UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

EWERTON FELIPE MIGLIORANZA

# Estudo Comparativo de Linguagens de Programação para Manipulação de Vídeo em Telefones Móveis

Trabalho de Graduação.

Prof. Dr. Valter Roesler
Orientador

Porto Alegre, dezembro de 2009

*Dedico este trabalho aos meus pais, Félix e Soeli, por acreditarem em mim dando tudo de si, à minha irmã Thaís pelas horas de descontração e à minha namorada Bruna, por todo o apoio durante a realização deste. Amo vocês!*

# AGRADECIMENTOS

Gostaria de agradecer ao meu orientador, Prof. Valter Roesler, pela oportunidade deste trabalho, bem como a todos os meus professores pelo conhecimento concedido durante a minha graduação. Agradeço também à todas as pessoas do laboratório de Projetos em Áudio e Vídeo da UFRGS (PRAV) pelas incontáveis discussões e sugestões neste e em outros trabalhos: ao Alexandro Bordignon pelas diversas revisões e conselhos, ao André e ao Fernando pelas dicas em Java, ao Júlio e ao Guilherme pelos trabalhos em TV Digital e a todos que formam essa grande equipe.

Quero agradecer também a todos os amigos que conheci durante a graduação: ao Fábio pelas diversas reflexões teóricas, ao Oscar por compartilhar o gosto pela computação gráfica, à Marilena por ser minha parceira nos trabalhos mais difíceis da graduação, ao Rodrigo e ao Thiago pelo excelente trabalho que vêm desenvolvendo.

Por fim, agradeço também a todos aqueles que me ajudaram direta ou indiretamente durante esta etapa da minha vida. Muito obrigado!

# SUMÁRIO

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| AAC | Advanced Audio Coding |
| ADPCM | Adaptative Differential Pulse-Code Modulation |
| AMR | Adaptative Multi-Rate |
| ACL | Access Linux Plataform |
| API | Application Programming Interface |
| ASF | Advanced Systems Format |
| COM | Component Object Model |
| DLL | Dynamic-Link Library |
| GUID | Globally Unique Identifier |
| HTML | HyperText Markup Language |
| IDE | Integrated Development Environment |
| IP | Internet Protocol |
| JAR | Java Archive |
| JDK | Java Development Kit |
| JIT | Just in Time |
| JME | Java Micro Edition |
| JRE | Java Runtime Environment |
| JSR | Java Specification Requests |
| JVM | Java Virtual Machine |
| KVM | K Virtual Machine |
| Kbps | Kilobit por segundo |
| LOC | Lines of Code |
| PDA | Personal Digital Assistant |
| QoS | Quality of Service |
| RIM | Research in Motion |
| SDK | Software Development Kit |

| | |
|---|---|
| SMS | Short Message Service |
| MMAPI | Mobile Media API |
| Mbps | Megabit por segundo |
| MMS | Multimedia Messaging Service |
| UDP | User Datagram Protocol |
| URI | Uniform Resource Identifier |
| VoIP | Voice over Internet Protocol |
| WLAN | Wireless Local Area Network |
| WMA | Windows Media Audio |
| WMV | Windows Media Video |

# LISTA DE FIGURAS

# LISTA DE TABELAS

# RESUMO

A expansão do mercado de telefonia móvel impulsiona o surgimento de novos aparelhos e tecnologias, juntamente com diferentes meios de programá-los. Dentre essas tecnologias, a troca de informações multimídias está entre as mais promissoras, pois oferece mais capacidade de comunicação ao usuário final. Contudo, devido às inúmeras plataformas e linguagens de programação disponíveis, é comum se deparar com o problema de como analisar e escolher a melhor combinação para esse contexto. Logo, este trabalho realiza um estudo comparativo entre as principais plataformas móveis e linguagens de programação disponíveis, considerando características qualitativas referentes ao projeto das linguagens, características quantitativas focadas no desempenho de aplicações para manipulação e transmissão de vídeo em telefones móveis, bem como as ferramentas e os recursos oferecidos.

**Palavras-chave:** Telefonia móvel, celular, vídeo, multimídia, linguagens de programação, comparação.

**A Comparative Study of Programming Languages for Video Manipulation on Mobile Phones**

# ABSTRACT

The expansion of the mobile telephony market drives the development of new devices and technologies, and with it, new ways to program. Among these technologies, the multimedia transmission is one of the most promising, because it offers more communication capabilities to the end user. However, due to the many platforms and programming languages available, it is common to have questions on how to analyze and choose them for this context. Therefore, this work makes a comparative study of the main mobile platforms and programming languages available, considering qualitative characteristics of the languages project, quantitative characteristics focused in the performance of applications that manipulate and transmit video in mobile phones, as well as the resources and tools available.

**Keywords:** mobile telephony, cellphone, video, multimedia, programming languages, comparison.

# 1 INTRODUÇÃO

## 1.1 A Evolução da Telefonia Móvel

O termo telefonia móvel surgiu a partir da sua primeira utilização: permitir às pessoas usarem o telefone enquanto estão em movimento. De acordo com (FITZEK; REICHERT, 2007), durante a primeira geração (1G), dominada por equipamentos analógicos, não havia necessidade de se oferecer outros serviços além de voz. Contudo, a partir da segunda geração (2G), com a mudança para sistemas digitais, houve um grande avanço na capacidade de se prover serviços diferenciados, como mensagens SMS (Short Message Service) e conexão com a *Internet*.

A divisão entre a rede celular (operadora) e o provedor de serviços (Figura 1.1) foi outra mudança importante. Anteriormente, a operadora da rede celular detinha controle total sobre os serviços oferecidos aos telefones móveis (apenas voz nesse caso). Após essa divisão, novos serviços surgiram por parte dos provedores (por exemplo, *download* de conteúdos digitais como jogos e toques), mas ainda restringidos pela operadora.

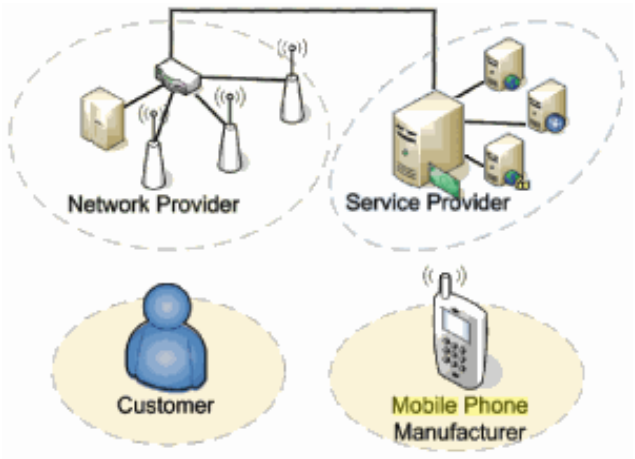


Figura 1.1: Modelo de uma rede celular e suas entidades. (FITZEK; REICHERT, 2007)

Com o surgimento de tecnologias *wireless* como WLAN (Wireless Local Area Network) e *Bluetooth*, o controle das operadoras sobre os serviços começou a enfraquecer, pois o usuário passou a ter formas alternativas de acessar mais serviços. Deste modo, o controle de acesso a tais serviços passou a ser feito a partir do próprio aparelho, ou seja, os fabricantes, pressionados pelas operadoras que comercializam seus planos juntamente com o telefone, passaram a manter essa nova liberdade sob controle, restringindo o acesso à certas partes do *hardware*. Um exemplo disso é a grande liberdade em termos de desenvolvimento de aplicativos por parte de terceiros para telefones móveis, visto que a grande

maioria dos aparelhos oferecem acesso a recursos básicos, como câmera, tela e teclado. Porém, quando o assunto é sensível à operadora, como o VoIP[1], o acesso aos recursos (de rede, neste caso) são bem mais restritos.

Por fim, o advento da terceira geração (3G) trouxe melhorias às tecnologias introduzidas com a 2G, possibilitando uma maior largura de banda e, consequentemente, viabilizando serviços anteriormente deixados em segundo plano, como a troca de conteúdos multimídia, que eram limitados até então pelo custo dos aparelhos e pelo limite de qualidade imposto (baixas resoluções e taxas de transmissão). Observa-se, no entanto, que esses serviços agora despontam como novas oportunidades para o mercado de telefonia móvel pois, como demonstrado por K. O'hara (O'HARA; MITCHELL; VORBAU, 2007), o consumo de conteúdos multimídia através de telefones móveis é uma tendência crescente, visto a possibilidade de se receber e transmitir vídeo, como em mensagens MMS (Multimedia Messaging Service).

Dessa forma, com uma nova gama de serviços disponíveis, um novo fator mercadológico surgiu, onde os consumidores passam a buscar as aplicações que melhor utilizem esses serviços, deixando de se basear somente na tecnologia empregada. Em outros termos, o diferencial de um telefone celular para o consumidor final passou a ser os aplicativos que ele suporta, e não mais as tecnologias que ele utiliza. Um exemplo disso é o iPhone (APPLE, 2009) que, mesmo utilizando a mesma tecnologia dos demais aparelhos do mercado (3G), conseguiu se sobressair por oferecer aplicativos diferenciados (NYTIMES, 2009).

Dado a forma da evolução da telefonia móvel apresentada, é possível concluir que o desenvolvimento de aplicativos para telefones móveis está em constante crescimento e se torna cada vez mais importante. Ao exemplo da *Internet* convencional (em *desktops*), onde a demanda por troca de conteúdos multimídias tem se tornado cada vez maior (como é possível observar através do crescimento de serviços como YouTube [2] e derivados), os celulares despontam agora como plataformas capazes de inovar e estender esses serviços, como mostram as estatísticas da Teleco (TELECO, 2009).

Grandes avanços, por parte dos fabricantes de telefones móveis, têm sido feitos para oferecer tais serviços ao usuário final, melhorando a qualidade dos aparelhos (como telas e câmeras com mais resolução). Por outro lado, há também um esforço para oferecer meios de programá-los (mesmo que as vezes de forma restrita), visto que o mercado está baseado nos aplicativos oferecidos e que plataformas abertas e acessíveis facilitam o desenvolvimento de tais aplicativos.

## 1.2 Objetivos

Baseado no cenário descrito nas seções anteriores, o objetivo geral deste trabalho será realizar um levantamento dos diversos ambientes disponíveis para se manipular vídeo em telefones móveis, como sistemas operacionais e APIs (Application Programming Interface), realizando uma análise comparativa entre as principais linguagens de programação utilizadas através de metodologias qualitativas e quantitativas, a fim de esclarecer quais as vantagens e desvantagens encontradas para diferentes contextos, como captura, envio e recepção de vídeo em tempo real. Para isso, o objetivo específico deste trabalho

[1] VoIP (Voice over Internet Protocol): Permite realizar chamadas utilizando o protocolo IP pela *Internet*. Deste modo, chamadas pelo telefone celular por VoIP não seriam mais tarifadas de acordo com a operadora da rede celular, e sim, do provedor do serviço.

[2] Youtube: http://www.youtube.com/

é implementar aplicativos em linguagens diferentes que exemplifiquem esse contexto e, em seguida, utilizá-los como base para realizar uma análise comparativa das linguagens empregadas.

