figura 5.9 – Gráfico de comparação de tempo para envio de *e-mail*.



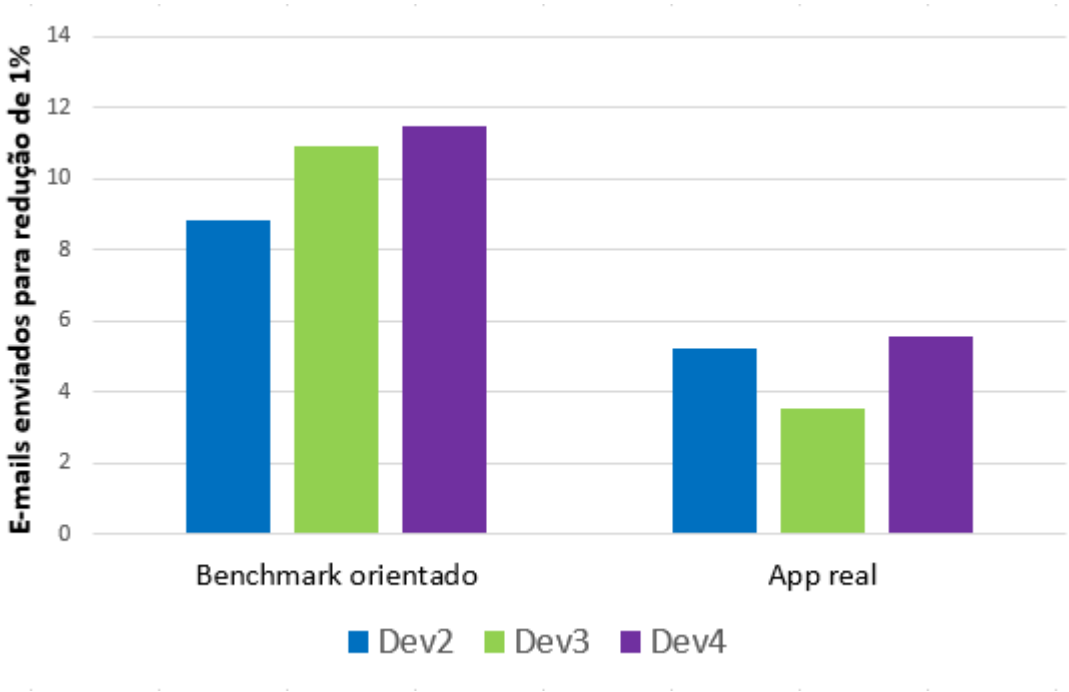
Fonte: Próprio autor

De acordo a primeira parte da Figura 5.10, na aplicação de teste o Dev2 apresenta o desempenho mais inferior, enviando uma quantidade menor de *e-mail* para a variação de 1% da bateria. O Dev4 tem o melhor desempenho seguido pelo Dev3. No entanto, na simulação da aplicação real o Dev3 se mostrou com pior desempenho em comparação aos demais.

Visando investigar o motivo dessa discrepância entre as estimativas e a aplicação real, foram observadas as outras métricas coletadas. Como mostrado na Figura 5.11, a métrica de quantidade de objetos alocados globalmente previa a semelhança entre a quantidade de itens entre os três dispositivos com uma pequena tendência de redução no Dev3. Entretanto, quando executada a simulação da aplicação real, foi percebido um aumento significativo da quantidade de objetos alocados pelo Dev3, indicando que devido a alocação de quase o triplo de objetos esse dispositivo teve uma redução no seu desempenho.

Essa diferença pode ser explicada por dois fatores: a relação entre a biblioteca utilizada para enviar os *e-mails* e a versão do sistema operacional; e, a semelhança entre a operação da biblioteca e da aplicação final. A aplicação de teste usa uma biblioteca para envio de *e-mails* disponível no Java, a JavaMail. Já a aplicação real não utiliza essa biblioteca, mas um código personalizado para implementar o protocolo SMTP para envio de *e-mails*. Como o Dev2 e Dev4 tem a mesma versão do sistema operacional, há um relacionamento semelhante tanto quando usam a biblioteca JavaMail[33] como o código personalizado do K9, fato que não acontece com o Dev3 que tem uma versão diferente de sistema operacional. Na aplicação final, muitos objetos necessitam ser instanciados antes de se enviar um *e-mail*, como por exemplo, o objeto que armazena dados da conta de usuário. Já na aplicação de teste a operação de envio de *e-mails* instancia apenas aqueles necessários para realizar a operação.

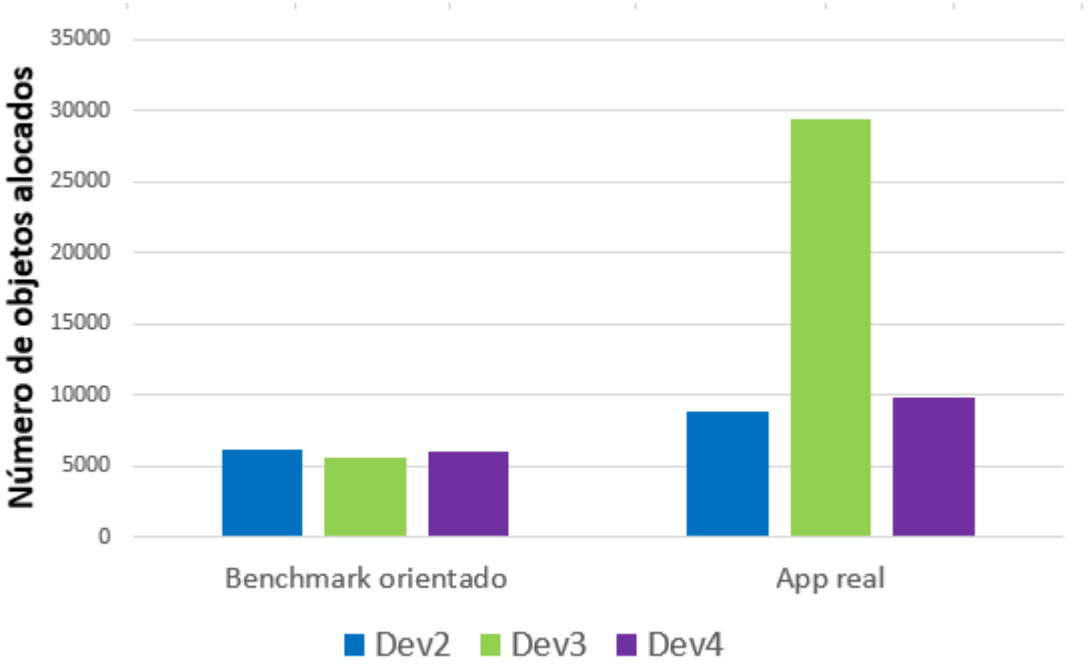
Figura 5.10 – Gráfico de comparação do consumo de bateria



Fonte: Próprio autor

Figura 5.11 – Gráfico de comparação da métrica de Objetos alocados globalmente

[33] http://www.oracle.com/technetwork/java/javamail/index.html



Fonte: Próprio autor

Como vimos na Tabela 1.1, os *benchmarks* genéricos nem sempre tem um consenso na classificação de dispositivos. O *framework* proposto visa guiar o desenvolvimento de *benchmarks* baseados nas necessidades reais das aplicações.