## 1.3 Trabalhos Relacionados

O estudo realizado neste trabalho surgiu da necessidade de se conhecer e comparar as principais formas disponíveis para se manipular vídeo em telefones móveis. Na literatura atual, existem diversos trabalhos que abordam comparações de linguagens de programação, mas comumente essas comparações são limitadas a análises quantitativas (medidas de desempenho) e restringidas a contextos específicos, que normalmente não se aplicam aos telefones móveis. Contudo, alguns trabalhos encontrados, listados a seguir, serviram de base para a realização deste.

No trabalho de V. M. Cruz (CRUZ; MORENO; SOARES, 2008) é realizada uma comparação entre diversas plataformas e linguagens de programação existentes para telefones móveis, porém, focando a implementação de um *middleware* para a TV Digital. Visto que o foco do trabalho é apenas um estudo das ferramentas já disponíveis, não é apresentado uma análise detalhada dessas comparações. Contudo, o trabalho analisa fatores importantes como a portabilidade, a arquitetura e a eficiência de sistemas para as plataformas Symbian OS e JME (Java Micro Edition).

Já no trabalho de C. Campo (CAMPO; NAVARRETE; GARCIA-RUBIO, 2007) é feito uma análise mais aprofundada sobre o desempenho de aplicações em Symbian e JME que utilizam a câmera do celular, mas tratando apenas aspectos quantitativos da captura do vídeo, como tamanho e resolução, não relevando questões de implementação, ferramentas disponíveis e análises qualitativas.

Outros trabalhos que abordam o uso de *stream* de vídeos em dispositivos móveis é o de F. Coriolano (FáBIO S. CORIOLANO, 2008) e o de R. Clemente (CLEMENTE, 2006). Nestes trabalhos são apresentadas algumas soluções para se transmitir e reproduzir vídeo em um dispositivo móvel através da MMAPI Java. Embora não façam uso da câmera integrada para gerar vídeo, os estudos sobre os conceitos e protocolos de transmissão podem ser utilizados tanto para receber como enviar *streams* de vídeo e áudio, uma vez que protocolos de comunicação em geral são independentes da plataforma utilizada.

Por fim, o trabalho de L. Prechelt (PRECHELT, 2000) realiza uma análise comparativa entre várias linguagens de programação. Mesmo essas linguagens sendo voltadas para outra plataforma e estando fora do escopo deste trabalho, essa pesquisa releva técnicas quantitativas que podem ser usadas na comparação de linguagens de programação.

Assim, este trabalho visa complementar esses estudos realizando análises quantitativas e qualitativas aplicadas ao contexto de criação e transmissão de conteúdos multimídia em dispositivos móveis.

## 1.4 Estrutura

Adiante, a estrutura deste trabalho segue da seguinte forma.

No capítulo 2, **Dispositivos Móveis**, são apresentados conceitos introdutórios para o desenvolvimento deste trabalho, com definições acerca dos aparelhos móveis, suas limitações e os formatos e meios utilizados para a transmissão de conteúdos multimídia. É apresentado também uma descrição acerca das principais plataformas disponíveis no mercado, bem como as linguagens e ferramentas suportadas por estas, as quais influenciaram

a escolha das linguagens avaliadas Java e C++.

No capítulo 3, **Metodologias de Validação**, é citado e explicado as metodologias de validação utilizadas, dividindo-as em métricas qualitativas e quantitativas. Juntamente, é feito um adendo explicando e exemplificando as principais diferenças no projeto das linguagens empregadas. Este capítulo serve então, de base para a avaliação das linguagens e dos aplicativos desenvolvidos ao longo deste trabalho.

O capítulo 4 discorre sobre a **Implementação do Aplicativo**, explicando o projeto do aplicativo proposto e como o mesmo foi implementado usando as linguagens propostas, considerando suas diferentes ferramentas e limitações.

O capítulo 5 é destinado aos **Testes e Validação**, onde os conceitos descritos e formulados ao longo do trabalho são aplicados aos exemplos desenvolvidos, fornecendo uma base teórica e prática que relacione as vantagens e desvantagens de cada linguagem, direcionando o estudo ao contexto de transmissão de vídeo em telefones móveis.

Por fim, o capítulo 6 trás as **Conclusões** deste trabalho, descrevendo os principais pontos e as considerações finais sobre o estudo realizado ao longo deste. Em seguida, é apresentado as referências bibliográficas utilizadas e uma última seção com apêndices sobre assuntos secundários.

# 2 DISPOSITIVOS MÓVEIS

Dispositivos móveis se caracterizam por serem pequenos aparelhos capazes de oferecer a conveniência e a assistência de um computador convencional, porém com algumas limitações, como será detalhado a seguir.

Como exemplo de dispositivos móveis comumente utilizados pode-se citar celulares, smartphones e PDAs (Personal Digital Assistant), bem como câmeras digitais, vídeo games portáteis, entre outros. Porém, para fins deste trabalho, refere-se a dispositivo móvel aquele capaz de interagir através de uma rede de telefonia móvel ou celular, como os aparelhos celulares, smartphones e PDAs.

Não há uma clara distinção entre as categorias de telefones móveis. Geralmente refere-se ao celular como o aparelho com funcionalidades básicas de comunicação por voz e dados. Já um *smartphone*, normalmente é considerado como um celular com funcionalidades adicionais, como acesso a *Internet*, *e-mails*, teclado completo, etc. Por fim, um PDA possui adicionalmente serviços que facilitam tarefas corporativas, como agendas, calendários, além de tela sensível ao toque para anotações e outras atividades. Desse modo, este trabalho é focado em dispositivos móveis capazes de se comunicar através de redes de telefonia celular e de interagir com componentes multimídia, como som, vídeo e imagem.

Assim, devido à vasta gama de tipos e modelos de aparelhos disponíveis, é de se esperar que os mesmos possuam diferentes sistemas e modos de interagir com conteúdos multimídia, implicando em várias plataformas ou sistemas operacionais que oferecem suporte a várias linguagens.

## 2.1 Limitações dos Dispositivos Móveis

Em virtude do seu tamanho reduzido, os dispositivos móveis possuem algumas limitações a serem levadas em consideração ao se desenvolver aplicativos. Dentre elas pode-se citar:

- *Consumo de energia.* Quanto menor o consumo da bateria, maior o tempo de uso do aparelho.
- *Limite de processamento e de memória.* Recursos limitados pelo uso de energia e pelo tamanho reduzido.
- *Menor banda disponível.* Qualidade e disponibilidade do sinal depende da área de cobertura.
- *Interação com o usuário limitada.* Dispositivos de interação, como teclado e tela, são menores.

- *Compatibilidade entre diversas plataformas. Hardware* muito específico e APIs de desenvolvimento fechadas reduzem a compatibilidade entre as plataformas.

Deste modo, é necessário garantir que os aplicativos se adaptem a essas limitações, incluindo o sistema operacional, o qual desempenha um papel chave no desenvolvimento para dispositivos portáteis. A partir dele é possível saber quais são as APIs disponíveis para programação e as ferramentas que elas oferecem para gerenciar os recursos do dispositivo.

## 2.2 Transmissão multimídia e o celular

Um conteúdo multimídia visa integrar diversas mídias como textos, sons, imagens e vídeos, representando um grande poder de expressão. Dada a crescente necessidade de comunicação entre as pessoas, e a facilidade de se replicar conteúdos digitais, advém a necessidade de se transmitir e compartilhar tais conteúdos de uma forma cada vez mais abrangente.

Com o avanço da tecnologia, transmissões multimídia se tornam cada vez mais ricas, provendo maior qualidade em termos de imagem, som e vídeo, ao mesmo tempo em que requerem novos meios para suportar essa transmissão. Esses meios envolvem uma melhor codificação do conteúdo e a sua forma de transmissão.

Dentre as codificações mais utilizadas no contexto de telefones móveis, pode-se citar:

- *3GP*: O padrão 3GP (3GPP, 2008) representa um contêiner de formatos de áudio e vídeo, ou seja, um arquivo no formato 3GP pode ser codificado utilizando-se um conjunto padrão de codificadores. Para o áudio, é possível utilizar os codificadores AAC (Advanced Audio Coding) ou AMR (Adaptative Multi-Rate). Já no caso do vídeo, são disponibilizados os codificadores MPEG-4 Part 2, H.263 e H.264/AVC.


Figura 2.1: Modelo do contêiner de formatos 3GP (criado pelo autor).

- *Flash Video*: O contêiner de formatos Flash Video, propriedade da Adobe Systems Inc.[1], é amplamente utilizado para a transmissão de vídeos na *internet* através do Adobe Flash Player. Por ser facilmente embutido em páginas HTML, ao exemplo do YouTube [2], se tornou rapidamente um padrão de escolha em ambientes *web*.

Dentre os vários codificadores empregados (alguns proprietários), os principais são baseados nos padrões de vídeo H.263, H.264 e VP6 (On2 Technologies Inc.[3]), e nos padrões de áudio ADPCM (Adaptative Differential Pulse-Code Modulation), MP3 e AAC.

- *RealMedia*: O padrão RealMedia, propriedade da RealNetworks Inc.[4] também é um contêiner de formatos, encapsulando os codificadores (proprietários) RealVideo e RealAudio. Este padrão é otimizado para trabalhar com *streaming* de vídeos, sendo utilizado para transmissão em ambientes como a *internet*.

- *Advanced Systems Format*: O Advanced Systems Format (ASF) é um contêiner de formatos proprietário da Microsoft[5]. Esse padrão suporta os codificadores, também proprietários da Microsoft, Windows Media Video (WMV) e Windows Media Audio (WMA), bem como implementações proprietárias do codificador MPEG-4.

Já para a transmissão de dados entre celulares, segundo Dhir (DHIR, 2004), é possível utilizar os seguintes meios:

- *Rede celular*: A conexão pela rede celular permite transmitir dados utilizando as mesmas frequências que o aparelho usa para enviar e receber voz. Assim, a operadora se encarrega de repassar esses dados para um *gateway* conectado à *internet* ou para um provedor de serviços. A tecnologia empregada na rede celular pode ser dividida basicamente da seguinte forma:

  - *2G*: Redes 2G foram construídas basicamente para tráfego de voz e baixa transmissão de dados. Por ser uma tecnologia geralmente baseada em conexão (*connection based*), é necessário que ela seja efetivamente estabelecida antes de qualquer troca de dados, aumentando a latência de comunicação. A baixa taxa de transmissão (em torno de 9.6Kbps), aliada ao alto custo por *byte* transmitido, faz com que transmissões multimídias se tornem inviáveis em redes 2G.
  - *2.5G*: Dada a inviabilidade de prover transmissões multimídias das redes 2G, surgiu o modelo de rede 2.5G. Ainda utilizando os mesmos espectros de sinais da 2G, a 2.5G trouxe algumas melhorias na codificação dos sinais de rádio e um modelo baseado em pacotes (*packet-based*), provendo serviços sempre *online* (sem necessidade de conexão) e maiores taxas de transmissão (entre 14.4Kbps e 384Kbps).
  - *3G*: O modelo de rede 3G surgiu para suportar uma vasta gama de serviços de voz, dados e multimídia sobre redes móveis e fixas, combinando alta velocidade de acesso móvel com melhorias em serviços baseados em IP (Internet Protocol), como voz, texto e dados. Provendo melhorias nas tecnologias empregadas na 2.5G e um espectro maior para transmissão de dados, uma rede 3G é capaz de oferecer taxas de transmissão de 44Kbps (carros em movimento), 384kbps (pedestres) e 2Mbps (estacionários), provendo serviços multimídias de alta qualidade.

[1] Youtube: http://www.youtube.com/
[2] Adobe Systems Inc.: http://www.adobe.com/
[3] On2 Technologies Inc.: http://www.on2.com/index.php
[4] RealNetworks Inc.: http://www.realnetworks.com/
[5] Microsoft ASF: http://www.microsoft.com/windows/windowsmedia/forpros/format/asfspec.aspx

- *Conexão WLAN*: WLANs (*Wireless Local Area Network*) disponibilizam conexões sem fio de alta velocidade. Geralmente utilizada em meios corporativos, esse tipo de conexão tem se tornado popular pela facilidade de uso e baixo custo de instalação, oferecendo acesso à *internet* e transmissões de dados a taxas de até 54Mbps. Ao contrário da 3G, que é basicamente baseada em telefonia e lida com voz e dados ao mesmo tempo, WLANs são puramente orientadas a dados, sendo assim, uma boa solução para transmissões que possuem um alto consumo de banda, como vídeos. Contudo, o raio de cobertura de uma WLAN (algumas dezenas de metros) é cerca de 10.000 vezes inferior a de uma estação base 3G (cerca de 15 quilômetros), o que acaba limitando a mobilidade.

## 2.3 Plataformas Disponíveis

Atualmente há diversas plataformas disponíveis para o desenvolvimento de aplicativos para móveis. O estudo, porém, será focado nas que possuem maior representatividade no mercado e que ofereçam APIs de programação abertas.

De acordo com a Tabela 2.1 por (GARTNER, 2008), o uso do Symbian OS lidera a parcela do mercado de sistemas operacionais para dispositivos móveis, seguido pelo RIM (Research in Motion), Windows Mobile, Linux, Mac OS X e Palm OS. A seguir é apresentada uma análise mais detalhada sobre cada plataforma.

Tabela 2.1: Vendas de Smartphones por Sistema Operacional, em unidades (GARTNER, 2008).

| Company | 2Q08 Sales | 2Q08 Market Share (%) | 2Q07 Sales | 2Q07 Market Share (%) | 2Q08- 2Q07 Growth (%) |
|---|---|---|---|---|---|
| Symbian | 18,405,057 | 57.1 | 18,273,255 | 65.6 | 0.7 |
| Research In Motion | 5,594,159 | 17.4 | 2,471,200 | 8.9 | 126.4 |
| Microsoft Windows Mobile | 3,873,622 | 12.0 | 3,212,222 | 11.5 | 20.6 |
| Linux | 2,359,245 | 7.3 | 2,816,490 | 10.1 | -16.2 |
| Mac OS X | 892,503 | 2.8 | 270,000 | 1.0 | 230.6 |
| Palm OS | 743,910 | 2.3 | 461,918 | 1.7 | 61.0 |
| Others | 352,679 | 1.1 | 349,501 | 1.3 | 0.9 |
| **Total** | **32,221,175** | **100.0** | **27,854,586** | **100.0** | **15.7** |

### 2.3.1 Symbian OS

Desenvolvido inicialmente por uma parceria entre indústrias de dispositivos portáteis composto pela Nokia, Ericsson, Sony Ericsson, Panasonic e Samsung, o Symbian OS atualmente pertence à Nokia, a qual adquiriu o sistema e hoje o distribui livre de royalties, em uma tentativa de alavancar ainda mais o uso e o desenvolvimento de aplicações para esta plataforma.

O Symbian OS (JODE; TURFUS, 2004) é um sistema projetado especificamente para dispositivos móveis, sendo assim, seu desenvolvimento é focado no gerenciamento otimizado da memória e do consumo de energia, incorporando elementos que evitam usos

desnecessários e excessivos destes recursos, como o conceito de Active Objects (SYMBIAN, 2004).

Outro fator que torna esta plataforma mais conhecida, é o suporte ao desenvolvimento de aplicações em diversas linguagens, como Symbian C++, Java, Python, .NET, Ruby, OPL, Perl e AdobeFlash. No entanto, as duas primeiras acabam sendo as mais utilizadas, onde o Symbian C++ é a linguagem nativa do sistema e o Java por ser altamente difundido no mercado atual.

Logo, esses fatores contribuem para que o Symbian OS seja um dos sistemas mais utilizados atualmente, possuindo uma grande comunidade e diversas ferramentas que auxiliam o desenvolvimento de aplicações para a plataforma.

### 2.3.2 RIM

A RIM, fabricante do BlackBerry, possui um sistema operacional proprietário fechado, sendo dedicado aos seus produtos.

Embora ofereça pacotes de desenvolvimento baseados em Java e o ambiente visual BlackBerry MDS Studio (KIRKUP, 2007), a sua falta de interação com outros dispositivos, alto acoplamento do software com o hardware exclusivo do BlackBerry, APIs proprietárias e a necessidade de assinaturas digitais da própria RIM acabam por encarecer o desenvolvimento de aplicações, resultando em uma comunidade de desenvolvimento pequena em comparação às outras plataformas.

### 2.3.3 Windows

A plataforma Windows para dispositivos embarcados é baseada no modelo de componentes do Windows CE, na qual é possível agregar diversos componentes em um sistema operacional para atender necessidades específicas. Através da escolha desses componentes, visando atender os requisitos de um sistema para o mercado de dispositivos móveis, surgiu o Windows Mobile.

A vantagem de se utilizar o Windows Mobile é que suas ferramentas para desenvolvimento se assemelham às oferecidas para sua versão *desktop*, altamente difundidas, como o Visual Studio e o .NET juntamente com o suporte às linguagens C++, C# e Visual Basic (MICROSOFT, 2008). Essas ferramentas são oferecidas tanto de forma gratuita, porém limitada, como o eMbedded Visual C++, e de forma paga mas completas como o Windows Mobile Developer Resource Kit.

Contudo, alguns problemas conhecidos na versão *desktop* também tendem a aparecer na versão mobile, como vazamentos e uso desnecessário de memória, principalmente no uso de aplicativos de terceiros, uma vez que o gerenciamento correto da memória deve ser garantida pelo programador.

### 2.3.4 Linux

Bastante conhecido por ser um sistema operacional de código aberto para *desktops*, a comunidade do Linux tem se esforçado para oferecer distribuições para dispositivos móveis. Por se tratar de um sistema aberto, o Linux se torna altamente flexível, podendo ser configurado para trabalhar em diversas plataformas de maneira eficiente.

Outro fator que auxilia o desenvolvimento de aplicativos para a plataforma é a sua alta portabilidade, onde um sistema desenvolvido para desktops pode ser portado quase que diretamente (com o uso das devidas bibliotecas) para um dispositivo móvel. Há também o fato de que diversas ferramentas para desenvolvimento estão disponíveis de forma

gratuita, utilizadas em ambas as plataformas. Sendo assim, como na versão *desktop*, as linguagens mais utilizadas são C++ e Java.

Apesar dessas vantagens, frequentemente a configuração de um sistema Linux pode se tornar uma tarefa complicada, devido à dificuldade de se encontrar *drivers* para o hardware do dispositivo móvel. Além disso, versões comerciais mais estáveis geralmente são pagas, e nem sempre se consegue uma compatibilidade entre elas. Dentre as versões mais comuns estão a Montavista, A-la-Mobile e Android.

### 2.3.5 iPhone OS

Desenvolvido pela Apple e derivado do Mac OS X, o iPhone OS surgiu recentemente com o lançamento do iPhone e do iPod Touch. Baseado em um sistema do tipo Unix, o iPhone OS oferece suporte à aplicações em Objective-C, uma variante da linguagem C com mensagens no estilo de Smalltalk.

Inicialmente um sistema fechado, havia poucas possibilidades de se desenvolver programas para a plataforma, se limitando a alguns aplicativos *web* feitos em *javascript* e HTML através do navegador nativo Safari. Contudo, a Apple recentemente abriu sua plataforma de desenvolvimento para suportar aplicações de terceiros, as quais são distribuídas através da sua loja *online*, a Apple Store.

Logo, para se desenvolver aplicativos para o iPhone, é necessário pagar pela API proprietária da Apple, o iPhone SDK (APPLEOS, 2009). Porém, uma vantagem dessa SDK (Software Development Kit) é que ela possui o mesmo ambiente de desenvolvimento utilizado nos desktops que rodam Mac OS X, o Xcode, bem como ferramentas usadas para simulações (emuladores) e análises de desempenho.

### 2.3.6 Palm OS

Plataforma de grande sucesso no passado, o Palm OS tem passado dificuldades ultimamente para se sobressair no mercado em relação aos seus concorrentes. Embora tenha anunciado o lançamento de duas novas versões, o Palm OS Garnet para dispositivos mais simples e o Palm OS Cobalt para dispositivos mais sofisticados (smartphones), apenas a primeira foi lançada comercialmente, provocando uma defasagem no desenvolvimento de aplicativos para a plataforma.

Com o intuito de tentar reerguer a plataforma, o desenvolvimento dessas versões foi paralisado, dando espaço para uma nova versão desta vez baseado em um *kernel* Linux. Assim, surgiu o ACL (Access Linux Plataform), uma alternativa *open source* com suporte às linguagens C++ e Java no mesmo ambiente, bem como a execução de aplicações dos antigos Palm OS.

Embora alguns aparelhos da própria Palm tenham sido lançados utilizando Windows Mobile durante este período, o ACL aparece como uma alternativa promissora para o desenvolvimento de aplicações para dispositivos Palm.

## 2.4 Plataformas e linguagens avaliadas

Considerando as plataformas e suas respectivas linguagens para programação descritas na seção anterior, é possível relacioná-las de acordo com a Tabela 2.2. Nesta tabela é possível perceber que algumas linguagens são utilizadas em poucas ou até mesmo em apenas uma plataforma, como é o caso da OPL, Objective-C, Python, Ruby, Perl e Flash, enquanto as linguagens C++ e Java possuem uma maior abrangência.

Outro fator importante a ser considerado é o propósito para o qual a linguagem foi

Tabela 2.2: Linguagens disponíveis por plataforma

| Linguagem Plataforma | Symbian OS | RIM | Windows Mobile | Linux | iPhone OS | Palm OS |
|---|---|---|---|---|---|---|
| C++ | • | | • | • | | • |
| Basic [2] | • | | • | | | |
| C# | • | | • | | | |
| Java | • | • | • | • | | • |
| OPL | • | | | | | |
| Objective-C | | | | | • | |
| Python | • | | | | | |
| Ruby | • | | | | | |
| Perl | • | | | | | |
| Adobe Flash | • | | | | | |

desenvolvida. Enquanto linguagens como Python, Ruby, Perl e Flash são geralmente utilizadas como linguagens de *script*[1], outras linguagens (compiladas) como C++, Basic, C#, Java, OPL e Objective-C são normalmente utilizadas para interagir com as APIs nativas do sistema operacional.