É importante ressaltar que os resultados foram coletados com dispositivos de tempo de uso distintos, sendo que isso pode influenciar na qualidade de suas baterias e, consequentemente, nos dados dessa métrica. É interessante pontuar que o consumo das baterias pode ser influenciado por elementos variados de uma aplicação, e o mecanismo de coleta dessa métrica não determina o consumo de cada um dos elementos de *hardware* ou *software*, mas o geral do dispositivo, composto pela combinação dos elementos de *hardware*, como exibição de cores na tela e ativação de sensores e as operações de teste propriamente ditas. Essa peculiaridade da bateria fortalece a premissa de que a biblioteca sempre deve ser enriquecida com mais operações de teste que serão cada vez mais semelhantes à aplicação final.

Mesmo com as ressalvas apresentadas acima, é possível afirmar que os resultados do *framework* são promissores em auxiliar o desenvolvedor em estimar os requisitos não funcionais de suas aplicações. Como a estrutura de coleta de métrica utiliza POA e essa foi apresentada como responsável por alguma sobrecarga em alguns trabalhos acadêmicos, por exemplo Chawla; Orso (2004), foram realizados, e são apresentados, na próxima seção experimentos para comparar a versão do *framework* com POA e a versão apenas com POO.

5.2 Experimentos com POA e POO

O uso da programação orientada a aspectos possibilitou estruturar o código do *framework* proposto, separando completamente o código voltado à coleta das métricas dos interesses do código referente às operações testadas. Essa separação, além de uma melhor organização,

aumenta o potencial de reúso do código do *framework*, além de facilitar sua expansão tanto na adição de novas métricas quanto na de novas operações de teste.

No entanto, o custo da orientação a aspectos no desempenho de uma aplicação é uma preocupação para os desenvolvedores. De fato, diversos trabalhos na área de programação orientada a aspectos têm avaliado o impacto de utilizar esse paradigma sobre o desempenho da aplicação. Por exemplo Chawla; Orso (2004).

De maneira geral, nos testes avaliados, o *overhead* por utilizar os *advices* do tipo *before* e *after* têm se mostrado insignificante, por exemplo, Dufour et al. (2004). Porém, como o *framework* proposto envolve dispositivos móveis, diferente da maioria dos trabalhos já realizados, é interessante avaliar se a abordagem utilizada afeta de modo significativo as métricas que estão sendo avaliadas.

Com o propósito de comparar a abordagem com orientação a aspectos com uma abordagem orientada a objetos, foi desenvolvida uma versão do *framework* com uma classe denominada *Metrics*. Essa é responsável por realizar a coleta das métricas a cada execução de uma operação de teste. Desse modo, é necessário acionar a classe métrica dentro do código responsável por efetuar as operações de teste. As Figuras 44 e 45 mostram o código de uma mesma operação de teste, sendo a primeira proveniente da versão do *framework* utilizando a programação orientada a aspectos e a segunda da abordagem orientada a objetos. É certo que outras versões de código orientado a objetos poderiam ser implementadas, mas inicialmente foi realizada a comparação apenas com uma versão.

Figura 5.12 – Trecho de código do método execute na versão com AspectJ.

```
@Override
public void execute() {
    People [] p = new People().createPeople(this.getLevel());
    testTJMmergesort(p);
    p = new People().createPeople(this.getLevel());
    testTJMquicksort(p);

    Bundle extras = new Bundle();
    extras.putBoolean(PerformanceTestActivity.RESULT_WAS_OK, true);
    activity.finishTest(extras);


}
```

Figura 5.13 – Trecho de código do método execute sem AspectJ.

```
@Override
public void execute() {
    People [] p = new People().createPeople(this.getLevel());
    Object[] parameters = new Object[1];
    Object[] arguments = new Object[1];

    Library.metrics.logBeforeTM();
    parameters[0] = "a";
    arguments[0] = p;
    testTJMmergesort(p);
    Library.metrics.logAfterTM("testTJMmergesort", parameters, arguments);

    p = new People().createPeople(this.getLevel());
    Library.metrics.logBeforeTM();
    parameters[0] = "a";
    arguments[0] = p;
    testTJMquicksort(p);
    Library.metrics.logAfterTM("testTJMquicksort", parameters, arguments);


    Bundle extras = new Bundle();
    extras.putBoolean(PerformanceTestActivity.RESULT_WAS_OK, true);
    activity.finishTest(extras);


}
```

O teste foi realizado criando-se uma aplicação hipotética com, pelo menos, uma operação de cada categoria da biblioteca de testes. Assume-se, nesta configuração, que o comportamento observado em uma operação pode ser replicado para as demais de uma mesma categoria. Foram criadas, então, duas versões da aplicação de teste: uma utilizando aspectos e a outra o código puramente orientado a objetos. Os testes foram realizados com dispositivo Xperia U, que tem sua especificação apresentada na segunda linha da

Tabela 1.2.

5.2.1 Resultados experimentais

Os experimentos foram repetidos dez vezes e os dados coletados salvos em arquivo .xml. A partir do arquivo de resultados, foram montadas tabelas com a comparação do valor médio para cada métrica de uma mesma operação, ora implementada com orientação a aspectos ora com orientação a objetos. As Tabelas 5.1 a 5.7 mostram esses resultados.

Para cada uma das categorias, as tabelas mostram dados de uma das operações nelas contidas. Para a categoria de CPU, por exemplo, a operação escolhida foi o cálculo de ponto flutuante com a execução do *Linpack*. Para a de memória, o teste foi a operação de ordenamento com o algoritmo *MergeSort*. Para a de rede, a operação escolhida foi a de *download* de arquivo. Na categoria de armazenamento (Arm), a escolhida foi a escrita sequencial em arquivo e na categoria de sensores foi escolhida a operação de acionamento do GPS. Para a categoria GPU, foi escolhida a operação de desenho de círculos. Ainda, foram

realizados testes de código nativo para CPU e armazenamento (Arm - N), seguindo as mesmas operações dessas categorias.

A Tabela 5.1 mostra que, em relação à métrica de tempo de execução, não se pode dizer que a POA influencie mais ou menos que POO, pois nenhuma das abordagens é sempre mais rápida que outra. Apesar de apresentar pequenas variações, os valores das métricas são próximos, não caracterizando que uma abordagem ou outra tenha influenciado as métricas.

Tabela 5.1 – Métrica de tempo de execução em milissegundos.

	Rede	*CPU*	*Sensores*	*CPU - N*	*Arm -N*	*Memória*	*Arm*
POA	5108,295	74,87045	13,01388	19,66218	2,602854	31,53549	276,949
POO	5260,916	76,50658	15,50608	19,69959	2,48669	33,11059	304,1707

Fonte: Próprio autor.