Por fim, há também a limitação das plataformas disponíveis para teste. Como a interação de uma API multimídia depende do *driver* disponível (da câmera por exemplo) no dispositivo, o uso de emuladores não refletem completamente essa interação. Assim, embora seja possível testar diversos aplicativos em emuladores, no caso da captura de vídeo as funções de mais baixo nível tendem a não funcionar, pois em grande parte dos emuladores é utilizado apenas um vídeo demonstrativo para simular a câmera. Deste modo é necessário utilizar aparelhos reais para os testes, o que acaba limitando as plataformas disponíveis. Logo, o aparelho selecionado para este trabalho é o Motorola Q11[3], o qual disponibiliza as plataformas Windows Mobile e Java.

Portanto, em virtude da abrangência e da disponibilidade das plataformas para testes, as linguagens avaliadas neste trabalho são o C++ (e em parte sua variante Symbian C++) e Java.

## 2.5 APIs Multimídia para Dispositivos Móveis

Uma API visa oferecer uma interface de programação para facilitar o acesso a um determinado grupo de recursos. Neste âmbito, as APIs multimídias em geral integram a comunicação entre o programa em si e os *drivers* de câmera, som e vídeo.

A API a ser empregada é dependente das plataformas suportada pelo dispositivo. Assim, dentre mais reconhecidas e utilizadas podemos citar as APIs C++ *DirectShow* da Microsoft (para o Windows Mobile) e o conjunto de APIs multimídia para plataformas Symbian (o conjunto não possui um nome padrão) e a MMAPI (Mobile Media API) para plataformas Java. A seguir, é mostrado mais especificamente a estrutura dessas APIs.

[1]Linguagem de script: Uma linguagem de script é utilizada para controlar e estender as funcionalidades de um programa. Por serem interpretadas, normalmente não possuem acesso direto à chamadas internas do sistema, fazendo uso das APIs do programa hospedeiro para tal.

[2]Basic: Em Symbian OS é implementada como NS Basic, enquanto que na plataforma Windows Mobile é implementada como Visual Basic.

[3]Especificações do Motorola Q11: http://www.motorola.com/motoinfo/product/details.jsp?globalObjectId=267

### 2.5.1 A API DirectShow

A API DirectShow, produzida pela Microsoft (DIRECTSHOW, 2009), é utilizada para a manipulação de elementos multimídias como imagens, vídeo e som em diversos formatos de arquivos e *streams*. Baseada no *framework* COM[1], a API oferece uma interface comum entre várias linguagens de programação, como C++ e C#.

A API foi construída baseada na ideia de filtros, os quais decompõem a tarefa em diversos passos. As entradas e saídas desses filtros são chamadas de pinos (*pins*), os quais possuem um tipo de dado definido e permitem que o filtro se conecte a outros por meio desses pinos lógicos. Logo, para as várias etapas de uma captura, é possível utilizar diversos filtros em cada etapa. As etapas de um grafo de filtros são as seguintes:

- Captura: os filtros de captura (*source filters*) são responsáveis por prover um fluxo de entrada de dados, seja este proveniente da leitura de um arquivo ou da captura da imagem fornecida pelo *driver* da câmera.

- Transformação: os filtros de transformação (*transform filters*) operam sobre o fluxo de dados proveniente da saída de outros filtros através de seu pino de entrada (*input pin*), redirecionando-o para seu pino de saída (*output pin*). Essas operações podem envolver conversões de formatos ou alteração de dados do vídeo, por exemplo.

- Renderização: os filtros de renderização (*render* ou *sink filters*) representam a última etapa do grafo, realizando tarefas como exibir o vídeo na tela ou escrever os dados em um arquivo.

A Figura 2.2 demonstra as etapas envolvidas na captura multimídia, bem como os tipos de filtros que podem ser utilizados em cada etapa. Assim, por exemplo, caso seja necessário mudar o tipo de codificação do vídeo, basta desconectar o filtro de *encoder* atual e conectar outro que possua os mesmos tipos de pinos de entrada e saída. Uma vantagem de utilizar esse modelo de filtros é a possibilidade de utilizar diversas *threads* para cada etapa, o que tende a aumentar o desempenho do aplicativo e é feito de forma transparente pela API. Outra vantagem dessa arquitetura é a possibilidade de utilizar o mesmo filtro *DirectShow* (compilado como uma biblioteca DLL) em qualquer linguagem que suporte o modelo de objetos COM, o que aumenta a sua portabilidade.

A própria API já oferece alguns filtros por padrão na construção do grafo de filtros e outros que podem ser construídos através desta. Porém, para a plataforma Windows Mobile, muitos desses filtros não são implementados, necessitando assim que o programador possua algum conhecimento do funcionamento interno da API para poder implementar seus próprios filtros, que em geral é considerado uma tarefa complexa.

[1] COM *framework*: O modelo COM da Microsoft representa uma interface binária padrão para a comunicação entre processos e criação de objetos. Assim, um objeto criado no modelo COM pode ser utilizado em diversas linguagens, a qual precisa definir apenas a interface utilizada pelo objeto binário.

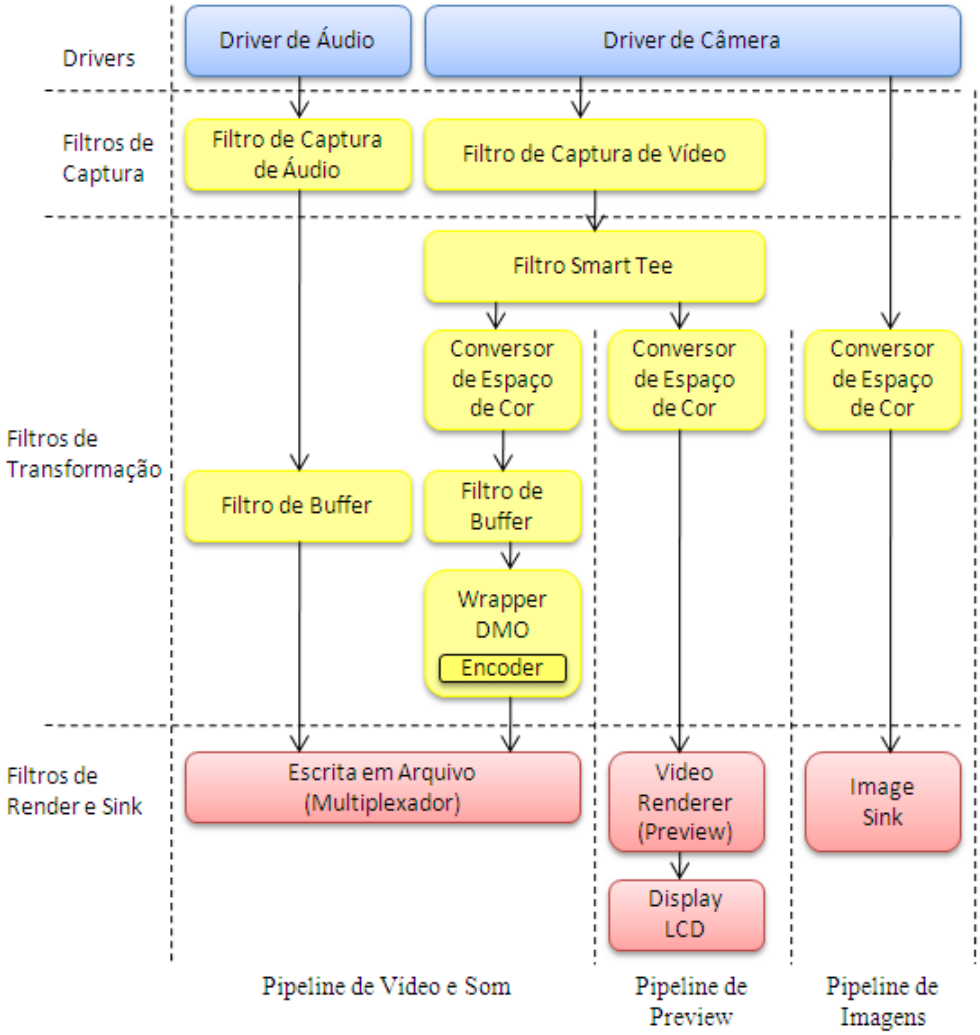


Figura 2.2: Esquema de um grafo de filtros, retirado de (DIRECTSHOW, 2009)

### 2.5.2 A API Symbian

O Symbian OS oferece APIs de alto e baixo nível que auxiliam a captura e manipulação de conteúdos multimídias em dispositivos móveis na sua linguagem nativa, o Symbian C++.

A Figura 2.3 relaciona as APIs de alto e baixo nível disponíveis para o sistema Symbian OS e sua plataforma S60, onde a Camera API é utilizada para a captura de vídeos, a ICL API (Image Conversion Library) para a captura e conversão de imagens e a EXIF API (Exchangeable Image File Format) para a manipulação de metadados.

A API de alto nível (CNewFileServiceClient) é usada sobre um aplicativo cliente através de uma interface, fazendo uso das APIs de baixo nível e capturando imagens ou vídeos diretamente para um arquivo, não oferecendo controle sobre os parâmetros de captura.

Já o conjunto de APIs de baixo nível permite capturar imagens ou vídeos na memória, possibilitando pós-processamento (como *stream* de vídeo) e oferecendo um maior controle sobre parâmetros como a qualidade da imagem e o número de frames em um vídeo. Contudo, é necessário implementar manualmente o relacionamento com uma interface e o tratamento de eventos do aparelho.

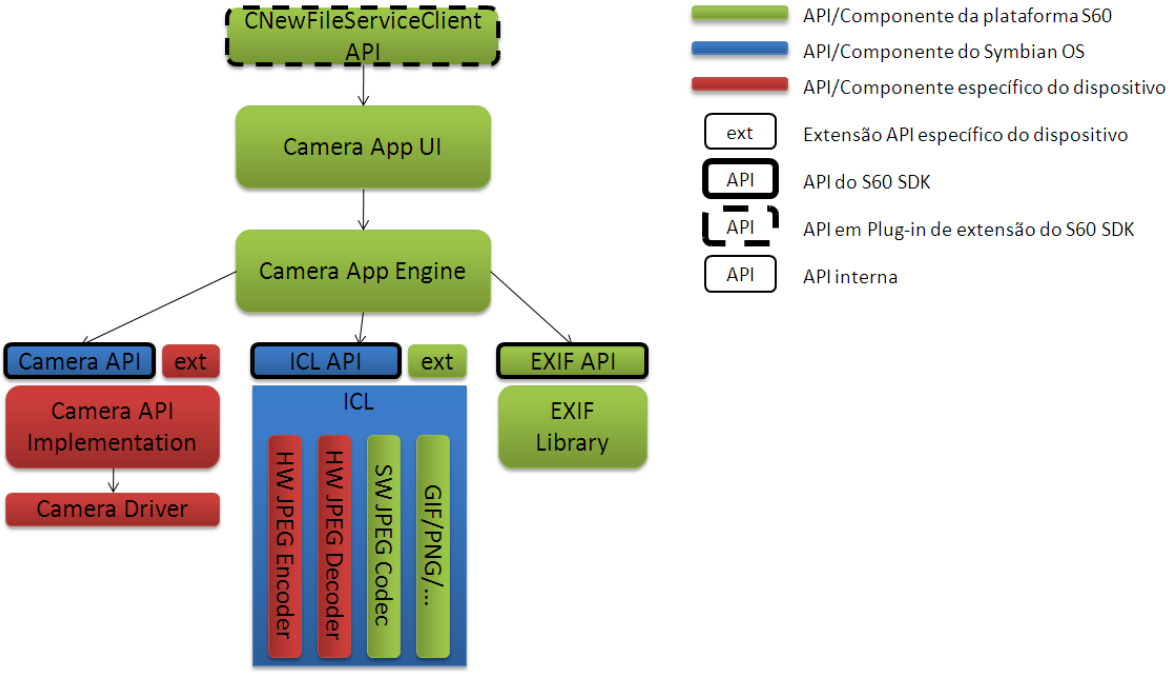


Figura 2.3: APIs multimídias disponíveis no Symbian OS (S60) (NOKIA, 2007).

### 2.5.3 A API Mobile Media (JSR-135)

A MMAPI (Mobile Media API) é utilizada em sistemas JME (Java Micro Edition), a especificação Java para dispositivos móveis. Também conhecida por JSR-135 (Java Specification Requests), a API foi criada para padronizar o acesso a recursos multimídias em móveis, visto que a implementação de suas interfaces (baixo nível) depende do fabricante do aparelho (STEFFEN; CUNHA, 2007). Seguindo a convenção de orientação a objetos de Java, ela oferece quatro classes básicas, segundo (YUAN; SHARP, 2004) relacionados como na Figura 2.4:

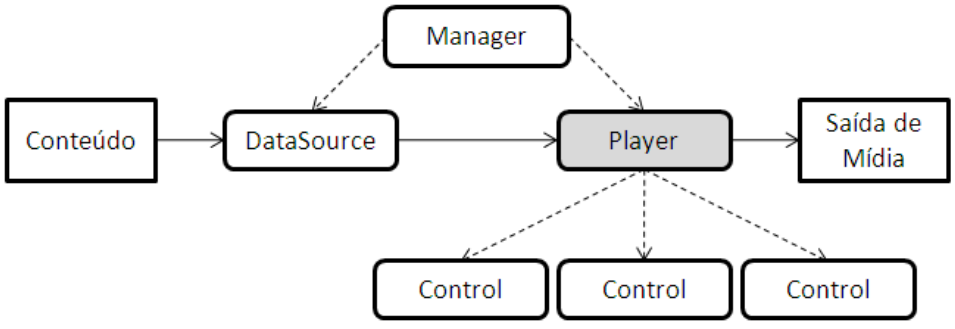


Figura 2.4: Arquitetura MMAPI (YUAN; SHARP, 2004).

- *Manager*: classe do tipo *static factory*, possui a função de instanciar objetos com a interface *Player* através do método *createPlayer()* e ligá-los a uma fonte de dados, a qual pode ser um endereço URI (Uniform Resource Identifier) ou objetos da classe *InputStream* ou *DataSource*. Por ser uma classe estática, não necessita de um construtor.

- *Player*: essa interface, especifica o comportamento padrão a ser adotado por quem implementa a MMAPI para gerenciar diferentes tipos de mídia. Esse comportamento é baseado em um ciclo de cinco estados (Figura 2.5):

  - *Closed*: o objeto *player* libera seus recursos, os quais podem ser acessados por outros objetos da classe ou aplicativos.
  - *Unrealized*: o objeto é apenas instanciado (no *heap*) e não aloca nenhum recurso.
  - *Realized*: o objeto adquire os recursos de mídia (fonte de dados) necessários para sua execução.
  - *Prefetched*: o objeto faz as inicializações necessárias e adquire o controle dos recursos físicos a serem usados, como câmera e microfone.
  - *Started*: o objeto inicia a sua execução.

- PlayerListener: essa interface declara o método *playerUpdate()*, responsável por receber eventos (como estados do objeto ou da mídia) de objetos *Player* que possuam um *PlayerListener* incorporado.

- Control: a interface *Control* permite controlar certos aspectos fornecidos por um *Player*, como volume e vídeo. Contudo, nem todos os fabricantes implementam os controles disponíveis, variando conforme o aparelho.

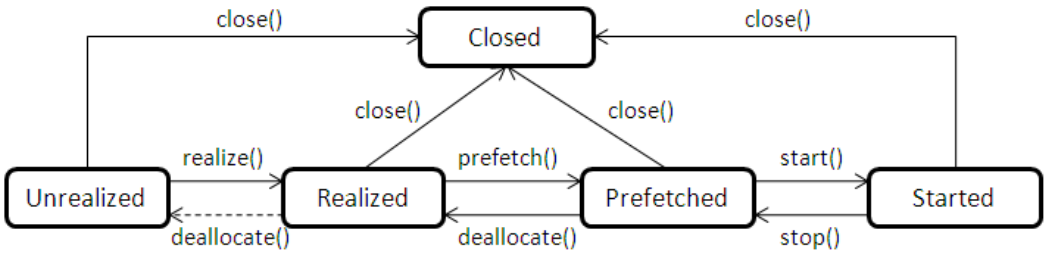


Figura 2.5: Estados e métodos da interface *Player* (YUAN; SHARP, 2004).

O trecho de código abaixo demonstra como as interfaces se relacionam. As interfaces *PlayerListener* e *Control* foram implementadas como *MediaListener* e *VideoControl*, respectivamente.

```
1  class MediaListener implements PlayerListener {
2      public void playerUpdate(Player player, String event,
3                               Object eventData)
4      {
5          //Tratamento de eventos programável
6      }
7  }
8
9  Player mediaPlayer;
10 MediaListener pListener;
11 VideoControl vidMediaControl;
12
13 mediaPlayer = Manager.createPlayer("capture://video");
14 mediaPlayer.addPlayerListener(pListener);
15
16 vidMediaControl = (VideoControl)mediaPlayer.getControl("VideoControl");
17 vidMediaControl.setDisplaySize(160, 140);
18
19 mediaPlayer.start();
```

Relacionamento das interfaces da MMAPI.

# 3 METODOLOGIAS DE VALIDAÇÃO

Para a validação deste trabalho, é necessário utilizar algumas metodologias que avaliem as características de cada linguagem de programação. Essas características porém, nem sempre se apresentam de forma quantitativa (tempo de execução, uso de memória, consumo de energia, entre outros) como normalmente é de se esperar. Há diversos outros fatores qualitativos referentes ao projeto da linguagem e a facilidade de utilizá-la. Por outro lado, analisando a forma como as linguagens utilizam e gerenciam recursos computacionais, é possível ter uma base de como ela irá se comportar em determinados contextos. Assim, essa seção começa descrevendo as principais diferenças no projeto das linguagens C++ e Java e a seguir, segue uma breve descrição das características mais importantes.

## 3.1 O Projeto das Linguagens

Entender como uma linguagem utiliza os recursos computacionais disponíveis é um passo importante para saber como utilizá-la eficientemente e conhecer o domínio dos problemas no qual ela pode ser empregada naturalmente.

A primeira grande diferença perceptível no projeto das linguagens avaliadas (C++ e Java) é a forma na qual elas são implementadas, vide as Figuras 3.1 e 3.2. C++, por ser uma linguagem compilada, transforma o código fonte em linguagem de máquina através de um compilador, o que em geral oferece um maior desempenho de execução. Já a linguagem Java é tida como híbrida, pois utiliza tanto compilação como interpretação. As primeiras implementações de Java utilizavam um sistema totalmente híbrido, onde o código fonte era compilado em uma forma intermediária chamada *Java bytecode*, o qual é interpretado (de *Java bytecode* para linguagem de máquina) por uma máquina virtual, a JVM (Java Virtual Machine). A vantagem dessa máquina virtual é a portabilidade de código Java entre qualquer máquina que possua um interpretador de *byte code*. Atualmente, Java utiliza o conceito de implementação *Just in Time* (JIT), onde o código em linguagem de máquina é gerado apenas quando o aplicativo invoca o método correspondente, guardando o resultado para chamadas subsequentes e melhorando o desempenho.

Outra diferença importante é a utilização de ponteiros para referenciar áreas de memória. Presentes em C++, os ponteiros são referências tratadas como números (endereços de memória), no qual é possível apontar objetos e funções. Utilizar esse recurso melhora o desempenho em certas ocasiões, onde é muito mais rápido copiar a referência para um objeto do que alocá-lo novamente, e também facilita a implementação de certas tarefas, como referenciar funções de *callback*, utilizado para tratar funções como parâmetros de chamada ou retorno. Por outro lado, os ponteiros também permitem referenciar áreas de memória não alocadas ou dados que não são pertinentes à execução, fato que geralmente

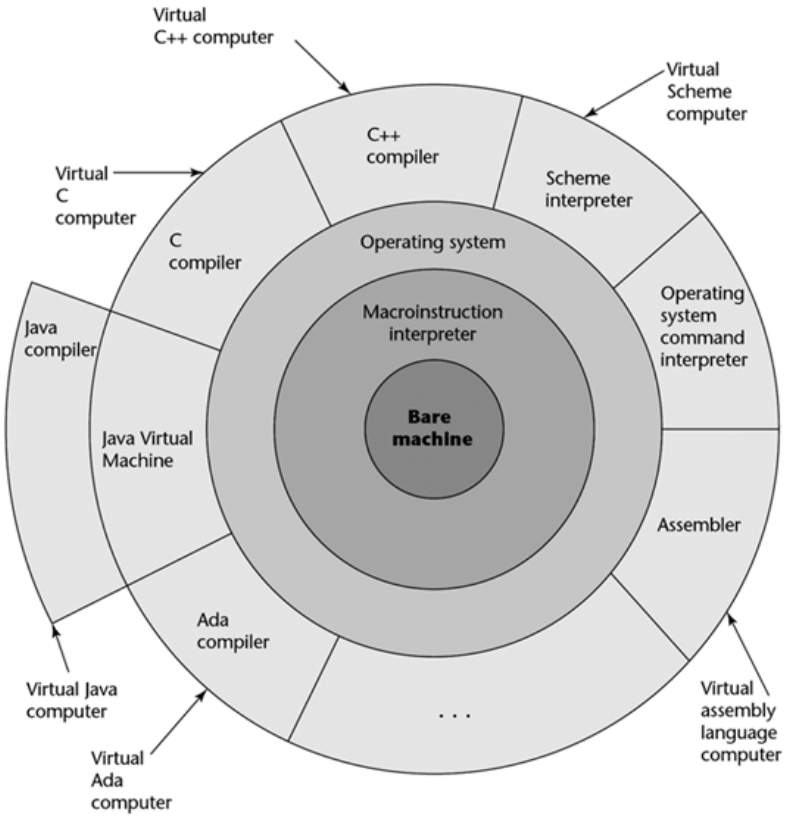


Figura 3.1: Níveis das linguagens de programação (SEBESTA, 2008).