A Tabela 5.2 apresenta a métrica de quantidade de objetos que são alocados em todo o dispositivo durante o teste. Exceto pela categoria de armazenamento, os valores de ambas as versões de código mostraram-se próximos, não caracterizando que alguma das abordagens tenha influência nos valores das métricas.

Tabela 5.2 – Métrica de quantidade de objetos alocados globalmente.

	GPU	*Rede*	*CPU*	*Sensores*	*CPU - N*	*Arm -N*	*Memória*	*Arm*
POA	8280,03	2835,20	381,94	90,37	163,81	96,32	2016,16	252,77
POO	8230,00	2694,50	382,87	96,90	153,16	102,03	2008,19	267,48

Fonte: Próprio autor.

A Tabela 5.3 contém os dados da métrica de tamanho total dos objetos alocados em todo o sistema. Assim como a métrica de quantidade de objetos, os valores de ambas as versões são semelhantes mesmo na categoria de armazenamento na qual a quantidade de objetos alocados foi maior na versão com POO.

Tabela 5.3 – Métrica de tamanho de objetos alocados globalmente em bytes.

	GPU	*Rede*	*CPU*	*Sensores*	*CPU - N*	*Arm -N*	*Memória*	*Arm*
POA	460825,41	158238,80	383464,74	3841,14	7497,03	4451,29	120485,97	933524,71
POO	456539,95	150469,00	383514,45	4113,72	7150,45	4740,13	120156,68	934113,84

Fonte: Próprio autor.

A Tabela 5.4 mostra os dados da quantidade de objetos alocados pela *thread* que está realizando o teste. Como esperado, esses dados são mais próximos entre as versões chegando a serem idênticos nas categorias de Armazenamento e sensores.

Tabela 5.4 – Métrica de quantidade de objetos alocados pelo *thread*.

	GPU	*Rede*	*CPU*	*Sensores*	*CPU - N*	*Arm -N*	*Memória*	*Arm*
POA	8180,61	2693,20	225,55	64,85	24,00	24,00	1999,00	59,06
POO	8128,40	2660,75	225,61	64,93	25,06	24,13	1999,16	59,19

Fonte: Próprio autor.

A Tabela 5.5 apresenta os dados da métrica de tamanho dos objetos alocados, sendo que os dados de todas as categorias são similares, indicando que utilizar a programação orientada a aspectos não alterou, significativamente, a quantidade de memória necessária para o teste.

Tabela 5.5 – Métrica de tamanho de objetos alocados pelo *thread* em bytes.

	GPU	*Rede*	*CPU*	*Sensores*	*CPU - N*	*Arm -N*	*Memória*	*Arm*
POA	456652,65	153029,00	377309,16	2819,38	1380,00	1380,00	119792,00	926060,77
POO	452291,61	149674,00	377312,13	2821,65	1398,97	1384,65	119800,84	926067,81

Fonte: Próprio autor.

As Tabelas 5.6 e 5.7 apresentam a métrica de acionamento do coletor de lixo da máquina virtual, de forma que na primeira constam todos os acionamentos realizados no sistema operacional e na segunda estão aqueles que vieram da *thread* que está realizando o teste. Apenas os testes de CPU e Armazenamento geraram chamadas do coletor, pois os demais não tiveram qualquer chamada.

Tabela 5.6 – Métrica de acionamento do coletor de lixo globalmente

	CPU	*Arm*
POA	0,29	2,68
POO	0,26	2,94

Fonte: Próprio autor.

Tabela 5.7 – Métrica de acionamento do coletor de lixo oriundo do *thread*

	CPU	*Arm*
POA	0,29	2,68
POO	0,26	2,94

Fonte: Próprio autor.

Para o teste de GPU, ainda é possível obter a métrica de FPS, de forma que na versão com POA o valor da métrica foi 36,18 e na versão com POO 35,93. Novamente, os valores da métrica são bem próximos. Com relação à métrica de energia, a versão com POA utilizou 5%, enquanto que a puramente orientada a objetos gastou 4% para executar as dez repetições. A versão com POA realizou dois ciclos totais para cada percentual enquanto que a versão POO realizou 2,5 ciclos totais.

De modo geral, com base nos dados apresentados, é possível dizer que utilizar a POA não influencia diretamente as métricas. Assim, a utilização desse paradigma no *framework* é bastante interessante, pois não impacta significativamente as métricas e permite uma maior modularidade dessa ferramenta.

6 CONCLUSÃO E TRABALHOS FUTUROS

A atenção ao desempenho de requisitos não funcionais em aplicações móveis é importante para assegurar uma boa experiência para o usuário final. Nesse trabalho, foram estudadas diversas abordagem que avaliam o desempenho de sistemas móveis. Também, foram analisadas ferramentas disponíveis para desenvolvedores com o intuito de auxiliá-los na avaliação de suas aplicações.

Em uma primeira abordagem foram realizados experimentos com o objetivo de avaliar como estruturas de código podem afetar o desempenho de aplicações móveis. Foi identificado que, devido ao intenso uso de bibliotecas de código nas aplicações móveis, algumas refatorações no código da aplicação visando o aumento de desempenho podem ser ofuscadas pelo impacto introduzido por essas bibliotecas.

Em seguida, foi avaliada a aplicabilidade de *benchmarks* genéricos como ferramenta de auxílio ao desenvolvedor durante o projeto e implementação de uma aplicação móvel, levando-se em consideração a variedade de dispositivos alvos. Nesta avaliação, foi possível perceber que os *benchmarks* genéricos não apresentam um consenso em classificar os dispositivos, dependendo de cada implementação de *benchmark* para selecionar o melhor dispositivo. Além da dificuldade de se estabelecer qual operação prejudicou o resultado do *benchmark*.

Essa dissertação propôs, por fim, um *framework* visando a criação semiautomática de aplicações de teste (aqui também chamadas de *benchmarks* orientados à aplicação) para estimar de forma rápida, porém precisa, os requisitos não funcionais de aplicações móveis. O *framework* baseia-se em uma biblioteca de trechos de código, um conjunto de métricas e um mecanismo de geração do programa de teste. Esse *framework* foi desenvolvido utilizando *AspectJ* para possibilitar uma separação clara entre o código de execução dos testes e código de coleta das métricas.

Para a validação do *framework,* foram selecionadas cinco aplicações reais disponíveis na loja de aplicativos *Android* e, a partir dessas aplicações, foram gerados *benchmarks* específicos usados para estimar o comportamento não funcional da aplicação em um conjunto de dispositivos. Os experimentos mostraram que as aplicações de teste puderam indicar a tendência de consumo de recursos das aplicações reais. Foi possível observar, ainda, que essa tendência será cada vez mais próxima da realidade à medida que o código de testes seja semelhante ao das aplicações. Quando essa tendência não se mostrou equivalente entre a aplicação real e a aplicação de teste, analisou-se através de outras métricas os fatores que influenciaram essa discrepância.