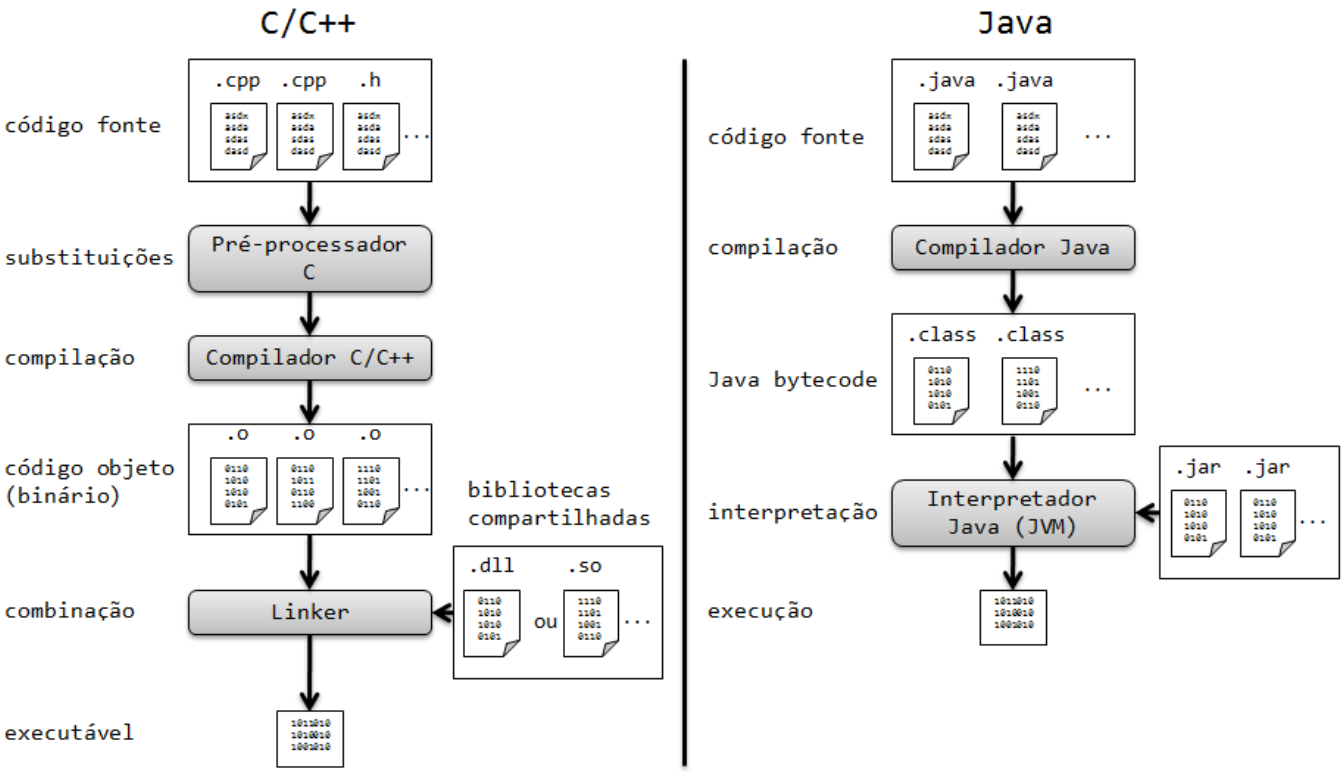


Figura 3.2: Etapas da execução de código fonte em C++ e Java (criado pelo autor).

ocasiona erros de execução e que deve ser evitado obrigatoriamente pelo programador, uma vez que tais erros não são detectados em tempo de compilação. Em virtude disso, Java abandonou a utilização de ponteiros, adotando um modelo de referência (também presente em C++), a qual não pode ser tratada como um número (endereço).

Considerando então a ausência de ponteiros explícitos em Java, advém a impossibilidade de passar parâmetros por referência diretamente. Enquanto C++ permite que valores sejam passados tanto por valor[1], quanto por referência[2], Java utiliza apenas passagem por valor. Embora utilize referências para passar objetos, ou seja, o endereço deste, essa referência não pode ser alterada no escopo interno de uma função. As figuras 3.3, 3.4 e 3.5 demonstram os modos de passagem de parâmetros de C++ e Java.

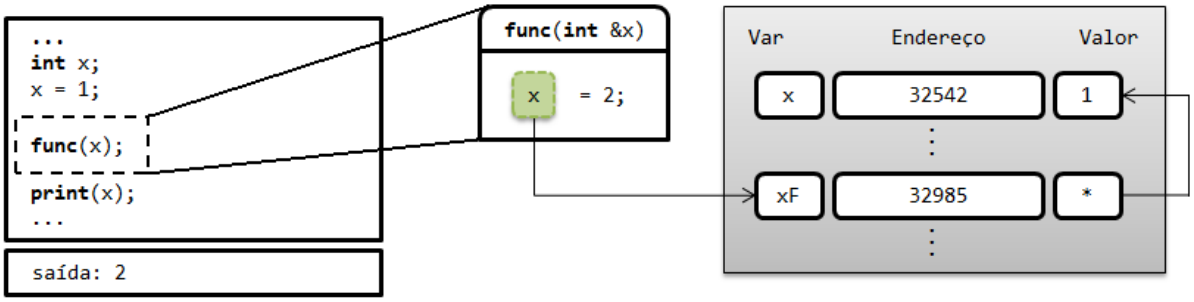


Figura 3.3: Passagem de parâmetros por referência explícita, presente em C++. Neste caso a variável alterada dentro da função retém o seu valor (criado pelo autor).

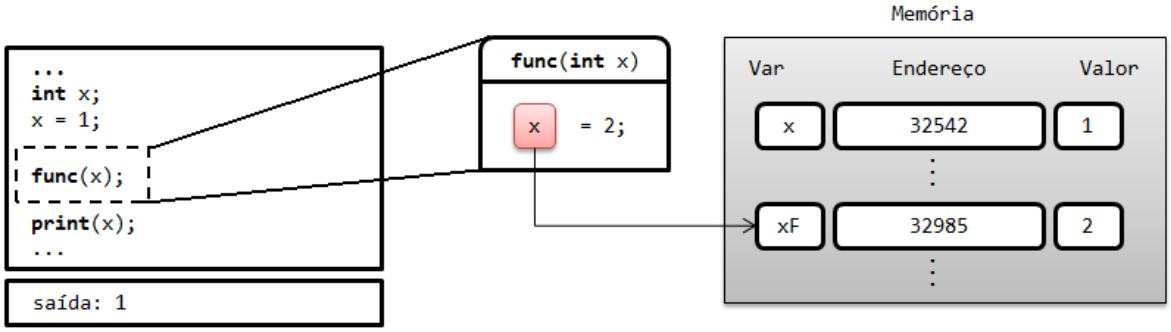


Figura 3.4: Passagem de parâmetros por valor, presente em C++ e Java. Deste modo, a variável retém o seu valor apenas no escopo da função (criado pelo autor).

Mais diferenças surgem quando se trata de como essas linguagens gerenciam a memória. Em C++ fica a encargo do programador alocar e desalocar espaço através dos operadores ***new*** e ***delete***, que por descuido ou má utilização podem ocasionar vazamentos (perda da referência à memória alocada, impossibilitando seu reuso). Já em Java é utilizado um processo de *garbage collection*, responsável por localizar áreas sem referência e desalocá-las, evitando a necessidade de se utilizar o operador ***delete***. Contudo, esse

[1]Passagem por valor: Quando um parâmetro é passado por valor, uma cópia local (temporária) é criada, a qual pode ser alterada dentro de uma função sem modificar o parâmetro original.

[2]Passagem por referência: Na passagem por referência, o endereço do parâmetro é utilizado para acessar a área de memória representada por este, alterando assim o seu valor, o que consequentemente pode ser feito dentro da chamada de uma função.

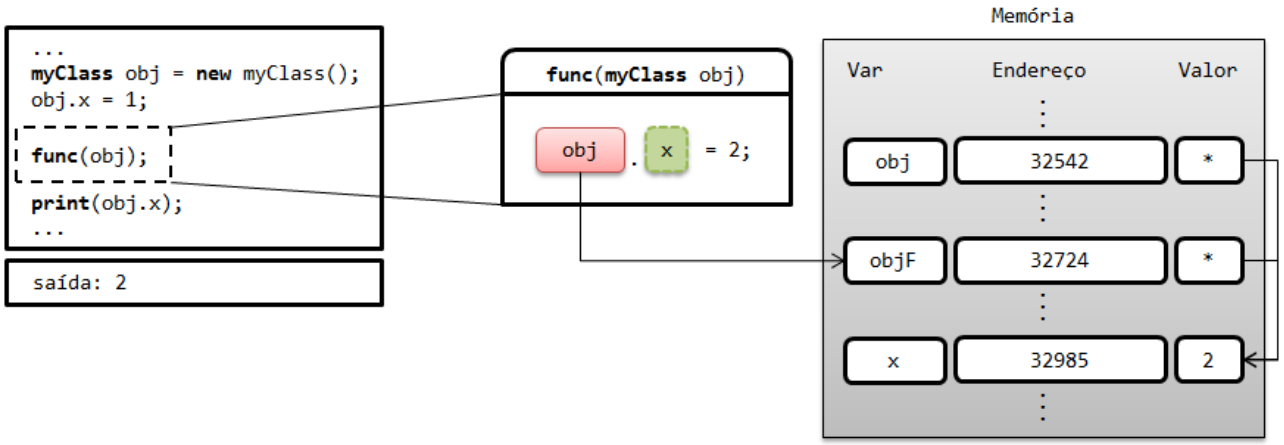


Figura 3.5: Passagem de parâmetros por referência implícita, presente em Java. Neste caso, os atributos do objeto podem ser alterados em qualquer escopo, ao contrário da sua referência, que não sofre alterações dentro da função.(criado pelo autor).

processo reduz o desempenho geral da aplicação, que de tempos em tempos precisa varrer a memória em busca de tais vazamentos.

O problema do gerenciamento de memória de C++ é resolvido em parte na sua variante Symbian C++. Preocupado com a escassez de recursos dos dispositivos móveis (vide seção 2.1), o Symbian C++ adota algumas convenções de programação procurando otimizar os acessos e evitar usos desnecessários de memória. Um exemplo claro dessa preocupação é a forma de como classes devem ser declaradas, onde o programador deve informar implicitamente quais áreas serão utilizadas, ao contrário de uma aplicação C++ comum que aloca objetos na área de *heap* ou *stack* automaticamente. Assim, os seguintes tipos de classes estão disponíveis:

- Classes T: Classes do tipo T não possuem referências a objetos externos, podendo ser alocadas diretamente na área de stack, além de não necessitarem de um destrutor, pois o espaço utilizado no *stack* é limpo automaticamente quando o objeto perde o escopo.
- Classes C: Classes do tipo C representam classes que alocam memória na área de *heap*, e devem ser obrigatoriamente derivadas da classe interna do Symbian OS CBase, além de necessitarem de um construtor de duas fases[1].
- Classes R: Classes do tipo R são utilizadas para armazenar *handlers*, obrigatoriamente possuindo os métodos *Open* e *Close*, não necessitando de construtores ou destrutores. Um exemplo é a utilização de classes R para acessar arquivos.
- Classes M: Classes M representam classes abstratas e servem para declarar uma interface, não alocando memória.

Existem ainda diversas outras convenções utilizadas para o mesmo propósito (encontradas em (JODE; TURFUS, 2004), mas estão fora do escopo deste trabalho.

Por fim, este trabalho analisa também as diferenças presentes no projeto das APIs utilizadas (descritas na seção 2.5), como estruturas e tipos de dados utilizados.