O *framework* possibilita ainda que algumas operações sejam realizadas através de código Java ou do acionamento de código legado C/C++, funcionalidade que possibilita ao desenvolvedor comparar quando usar outra linguagem para desenvolver suas aplicações.

Os resultados fornecidos pelo *framework* proposto podem auxiliar empresas na escolha de dispositivos móveis a serem adquiridos visando um melhor custo-benefício ou podem orientar o desenvolvedor individual a ajustar sua aplicação com o objetivo de melhorar a experiência do usuário final. Isso é possível, pois o programa de teste é gerado de forma rápida e eficaz e pode ser executado nas primeiras etapas do desenvolvimento e antes mesmo da

implementação, quando as principais decisões de projeto são tomadas. Além disso, o *framework* fornece informações detalhadas sobre o desempenho de cada operação executada, bem como um mecanismo eficiente de definição da carga de trabalho de cada operação. Essa flexibilidade e velocidade permitem uma verdadeira exploração do espaço de projeto, permitindo explorar mais facilmente diferentes APIs, diferentes implementações de determinados serviços, diferentes estruturas de dados.

Apesar dos resultados promissores apresentados, o *framework* pode ser aprimorado em diversas dimensões. Do ponto de vista de usabilidade, pode ser desenvolvida uma interface gráfica para que o usuário possa selecionar de modo mais amigável os testes que serão realizados e também auxiliar na inclusão dos parâmetros para ponderar os testes. Ainda, é interessante desenvolver mecanismos de abstração para auxiliar o desenvolvedor na escolha dos testes. Do ponto de vista de precisão e completude da biblioteca de testes, deve-se considerar a inclusão de novos testes via código para que o desenvolvedor possa incluir trechos de código para experimentação ou que seja utilizado repetidamente. É possível, também, avaliar a possibilidade de inserção de arquivos binários de teste, na forma de componentes. Outro campo a ser explorado é o aperfeiçoamento na coleta da métricas, por exemplo melhorar a precisão da métrica de bateria. Por fim, sugere-se considerar o uso de técnicas de teste combinatório para a geração automática de cenários e dados de teste, como forma de automatizar e qualificar ainda mais o processo de avaliação da aplicação.

BIBLIOGRAFIA

ALEXANDERSSON, R.; ÖHMAN, P. On Hardware Resource Consumption for Aspect-Oriented Implementation of Fault Tolerance. In: EUROPEAN DEPENDABLE COMPUTING CONFERENCE, 2010...**Proceedings**, [S.I]:IEEE, 2010. p. 61–66. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5474196>. Acesso em: 15/11/2014.

ANTOCHI, I.; JUURLINK, B.; VASSILIADIS, S.; LIUHA, P. GraalBench. **ACM SIGPLAN Notices**, v. 39, n. 7, p. 1, 2004. ACM. Disponível em: <http://dl.acm.org/citation.cfm?id=998300.997165>. Acesso em: 26/2/2014.

BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In: ANNUAL CONFERENCE ON USENIX ANNUAL TECHNICAL CONFERENCE, 2005**. Proceedings**, Berkeley, CA, USA: USENIX Association, 2005. p.41:. Disponível em: <http://dl.acm.org/citation.cfm?id=1247360.1247401>. .

BERRY, M. J.; LINOFF, G. **Data mining techniques**: for marketing, sales, and customer support. [S.I.]: John Wiley, 1997.

CAELLUM. Guj - Android. Disponível em: <http://www.guj.com.br/tag/android>. .

CHAWLA, A.; ORSO, A. A Generic Instrumentation Framework for Collecting Dynamic Information. **Software Engineering Notes**. v. 29, p.5–8, 2004. ACM.

CHEN, G.; KANDEMIR, M.; VIJAYKRISHNAN, N.; IRWIN, M. J. PennBench: a benchmark suite for embedded Java. In: IEEE INTERNATIONAL WORKSHOP ON WORKLOAD CHARACTERIZATION, 2002... **Proceedings,** [S.I.]:IEEE, 2002. p.71–80. Disponível em: <http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1226495>. Acesso em: 13/2/2014.

DEBUSMANN, M.; GEIHS, K. Efficient and Transparent Instrumentation of Application Components Using an Aspect-Oriented Approach. In: **Self-Managing Distributed Systems**. [S.I.]: Stringer, 2013. p.209–220. Disponível em: <http://www.springerlink.com/content/ap2u6l0ct1x1jep2>.

DUFOUR, B.; GOARD, C.; HENDREN, L.; VERBRUGGE, C. Measuring the Dynamic Behaviour of AspectJ Programs. In: ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, 2004…**Proceedings**, New York, NY, USA: ACM, 2014. p. 150–169.

FERGUSON, P.; HUSTON, G. **Quality of Service**: Delivering QoS on the Internet and in Corporate Networks. New York: John Wiley, 1998.

FERNANDES, T. S.; COTA, É.; MOREIRA, Á. Performance Evaluation of Android Applications : a Case Study. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SISTEMAS COMPUTACIONAIS, 2014... **Anais**, Manaus:[s.n], 2014. p???

GATTI, S.; BALLAND, E.; CONSEL, C. A step-wise approach for integrating QoS throughout *software* development. , p. 217–231, 2011. Springer-Verlag. Disponível em: <http://dl.acm.org/citation.cfm?id=1987434.1987456>. Acesso em: 7/2/2014.

GONCALVES, J.; FERREIRA, L. L.; PINHO, L. M.; SILVA, G. Handling Mobility on a QoS-Aware Service-based Framework for Mobile Systems. In: IEEE/IFIP INTERNATIONAL CONFERENCE ON EMBEDDED AND UBIQUITOUS COMPUTING, 2010… **Proceedings**, [S.I.]:IEEE, 2010. p. 97–104. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5703504>. Acesso em: 4/2/2014.

GUTHAUS, M. R.; PINGENBERG, J. S.; EMST, D.; et al. MiBench : A free , commercially representative embedded benchmark suite. In: WORKLOAD CHARACTERIZATION, 2001…**Proceedings**, Washington, DC, USA:IEEE, 2001. p. 3–14.

GUTIERREZ, A.; DRESLINSKI, R. G.; WENISCH, T. F.; et al. Full-System Analysis and Characterization of Interactive Smartphone Applications. In: IEEE INTERNATIONAL SYMPOSIUM ON WORKLOAD CHARACTERIZATION (IISWC), 2011…**Proceedings**, [S.I.]: IEEE, 2011. p. 81–90.

HASSAN, S. I. An empirical evaluation of the impact of aspectization of cross-cutting concerns in a Smart-phone based application. In: INTERNATIONAL CONFERENCE ON COMPUTING FOR SUSTAINABLE GLOBAL DEVELOPMENT (INDIACOM), 2014**. Proceedings**, [S.I.]:IEEE, 2014. p.448–454. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6828178>. Acesso em: 25/11/2014.

JAIN, R. **The Art of Computer Systems Performance Analysis**: Techniques for Experimental Design, Measurement, Simulation, and Modeling. [S.I]: Wiley, 1991.

KAYANDE, D.; SHRAWANKAR, U. Performance analysis for improved RAM utilization for Android applications. In: CSI SIXTH INTERNATIONAL CONFERENCE ON *SOFTWARE* ENGINEERING (CONSEG), 2012**. Proceedings**, [S.I.]:IEEE, 2012. p. 1–6. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6349500>.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; et al. An Overview of AspectJ. In: 15TH EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 2001**. Proceedings**, London, UK, UK: Springer-Verlag, 2001. p.327–353. Disponível em: <http://dl.acm.org/citation.cfm?id=646158.680006>. .

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; et al. Aspect-oriented programming. In: ECOOP'97-OBJECT-ORIENTED PROGRAMMING, 1997… **Proceedings**, Berlin: Springer, 1997. p.220–242, 1997.

KIM, H.; AGRAWAL, N.; UNGUREANU, C. Examining storage performance on mobile devices. In: **ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds - MobiHeld '11**, 2011…**Proceedings**, New York, New York, USA: ACM Press, 2011. p. 1–6. Disponível em: <http://dl.acm.org/citation.cfm?doid=2043106.2043112>. .

KIM, J.; KIM, J. AndroBench : Benchmarking the Storage Performance of Android-Based Mobile Devices. In: **Frontiers in Computer Education**, [S.I.]:Springer, 2012 p. 667–674.

KIM, K. J.; HWANG, E. H.; CHO, S. J.; et al. Web based multi-platform benchmark program construction in smartphone. IN: 7TH INTERNATIONAL CONFERENCE ON UBIQUITOUS INFORMATION MANAGEMENT AND COMMUNICATION - ICUIMC '13, 2013**. Proceedings**, New York, New York, USA: ACM Press, 2013. p. 1–9. Disponível em: <http://dl.acm.org/citation.cfm?doid=2448556.2448625>.

KIM, Y.; CHO, S.; KIM, K.; et al. Benchmarking Java Application Using JNI and Native C Application on Android. , p. 284–288, 2012.

LEE, C.; KIM, E.; KIM, H. **The AM-Bench :** An Android Multimedia Benchmark Suite. [S.I.]:[s.n], 2012.

LEE, C.; POTKONJAK, M.; MANGIONE-SMITH, W. H. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 1997…**Proceedings**, [S.I]:IEEE Computer Society, 1997. p. 330–335.

LEE, S.; JEON, J. W. Evaluating Performance of Android Platform Using Native C for Embedded Systems. In: INTERNATIONAL CONFERENCE ON CONTROL AUTOMATION AND SYSTEMS (ICCAS), 2010…**Proceedings**, [S.I.]: IEEE, 2010. p. 1160–1163.

LIN, C.-M.; LIN, J.-H.; DOW, C.-R.; WEN, C.-M. Benchmark Dalvik and Native Code for Android System. In: SECOND INTERNATIONAL CONFERENCE ON INNOVATIONS IN BIO-INSPIRED COMPUTING AND APPLICATIONS, 2011**. Proceedings**, [S.I.]:IEEE, 2011. p. 320–323, 2011. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6118781>. Acesso em: 24/3/2014.

LIU, W.-L.; LUNG, C.-H.; AJILA, S. Impact of Aspect-Oriented Programming on *Software* Performance: A Case Study of Leader/Followers and Half-Sync/Half-Async Architectures. In: IEEE 35TH ANNUAL COMPUTER *SOFTWARE* AND APPLICATIONS CONFERENCE, 2011**. Proceedings**, [S.I]:IEEE, 2011. p. 662–667. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6032414>. Acesso em: 15/11/2014.

LUIZ SARTOR, A.; BRISOLARA CORREA, U.; SCHNEIDER BECK, A. C. AndroProf: A Profiling Tool for the Android Platform. In: III BRAZILIAN SYMPOSIUM ON COMPUTING SYSTEMS ENGINEERING (SBESC), 2013. **Proceedings**, [S.I]:IEEE, 2014 . p.23–28.

MA, H.; YEN, I.-L.; ZHOU, J.; COOPER, K. QoS analysis for component-based embedded *software*: Model and methodology. **Journal of Systems and *Software***, v. 79, n. 6, p. 859–870, 2006. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121205001548>. Acesso em: 6/2/2014.

NGUYEN, D. T. Evaluating impact of storage on smartphone energy efficiency. In: ACM CONFERENCE ON PERVASIVE AND UBIQUITOUS COMPUTING ADJUNCT PUBLICATION - UBICOMP '13, 2013**. Proceedings**, New York, New York, USA: ACM Press, 2013. p. 319–324. Disponível em: <http://dl.acm.org/citation.cfm?doid=2494091.2501083>.

OKANOVIĆ, D.; VAN HOORN, A.; KONJOVIĆ, Z.; VIDAKOVIĆ, M. SLA-driven adaptive monitoring of distributed applications for performance problem localization. In: **Computer Science and Information Systems (ComSIS)**,[S.I]:[s.n] v. 10, n. 1, p. 25–50, 2013.

OKANOVIĆ, D.; VIDAKOVIĆ, M.; KONJOVI, Z. Towards Performance Monitoring Overhead Reduction. In: INTERNATIONAL SYMPOSIUM ON INTELLIGENT SYSTEMS AND INFORMATICS (SISY), 2013…**Proceedings**, [S.I.]: IEEE, 2013.p. 135–140.

PUNNIYAKOTTI, M. **BASCin: Benchmark for Android from Santa Cruz**. 2012. 46 f. Dissertação (Mestrado em Engenharia da Computação) -- University of California, Santa Cruz, 2012. Disponível em: <http://www.escholarship.org/uc/item/4j330932>.

SCHOEBERL, M. Application experiences with a real-time Java processor. In: IFAC WORLD CONGRESS, 2008…**Proceedings**, Seoul, Korea: [s.n.], 2008.

SCHOEBERL, M.; PREUSSE, T. B.; UHRIG, S. The embedded Java benchmark suite JemBench.In: 8TH INTERNATIONAL WORKSHOP ON JAVA TECHNOLOGIES FOR REAL-TIME AND EMBEDDED SYSTEMS - JTRES '10, 2010**Proceedings**, New York, New York, USA: ACM Press, 2010. p.120–127. Disponível em: <http://dl.acm.org/citation.cfm?id=1850771.1850789>. Acesso em: 13/2/2014.

SOMMERVILLE, I. ***Software*** **Engineering**. 9th ed. Upper Saddle River, NJ: Addison Wesley, 2010.

UTI, N. V.; FOX, R. Testing the Computational Capabilities of Mobile Device Processors: Some Interesting Benchmark Results.In: IEEE/ACIS 9TH INTERNATIONAL CONFERENCE ON COMPUTER AND INFORMATION SCIENCE, 2010**.Proceedings**, [S.I]:IEEE,2010. p. 477–481. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5590492>. Acesso em: 4/2/2014.