[2]Construtor de duas fases: Utilizado para garantir que o construtor de um objeto não saia de escopo perdendo a referência a seus objetos internos, o que causaria um vazamento na memória, uma vez que tais objetos não poderiam ser apagados diretamente.

## 3.2 Métricas Qualitativas

De acordo com Sebesta(SEBESTA, 2008), há uma certa dificuldade em se estabelecer critérios para avaliação de uma linguagem, pois muitas vezes esses critérios são subjetivos e até mesmo pessoais. Porém, como o autor sugere, há fatores que, independentemente de quem avalia, são considerados importantes para diferenciar as características entre linguagens. A Tabela 3.1 a seguir exemplifica esses fatores, também citados por Dershem (DERSHEM; JIPPING, 1990).

Tabela 3.1: Fatores que influenciam os critérios de avaliação de uma linguagem. (SEBESTA, 2008).

| Característica | Legibilidade | Capacidade de Escrita | Confiabilidade |
|---|---|---|---|
| Simplicidade/ortogonalidade | • | • | • |
| Estruturas de controle | • | • | • |
| Tipos de dados e estruturas | • | • | • |
| Projeto da sintaxe | • | • | • |
| Suporte para abstração | | • | • |
| Expressividade | | • | • |
| Verificação de tipos | | | • |
| Manipulação de exceções | | | • |
| Aliasing[1] restrito | | | • |

Além dessas características inerentes ao projeto da linguagem, há também fatores externos que devem ser considerados, como apontados por Watt (WATT, 2004). Dentre os critérios citados pelo autor no entanto, serão considerados os mais relevantes ao contexto deste trabalho, como portabilidade, nível de programação e familiaridade.

### 3.2.1 Legibilidade

Um dos fatores mais importantes para se avaliar uma linguagem de programação é a facilidade com a qual o seu código pode ser lido e entendido. Isso se deve ao fato de que a escrita do código em si passou a ter um papel secundário, pois, com o conceito de ciclo de vida de um software, a manutenção passou a ser reconhecida como uma parte mais importante, especialmente em termos de custo.

A legibilidade de uma linguagem deve ser considerada no contexto para a qual ela foi projetada, ou seja, um programa feito em uma linguagem que não foi projetada para seu uso, tende a ser mais enrolado e antinatural, consequentemente, mais difícil de ser lido.

#### *3.2.1.1 Simplicidade*

A simplicidade de uma linguagem influencia fortemente sua legibilidade. Isso se deve ao fato de que uma linguagem com muitos componentes é mais difícil de ser aprendida. Sendo assim, é comum os programadores aprenderem apenas uma parte dela, ignorando outros recursos que poderiam ser usados para facilitar a programação ou torná-la mais eficiente.

Há ainda outros fatores que podem complicar a linguagem, como multiplicidade de recursos e sobrecarga (*overloading*). A multiplicidade de recursos se refere ao fato de haver mais de uma maneira de realizar a mesma operação, como por exemplo incrementar uma variável simples em C:

[1] Aliasing: determinação de um nome alternativo ou apelido.

count = count + 1

count += 1

count++

++count

Já a sobrecarga permite que um único símbolo tenha mais de um significado, como por exemplo utilizar + para a adição de números inteiros e números em ponto flutuante. Neste caso, a sobrecarga ajuda a simplificar a linguagem, pois reduz o número de operadores e representa operações de consenso comum. Porém, caso seja utilizada de forma não criteriosa (como sobrecarregar o símbolo + para realizar comparação entre dois vetores) a linguagem acaba tornando-se confusa e difícil de ler.

Contudo, uma linguagem muito simples também pode ser difícil de ler, como é o caso da linguagem *assembly* que, embora seja constituída de comandos simples, falta-lhe estruturas de controle mais complexas, aumentando o número de instruções necessárias.

#### *3.2.1.2 Ortogonalidade*

A ortogonalidade de uma linguagem está relacionada com a capacidade de combinar suas estruturas primitivas para criar as estruturas de controle e de dados da linguagem.

De modo geral, a ortogonalidade está relacionada com a simplicidade do projeto, pois quanto mais ortogonal, menos exceções as regras existirão, tornando a linguagem mais fácil de ser aprendida, lida e entendida. Porém, muita ortogonalidade pode causar problemas, visto que há um grande número de possibilidades de se combinar estruturas. Deste modo, é mais fácil cometer erros semânticos na escrita que não são acusados pelo compilador.

#### *3.2.1.3 Estruturas de Controle*

A preocupação com estruturas de controle surgiu quando muitas linguagens ofereciam comandos do estilo *goto* para realizar iterações. O uso deste tipo de comando prejudica a legibilidade de uma linguagem, uma vez que a ordem de execução e leitura passa a ser não sequencial. Contudo, a maioria das linguagens atualmente implementam estruturas de controles mais robustas, eliminando a necessidade de se utilizar instruções do estilo *goto*. Deste modo, a estrutura de controle de uma linguagem é um fator menos relevante atualmente, dependendo das práticas adotadas pelo programador.

#### *3.2.1.4 Tipos de Dados e Estruturas*

Outro fator que auxilia a legibilidade de uma linguagem é a presença de opções adequadas para definir tipos e estruturas de dados. Por exemplo, a linguagem C utiliza uma representação numérica para o tipo booleano:

var_inicializada = 1

o que não expressa um significado claro, enquanto que na linguagem Java pode-se utilizar

var_inicializada = **true**

cujo significado é muito mais claro.

Do mesmo modo, linguagens que oferecem tipos de dados mais robustos como registros (*record*) ou classes, tendem a ser mais legíveis do que as que utilizam estruturas básicas como *arrays*.

#### *3.2.1.5 Projeto da Sintaxe*

A forma dos elementos disponíveis para se escrever um programa tem um efeito significativo na legibilidade. Como exemplo pode-se citar:

- *Palavras reservadas.* O uso de palavras reservadas geralmente gera conflito entre a simplicidade e a legibilidade da linguagem. Por exemplo, em C utiliza-se chaves para delimitar tanto um bloco de seleção como blocos de laço, o que simplifica a linguagem. Por outro lado, Ada utiliza mais palavras reservadas mas com uma legibilidade maior, como **end if** para blocos de seleção e **end loop** para blocos de laço. Outra questão importante é se palavras reservadas podem ser usadas como nomes para variáveis do programa, o que pode deixar o código confuso.

- *Forma e significado.* Projetar instruções que indiquem, pelo menos parcialmente, a sua finalidade, auxiliam na legibilidade do código. Normalmente essa característica é violada por linguagens que usam instruções idênticas que variam semanticamente de acordo com o contexto utilizado. Por exemplo, na linguagem C, o significado da palavra reservada **static** depende de onde ela aparece. Se for utilizada na definição de uma variável dentro de uma instrução, significa que a variável é criada durante a compilação do programa (*compile time*). Porém, se a mesma instrução for utilizada fora de todas as funções, significa que a variável é visível somente no arquivo no qual ela foi declarada.

### 3.2.2 Capacidade de Escrita

A capacidade de escrita representa a facilidade com a qual uma linguagem pode ser usada para um determinado domínio de problema. A maioria das características que afetem a legibilidade, também afetam a capacidade de escrita, pois escrever um programa exige frequentemente uma releitura do código já feito pelo programador.

#### *3.2.2.1 Simplicidade e Ortogonalidade*

Se uma linguagem possui um grande número de diferentes construções, alguns programadores podem não estar familiarizado com todas elas. Em decorrência disso, alguns recursos podem ser utilizados de forma errônea, não eficientemente, ou até mesmo não utilizados. Em casos mais extremos, recursos desconhecidos podem ser utilizados acidentalmente, gerando resultados inesperados e que não são detectados pelo compilador. Logo, um número menor de construções primitivas, juntamente com um conjunto consistente de regras (ortogonalidade) é melhor do que simplesmente possuir um grande número de primitivas. Assim, após dominar um conjunto simples de construções primitivas e o modo como elas interagem, um programador pode projetar soluções complexas mais facilmente.

#### *3.2.2.2 Suporte para Abstração*

A capacidade de abstração de uma linguagem permite definir e utilizar estruturas ou operações complexas de modo a ignorar muitos dos detalhes pertinentes à implementação. A abstração é um conceito fundamental nas linguagens de programação mais modernas, reflexo das metodologias de projeto de programas. Logo, a capacidade de escrita da linguagem é fortemente influenciada pela sua capacidade de abstração, pois expressões complexas podem ser escritas com uma maior naturalidade. Existem dois tipos de abstração: processos e dados.

Um exemplo simples de abstração de processo é o uso de bibliotecas para implementar um algoritmo que é utilizado várias vezes em um programa. Assim, ao invés do mesmo código ser replicado diversas vezes ao longo do programa, basta que ele referencie o subprograma da biblioteca, abstraindo os detalhes de sua implementação e facilitando a escrita.

Já a abstração de dados refere-se à capacidade de se utilizar estruturas adequadas para representá-los, como é o caso das classes adotadas em linguagens mais modernas, que torna o uso de objetos muito mais intuitivo.

#### *3.2.2.3 Expressividade*

A expressividade de uma linguagem está ligada à quantidade de computação que é possível realizar utilizando suas instruções. Comumente, a expressividade é utilizada para redefinir computações de uma maneira mais simples. Em C++, por exemplo, utilizar a notação *count++* é mais conveniente e mais breve do que *count = count + 1*. Logo, a expressividade tende a aumentar a capacidade de escrita de uma linguagem.

### 3.2.3 Confiabilidade

Diz-se que um programa é confiável quando ele se comporta de acordo com as suas especificações em todas as condições. A seguir, é descrito alguns recursos de linguagens de programação que influenciam significativamente a confiabilidade de programas. Vale ressaltar que a confiabilidade abrange o conceito de execução do programa, e não da segurança contra ataques maliciosos, que em geral depende da técnica de programação adotada pelo programador.

#### *3.2.3.1 Verificação de Tipos*

A verificação de tipos representa um papel importante na confiabilidade de uma linguagem. Como essa verificação é uma tarefa dispendiosa, é mais desejável que seja executada em tempo de compilação.

Uma linguagem com tipagem fraca é mais suscetível à erros em tempo de execução, o que reduz a sua confiabilidade. Ao exemplo da linguagem C++, é possível utilizar *arrays* que não fazem parte do seu tipo, logo não são verificados pelo compilador. Isso permite que áreas fora de seu limite sejam acessadas, o que frequentemente ocasiona erros que podem não aparecer instantaneamente, sendo assim, difícil de detectar. Ao contrário de C++, a linguagem Java exige que todos os acessos aos *arrays* estejam dentro da sua faixa de tamanho, aumentando a confiabilidade do código.

#### *3.2.3.2 Manipulação de Exceções*

A capacidade de uma linguagem interceptar e tratar erros em tempo de execução é um grande auxílio para a confiabilidade. Esse recurso, presente nas linguagens mais modernas como C++ e Java, visa impedir que um programa termine de forma inesperada ou funcione fora da sua especificação em condições incomuns, recuperando um estado estável da execução.