WAC, K. Towards QoS-Awareness of Context-Aware Mobile. In: **On the Move to Meaningful Internet Systems 2005**: OTM 2005 Workshops, [S.I.]:Springer, 2005. p. 751–760.

WAC, K.; ICKIN, S.; HONG, J.; et al. Studying the Experience of Mobile Applications Used in Different Contexts of Daily Life Categories and Subject Descriptors. In: ACM SIGCOMM WORKSHOP ON MEASUREMENTS UP THE STACK, 2011…**Procedings**, [S.I]:ACM, 2011. p. 7–12. Disponível em: <http://doi.acm.org/10.1145/2018602.2018605>. .

WANG, J.; YIN, P.; ZHANG, H. Analysis and measurement on factors influencing the performance of mobile platform. In: 2ND INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE AND NETWORK TECHNOLOGY, 2012**.Proceedings**, [S.I.]:IEEE, 2012. p. 116–119. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6525903>. .

YE, Y.; JAIN, N.; XIA, L.; et al. A Framework for QoS and Power Management in a Service Cloud Environment with Mobile Devices. In: FIFTH IEEE INTERNATIONAL SYMPOSIUM ON SERVICE ORIENTED SYSTEM ENGINEERING, 2010**.Proceedings**, [S.I.]: IEEE, 2010. p. 236–243. Disponível em:

<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5569901>. Acesso em: 4/2/2014.

YE, Y.; XIAO, L.; YEN, I.-L.; BASTANI, F. Leveraging Service Clouds for Power and QoS Management for Mobile Devices.In: IEEE 4TH INTERNATIONAL CONFERENCE ON CLOUD COMPUTING, 2011**.Proceedings**, [S.I.]:IEEE, 2011. p. 235–242. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6008715>. Acesso em: 4/2/2014.

ZECHNER, M. Libgdx. Disponível em: <http://libgdx.badlogicgames.com>. Acesso em: 30/1/2015.

APÊNDICE <ARQUIVO DE COLETA DE MÉTRICAS COM POA>

```
public aspect MetricsByAspects {
private long start = 0;
private long start2 = 0;
private long startActivity = 0;
final double nanoSegRate = 1000000.0;
BufferedWriter out = null;
//Variables to get FPS rate
protected long startTime;
protected long fpsStartTime;
protected long numFrames;

public MetricsByAspects() {

	try {

		out = new BufferedWriter(new FileWriter((new File(
				Library.DATA_FILE_NAME))));
		out.write("<?xml version=\"1.0\" encoding=\"utf-8\"?>" + "\n");
		out.write("<performanceData \n\t\t\t
xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" \n\t\t\t
xsi:noNamespaceSchemaLocation=\"model.xsd\">\n");
		out.flush();
	} catch (FileNotFoundException e) {
		e.printStackTrace();
		return;
	} catch (IOException e) {
		e.printStackTrace();
	}
}

/**
 * Esse pointcut define o joinpoint para todas as chamadas de métodos
```

```
     * nas classes de operações de código nativo
     */
    @Pointcut("call(* edu.performance.test.*.*.*.testN*(..)) ||"
                    + "call(* edu.performance.test.*.*.*.testTNM*(..))")
    public void metricsNativeMethods() {

    }
    /**
     * Esse pointcut define o joinpoint para todas as chamadas de métodos
     * nas classes de operações de JAVA
     */
    @Pointcut("call(* edu.performance.test.*.*.testA*(..)) || "
                    + "call(* edu.performance.test.*.*.testJ*(..)) || "
                    + "call(* edu.performance.test.*.*.testTJM*(..))")
    public void metricsJavaMethods() {

    }
    /**
     * Esse pointcut é usado para não monitorar possíveis chamadas de métodos na classe
     * Library e BatteryMetric
     */
    @Pointcut("call(* edu.performance.test.Library.*.testJ*(..)) ||"
                    + "within(Library) ||"
                    + "within(edu.performance.test.batterytest.BatteryMetric)")
    public void inNotMonitoredClasses() {

    }
    /**
     * Esse pointcut define o joinpoint para todas as chamadas de métodos
     * na classe de operações de BD
     */
    @Pointcut("call(* edu.performance.test.*.*.testTpJM*(..))")
    public void metricsDBMethods() {
```

```
	}


	@Pointcut("execution(* edu.performance.test.screenoperation.*.execute(..))")
	public void startMetricsFromUI() {

	}

	@Pointcut("call(* edu.performance.test.screenoperation.*.finish(..)) || "
			+ "call(* edu.performance.test.streamingoperation.*.finish(..))")
	public void finishMetricsFromUI() {

	}
	/**
	 * Esse pointcur define o joinpoint da criação das interfaces gráficas nas
	 * unidades de teste que compõem a categoria GPU
	 */
	@Pointcut ("execution(* edu.performance.test.*.*.*.surfaceCreated(..)) || "
			+  "execution(* edu.performance.test.*.*.surfaceCreated(..)) || "
			+  "execution(* edu.performance.test.*.*.*.onSurfaceCreated(..)) || "
			+  "execution(*
edu.performance.test.streamingoperation.*.onSurfaceTextureAvailable(..))")
	public void startGraphicMetrics(){

	}
	/**
	 * Esse pointcut define o joinpoint do método de desenho nas
	 * unidades de teste que compõem a categoria GPU
	 */
	@Pointcut("call(* edu.performance.test.*.*.*.doDraw(..)) || "
			+ "call(* edu.performance.test.*.*.doDraw(..)) || "
			+ "execution(* edu.performance.test.*.*.*.onDrawFrame(..)) ||"
```

```
                        + "execution(*
edu.performance.test.streamingoperation.*.onSurfaceTextureUpdated(..))")
        public void finishGraphicMetrics() {

        }
        /**
         *
         */
        @Pointcut("call(* edu.performance.test.Library.finish(..))")
        public void finishTestProgram() {

        }

        /**
         * Esse advice é utilizado para iniciar a coleta das métricas
         * de todos os métodos JAVA ou fim da Activities de Screen e Streaming.
         */
        @Before("(metricsJavaMethods() || startMetricsFromUI()) &&
!inNotMonitoredClasses()")
        public void logBefore(JoinPoint joinPoint) {
                Debug.resetAllCounts();
                start = System.nanoTime();
                start2 = SystemClock.uptimeMillis();

                Debug.startAllocCounting();
        }
        /**
         * Esse advice é utilizado para finalizar a coleta das métricas
         * de todos os métodos JAVA ou fim da Activities de Screen e Streaming.
         * Métricas coletadas:
         * - Tempo de CPU
         * - Tempo de execução
         * - Quantidade de objetos alocados global e localmente
```

```
	 * - Tamanho dos objetos alocados global e localmente
	 * - Acionamentos do GC global e localmente
	 */
	@After("(metricsJavaMethods() || finishMetricsFromUI()) &&
!inNotMonitoredClasses()")
	public void logAfter(JoinPoint joinPoint) {
		Debug.stopAllocCounting();
		double elapsedTime = (System.nanoTime() - start) / nanoSegRate;
		double elapsedTime2 = (SystemClock.uptimeMillis() - start2);
		long tempoCpu = android.os.Process.getElapsedCpuTime();
		int allocCountG = Debug.getGlobalAllocCount();
		int allocSizeG = Debug.getGlobalAllocSize();
		int gcInvocationG = Debug.getGlobalGcInvocationCount();
		int gcInvocationT = Debug.getThreadGcInvocationCount();
		int allocCountT = Debug.getThreadAllocCount();
		int allocSizeT = Debug.getThreadAllocSize();
		Debug.resetAllCounts();

		try {
			out.append("\t<method> \n"
					+ "\t\t<name>" + joinPoint.getSignature().getName() +
"</name>\n"
					+ "\t\t<methodParameters>" +
print(((MethodSignature)joinPoint.getSignature()).getParameterNames()) +
"</methodParameters>\n"
					+ "\t\t<methodArguments>" + print(joinPoint.getArgs())
+ "</methodArguments>\n"
					+ "\t\t<cpuTime>" + tempoCpu + "</cpuTime>\n"
					+ "\t\t<elapsedTime>"		+ elapsedTime +
"</elapsedTime>\n"
					+ "\t\t<allocCountG>" + allocCountG +
"</allocCountG>\n"
					+ "\t\t<allocSizeG>" + allocSizeG + "</allocSizeG>\n"
```

```
                                        + "\t\t<gcInvocationG>" + gcInvocationG +
"</gcInvocationG>\n"
                                        + "\t\t<allocCountT>" + allocCountT +
"</allocCountT>\n"
                                        + "\t\t<allocSizeT>" + allocSizeT + "</allocSizeT>\n"
                                        + "\t\t<gcInvocationT>" + gcInvocationT +
"</gcInvocationT>\n"
                                        + "\t\t<elapsedTime2>"        + elapsedTime2 +
"</elapsedTime2>\n"
                                        + "\t</method>\n");
                        out.flush();

                } catch (IOException e) {

                        System.err.println("No possible to write in file. logAfter");
                }

        }
        /**
         * Esse advice é utilizado para iniciar a coleta das métricas dos métodos
         * nativos
         */
        @Before("metricsNativeMethods()")
        public void logBeforeNative(JoinPoint joinPoint) {
                Debug.resetAllCounts();
                start = System.nanoTime();
                start2 = SystemClock.uptimeMillis();

                Debug.startAllocCounting();
                Debug.startNativeTracing();
        }
        /**
         * Esse advice é utilizado para finalizar a coleta das métricas dos métodos
```

```
	 * nativos.
	 * Métricas coletadas:
	 * - Tempo de CPU
	 * - Tempo de execução
	 * - Quantidade de objetos alocados global e localmente
	 * - Tamanho dos objetos alocados global e localmente
	 * - Acionamentos do GC global e localmente
	 * - Tamanho do heap
	 * - Heap alocado
	 */
	@After("metricsNativeMethods()")
	public void logAfterNative(JoinPoint joinPoint) {
		Debug.stopAllocCounting();
		Debug.stopNativeTracing();
		double elapsedTime = (System.nanoTime() - start) / nanoSegRate;
		double elapsedTime2 = (SystemClock.uptimeMillis() - start2);
		long tempoCpu = android.os.Process.getElapsedCpuTime();
		long nativeheapSize = Debug.getNativeHeapSize();
		long nativeallocated = Debug.getNativeHeapAllocatedSize();
		int allocCountG = Debug.getGlobalAllocCount();
		int allocSizeG = Debug.getGlobalAllocSize();
		int gcInvocationG = Debug.getGlobalGcInvocationCount();
		int gcInvocationT = Debug.getThreadGcInvocationCount();
		int allocCountT = Debug.getThreadAllocCount();
		int allocSizeT = Debug.getThreadAllocSize();
		Debug.resetAllCounts();


		try {
			out.append("\t<method>\n"
					+ "\t\t<name>" + joinPoint.getSignature().getName() +
"</name>\n"
```

```
                                        + "\t\t<methodParameters>" +
print(((MethodSignature)joinPoint.getSignature()).getParameterNames()) +
"</methodParameters>\n"
                                        + "\t\t<methodArguments>" + print(joinPoint.getArgs())
+ "</methodArguments>\n"
                                        + "\t\t<cpuTime>" + tempoCpu + "</cpuTime>\n"
                                        + "\t\t<elapsedTime>"          + elapsedTime +
"</elapsedTime>\n"
                                        + "\t\t<heapSize>" + nativeheapSize + "</heapSize>\n"
                                        + "\t\t<allocatedHeap>" + nativeallocated +
"</allocatedHeap>\n"
                                        + "\t\t<allocCountG>" + allocCountG +
"</allocCountG>\n"
                                        + "\t\t<allocSizeG>" + allocSizeG + "</allocSizeG>\n"
                                        + "\t\t<gcInvocationG>" + gcInvocationG +
"</gcInvocationG>\n"
                                        + "\t\t<allocCountT>" + allocCountT +
"</allocCountT>\n"
                                        + "\t\t<allocSizeT>" + allocSizeT + "</allocSizeT>\n"
                                        + "\t\t<gcInvocationT>" + gcInvocationT +
"</gcInvocationT>\n"
                                        + "\t\t<elapsedTime2>"        + elapsedTime2 +
"</elapsedTime2>\n"
                                        + "\t</method>\n");
                        out.flush();

                } catch (IOException e) {

                        System.err.println("No possible to write in file. logAfterNative");
                }

        }
        /**
```

```
 * Esse advice é usado para iniciar as métricas de operações
 * de banco de dados
 */
@Before("metricsDBMethods()")
public void logBeforeTpJM(JoinPoint joinPoint) {
        Debug.resetAllCounts();
        start = System.nanoTime();
        start2 = SystemClock.uptimeMillis();

        Debug.startAllocCounting();
}
/**
 * Esse advice é usado para finalizar as métricas de operações
 * de banco de dados.
 * Métricas coletadas:
 * - Tempo de CPU
 * - Tempo de execução
 * - Quantidade de objetos alocados global e localmente
 * - Tamanho dos objetos alocados global e localmente
 * - Acionamentos do GC global e localmente
 * - Throughput
 */
@After("metricsDBMethods()")
public void logAfterTpJM(JoinPoint joinPoint) {
        Debug.stopAllocCounting();
        double elapsedTime = (System.nanoTime() - start) / nanoSegRate;
        double elapsedTime2 = (SystemClock.uptimeMillis() - start2);
        int allocCountG = Debug.getGlobalAllocCount();
        int allocSizeG = Debug.getGlobalAllocSize();
        int gcInvocationG = Debug.getGlobalGcInvocationCount();
        int gcInvocationT = Debug.getThreadGcInvocationCount();
        int allocCountT = Debug.getThreadAllocCount();
        int allocSizeT = Debug.getThreadAllocSize();
```

```
		Debug.resetAllCounts();
		long tempoCpu = android.os.Process.getElapsedCpuTime();
		try {
			out.append("\t<method>\n"
					+ "\t\t<name>" +
joinPoint.getSignature().toShortString() + "</name>\n"
					+ "\t\t<methodParameters>" +
print(((MethodSignature)joinPoint.getSignature()).getParameterNames()) +
"</methodParameters>\n"
					+ "\t\t<methodArguments>" + print(joinPoint.getArgs())
+ "</methodArguments>\n"
					+ "\t\t<cpuTime>" + tempoCpu + "</cpuTime>\n"
					+ "\t\t<elapsedTime>"		+ elapsedTime +
"</elapsedTime>\n"
					+ "\t\t<allocCountG>" + allocCountG +
"</allocCountG>\n"
					+ "\t\t<allocSizeG>" + allocSizeG + "</allocSizeG>\n"
					+ "\t\t<gcInvocationG>" + gcInvocationG +
"</gcInvocationG>\n"
					+ "\t\t<allocCountT>" + allocCountT +
"</allocCountT>\n"
					+ "\t\t<allocSizeT>" + allocSizeT + "</allocSizeT>\n"
					+ "\t\t<gcInvocationT>" + gcInvocationT +
"</gcInvocationT>\n"
					+ "\t\t<elapsedTime2>"		+ elapsedTime2 +
"</elapsedTime2>\n"
					+ "\t\t<throughput>"  +
myToInteger(joinPoint.getArgs()[0]) /(elapsedTime/1000f) + "</throughput>\n"
					+ "\t</method>\n");
			out.flush();
		} catch (IOException e) {

			System.err.println("No possible to write in file. logAfterTpJM");
```

```
        }

}
/**
 * Esse advice é usado para iniciar a coleta de todas as métricas
 * relacionadas a categoria GPU
 */
@Before("startGraphicMetrics()")
public void logBeforeActivity(JoinPoint joinPoint) {
        Debug.resetAllCounts();
        Debug.startAllocCounting();
        startActivity = SystemClock.uptimeMillis();
        startTime = SystemClock.uptimeMillis();
        fpsStartTime = startTime;
        numFrames = 0;
}

/**
 * Esse advice é usado para iniciar a coleta de todas as métricas
 * relacionadas a categoria GPU.
 * Métricas coletadas:
 * - Tempo de CPU
 * - Tempo de execução
 * - Quantidade de objetos alocados global e localmente
 * - Tamanho dos objetos alocados global e localmente
 * - Acionamentos do GC global e localmente
 * - FPS
 */
@After("finishGraphicMetrics()")
public void logAfterActivity(JoinPoint joinPoint) {
        long fpsElapsed = SystemClock.uptimeMillis() - fpsStartTime;
        numFrames++;
        if (fpsElapsed > 5 * 1000) {
```

```
                double elapsedTime = (SystemClock.uptimeMillis() - startActivity);
                Debug.stopAllocCounting();
                String fpsString = "";

                        float fps = (numFrames * 1000.0F) / fpsElapsed;

                        fpsString = fpsString.concat("\t\t<fps>" + fps + "</fps>\n");

                        fpsStartTime = SystemClock.uptimeMillis();
                        numFrames = 0;

                int allocCountG = Debug.getGlobalAllocCount();
                int allocSizeG = Debug.getGlobalAllocSize();
                int gcInvocationG = Debug.getGlobalGcInvocationCount();
                int gcInvocationT = Debug.getThreadGcInvocationCount();
                int allocCountT = Debug.getThreadAllocCount();
                int allocSizeT = Debug.getThreadAllocSize();
                Debug.resetAllCounts();
                long tempoCpu = android.os.Process.getElapsedCpuTime();
                try {
                        out.append("\t<method>\n"
                                        + "\t\t<name>" +
joinPoint.getTarget().getClass().getSimpleName() + "</name>\n"
                                        + "\t\t<methodParameters>" +
print(((MethodSignature)joinPoint.getSignature()).getParameterNames()) +
"</methodParameters>\n"
                                        + "\t\t<methodArguments>" + print(joinPoint.getArgs())
+ "</methodArguments>\n"
                                        + "\t\t<cpuTime>" + tempoCpu + "</cpuTime>\n"
                                        + "\t\t<elapsedTime>"          + fpsElapsed +
"</elapsedTime>\n"
                                        + "\t\t<allocCountG>" + allocCountG +
"</allocCountG>\n"
```

```
                                        + "\t\t<allocSizeG>" + allocSizeG + "</allocSizeG>\n"
                                        + "\t\t<gcInvocationG>" + gcInvocationG +
"</gcInvocationG>\n"
                                        + "\t\t<allocCountT>" + allocCountT +
"</allocCountT>\n"
                                        + "\t\t<allocSizeT>" + allocSizeT + "</allocSizeT>\n"
                                        + "\t\t<gcInvocationT>" + gcInvocationT +
"</gcInvocationT>\n"
                                        + (!fpsString.trim().isEmpty() ? fpsString : "")
                                        + "\t</method>\n");
                        out.flush();
                                fpsString = "";
                                if(elapsedTime < 15000) //TODO Se é a ultima execução da
operação gráfica
                                        Debug.startAllocCounting();

                } catch (IOException e) {

                        System.err.println("No possible to write in file. logAfterActivity");
                }
                }

        }

        /**
         * Esse advice finaliza a coleta das métricas.
         */
        @Before("finishTestProgram()")
        public void logBeforeFinish(JoinPoint joinPoint) {
                System.out.println(joinPoint.getSourceLocation().getFileName());
                try {
                        out.write("</performanceData>");
                        out.flush();
```

```
                        out.close();
                        System.out.println("Closing " + Library.DATA_FILE_NAME + " file
at MetricsByAspect!!");
                } catch (IOException e) {
                        e.printStackTrace();
                }

        }
        /**
         * Esses métodos são utilizados para formatar os parâmetros dos métodos.
         */
        private String print(Object objets[]){
                if(objets == null || objets.length == 0)
                        return "";

                String message = "";
                for(Object s : objets){
                        if(s != null)
                                if(s instanceof Cursor)
                                        message = message.concat("\n\t\t\t<item>\"" +
((Cursor)s).getCount() + "\"</item>");
                                else
                                        message = message.concat("\n\t\t\t<item>\"" +
                                ((s.toString() != null && s.toString().length() < 80)? s.toString()
: "TooBigOrNull!" ) + "\"</item>");
                        }

                return message;
        }
                private Integer myToInteger(Object o){
                if(o instanceof Integer)
                        return (Integer) o;
                else if (o instanceof String)
```

```
                return Integer.parseInt((String)o);
        else if(o instanceof Cursor)
                return ((Cursor)o).getCount();
        else
                return 1;
    }
}
```