#### *3.2.3.3 Aliasing*

Por definição, *aliasing* é a capacidade de dois métodos ou nomes distintos referenciarem a mesma posição da memória. Esse recurso pode ser perigoso se utilizado de forma não criteriosa. Ao exemplo dos ponteiros (*pointers*) em C++, é possível que duas variá-

veis diferentes apontem para a mesma célula de memória, o que pode facilmente fugir do controle do programador e ocasionar erros difíceis de detectar. Já outras linguagens, como Java, restringem fortemente o *aliasing* a fim de aumentar a confiabilidade.

#### *3.2.3.4 Legibilidade e Capacidade de Escrita*

Tanto a legibilidade quanto a capacidade de escrita influenciam a confiabilidade de uma linguagem. Por exemplo, um programa escrito em uma linguagem com uma má representação do seu domínio, não irá suportar maneiras naturais de escreve-lo e, quanto menos natural for a implementação do código, mais suscetível a erros ele será.

### 3.2.4 Portabilidade

A portabilidade de uma linguagem se refere à possibilidade de utilizar o mesmo código em plataformas diferentes. Quanto menos mudanças forem necessárias, melhor é a sua portabilidade. Esse conceito se torna relativamente importante se tratando de dispositivos móveis, pois como descrito em 2.1, a diversidade de plataformas existentes impõe uma limitação ao desenvolvimento de aplicativos em determinados sistemas.

### 3.2.5 Nível de Programação e Familiaridade

O nível de programação de uma linguagem é importante em termos de abstração. Linguagens baixo nível geralmente obrigam o programador a pensar de uma maneira menos abrangente, visto a necessidade de se preocupar com o uso de ponteiros e *bits*, por exemplo. Logo, esse tipo de abordagem geralmente é mais propenso a erros, porém pode ser necessário em algumas partes do desenvolvimento. Por outro lado, linguagens de alto nível geralmente são orientadas a utilizar as abstrações inerentes a aplicação, o que torna a programação mais natural e intuitiva.

A familiaridade do programador com uma linguagem pode ser vista como um fator de custo relevante em um projeto, pois nem sempre a adoção de uma nova linguagem ou API justifica os custos necessários para treinar uma equipe.

### 3.2.6 Custo Final

O custo final de uma linguagem de programação envolve muitas de suas características que são pesadas diferentemente a partir de perspectivas diversas. Os desenvolvedores da linguagem normalmente estão preocupados em como implementar suas construções e recursos que serão oferecidos. Os usuários por usa vez, estão preocupados primeiramente com sua capacidade de escrita e depois com a sua legibilidade. Por fim, os projetistas provavelmente irão enfatizar a elegância e a capacidade de generalização da linguagem. Assim, pode-se listar alguns custos que podem influenciar e diferenciar fortemente uma linguagem.

Em primeiro lugar, há o custo do treinamento de programadores para utilizar a linguagem, onde a familiaridade, simplicidade e a ortogonalidade são fatores importantes, pois linguagens mais poderosas geralmente são mais difíceis de aprender.

Em segundo lugar, existe o custo para escrever programas na linguagem, refletido diretamente pela capacidade de escrita e pelo domínio das soluções oferecidas.

Em terceiro, vem o custo para compilar a linguagem. Com um poder computacional cada vez maior esse custo tende a ser menos importante. Contudo, no contexto de dispositivos móveis é um fator considerável, uma vez que além de compilar o aplicativo, é necessário também instalá-lo em outra plataforma (em geral de *desktop* para móvel),

operação que nem sempre pode ser feita diretamente.

Em quarto lugar, o custo de execução de programas é fortemente influenciado pelo projeto da linguagem utilizada. Por exemplo, se ela exigir muitos testes de tipos durante o tempo de execução, haverá uma grande redução na performance do programa, independente do compilador. Contudo, a eficiência de execução comumente é trocada por outros fatores, como simplicidade e portabilidade, no caso de Java.

O quinto fator é o custo do sistema de implementação da linguagem. Um dos fatores que explicam a rápida expansão e aceitação de Java é que compiladores e interpretadores estavam a disposição logo após o lançamento do seu projeto, além de que diversas plataformas móveis o suportam, ao contrário de C++ que é (inclusive as ferramentas) específico para cada sistema.

O sexto fator é a confiabilidade da linguagem. Se um sistema é considerado crítico, como o controle de uma usina nuclear ou um satélite, esse custo pode ser considerado como o mais importante. No contexto de móveis porém, a confiabilidade se torna importante em aplicativos comerciais, como operações em bolsas de valores, onde uma falha pode causar prejuízos financeiros.

Por fim, há o custo de manutenção da linguagem, seja para corrigir ou adicionar novas funcionalidades. O custo da manutenção depende principalmente da legibilidade, uma vez que essa tarefa geralmente é realizada por pessoas que não são autoras originais do código. Neste caso, uma legibilidade ruim pode tornar a tarefa extremamente desafiadora e complexa.

Obviamente, há um grande número disponível de critérios para avaliar linguagens de programação, como generalidade (ampla faixa de domínios) e boa definição (perfeição e precisão da definição da linguagem). Contudo, todos esses critérios não são precisamente definidos, nem exatamente mensuráveis, logo este trabalho atem-se em analisar os conceitos mais relevantes no contexto de dispositivos móveis.

## 3.3 Métricas Quantitativas

Existem diversos meios de medir o desempenho e a qualidade de uma aplicação. Embora muitas vezes esses métodos sejam estatísticos ou até mesmo pessoais, em geral eles oferecem uma boa visão sobre o uso de uma determinada linguagem em um dado contexto. Assim, em um primeiro momento é feito uma análise sobre os aspectos quantitativos do código escrito e, em seguida, sobre o desempenho dos aplicativos em execução.

### 3.3.1 Métricas de Código

A análise de um código fonte em geral está relacionada com a capacidade do programador em criar estruturas adequadas para o problema. Contudo, como este trabalho visa analisar comparativamente implementações que dependem fortemente das APIs disponíveis, não há uma grande flexibilidade em termos de se utilizar outras estruturas se não as fornecidas pelas próprias APIs.

Outra consideração importante é o fato de que relacionar métricas de códigos fonte de linguagens diferentes pode ser algo incoerente. Por exemplo, comparar código Java com Assembly em geral traz resultados inconclusivos, visto que as linguagens possuem paradigmas, aplicações e sintaxes diferentes. Contudo, como as linguagens avaliadas foram utilizadas para o mesmo propósito (manipulação de vídeo em celulares), possuem sintaxes semelhantes e suportam o paradigma de orientação a objeto, esta comparação pode ser vista como uma demonstração da complexidade envolvida ao aplicá-las ao mesmo

contexto. Em virtude disso, pode-se considerar esta análise como um adendo à análise qualitativa, pois mesmo representando uma base estatística para tal, seu objetivo é fornecer uma visão geral sobre os projetos implementados. Para isso, serão utilizadas as métricas descritas a seguir, conforme a definição RSM (Resource Standard Metrics) (MSQUARED, 2009).

Uma das métricas mais comuns e mais utilizadas para dimensionar um projeto é o número de linhas (LOC ou *Lines of Code*). Embora seja uma medida importante, o seu valor muitas vezes é arbitrário, pois nem toda linha presente no código resulta em uma instrução computável. Para cobrir essa falha, dois outros conceitos são utilizados, o número de linhas efetivas (eLOC ou *Effective Lines of Code*) e o número de linhas lógicas (lLOC ou *Logical Lines of Code*). O número de linhas efetivas desconsidera comentários, linhas em branco e parênteses ou colchetes isolados, que não representam um trabalho real feito pelo programador. Já o número de linhas lógicas considera apenas declarações de código terminadas por ponto e vírgula, representando instruções que geram computação. O exemplo abaixo demonstra os critérios utilizados por cada convenção.

```
Código Fonte                    LOC  eLOC  lLOC  Comentário  Espaço Branco
if (x<10) //teste de limite      *     *             *
{                                *
   //atualiza y                                      *
                                                                    *
   y = x + 1;                    *     *     *
}                                *
```

Critérios de contagem das métricas.

Métricas de função, por sua vez, procuram determinar o grau de modularidade de um sistema, bem como a qualidade geral do código desenvolvido. Uma função com muitas linhas de código, por exemplo, tende a ser difícil de ler e manter, assim como muitos parâmetros de entradas a torna difícil de usar e suscetível a erros de ordenamento de parâmetros. O mesmo acontece com os pontos de retorno que, sendo muitos, dificultam a rastreabilidade de erros em tempo de execução. Assim, o conceito de complexidade de interface de uma função é definido pela composição da quantidade de parâmetros com a quantidade de pontos de retorno.

Em seguida, analisando a quantidade de desvios lógicos e laços de iteração em uma função, tem-se uma ideia sobre a sua complexidade algorítmica e sobre os casos de testes necessários. Esse conceito, nomeado complexidade ciclomática, é definido pela presença dos comandos **for**, **while**, **if**, **case** e **goto**, indicando por exemplo, que códigos com muitos laços e desvios e poucos casos de testes são mais propensos a erros e efeitos colaterais indesejáveis. Embora o conceito de função venha do paradigma de programação procedural, essas métricas podem ser aplicadas sobre métodos, o equivalente a função no paradigma orientado a objetos.

Por fim, métricas de classes focam a análise para o conceito de orientação a objetos. Em geral, um sistema grande com poucas classes, ou uma classe com muitos métodos e atributos, indicam que estas possuem baixa coesão, ou seja, são utilizadas para propósitos diferentes das quais foram inicialmente planejadas, dificultando sua leitura e seu reuso.

### 3.3.2 Métricas de Desempenho

Métricas de desempenho, por sua vez, oferecem uma visão sobre o uso dos recursos computacionais do aparelho. Embora a capacidade dos computadores avance rapidamente, o controle desses recursos ainda desempenha um papel importante no caso dos

dispositivos móveis pois, conforme descrito na seção 2.1, existem limitações de memória e processamento em virtude do tamanho reduzido e da capacidade energética da bateria. Assim, aplicativos que consomem muitos recursos reduzem drasticamente a vida útil do aparelho, como demonstrado em Kato (KATO; LO, 2007), onde uma bateria com duração de aproximadamente duzentas horas em estado de espera (*stand-by*) é consumida em torno de cinco a seis horas executando aplicativos Java.

O aparelho selecionado para medir o desempenho dos aplicativos propostos é o Motorola Q11 (especificações técnicas no apêndice), por executar o sistema operacional Windows Mobile 6, que suporta tanto aplicativos em C++ como em Java. Porém, essas informações nem sempre estão disponíveis com a facilidade encontrada em *desktops*, pois requerem ferramentas externas capazes de monitorar o aparelho, uma vez que o ambiente mono-tarefa da maioria dos dispositivos móveis impede que essas medidas sejam tomadas enquanto se executa o aplicativo. Logo, as medidas de tempo de execução serão obtidas diretamente pelo código, enquanto que o consumo de memória do programa pode ser obtido pelo gerenciador de tarefas do aparelho. O uso do processador não será considerado nesta análise pelo fato de os processadores da arquitetura ARM (utilizado pelo aparelho) não possuir registradores que permitam tal medida, ao exemplo da arquitetura x86 para *desktops*. Embora seja possível medir o uso através do aplicativo, isso consumiria um tempo de execução adicional diferenciado nas linguagens.

O tempo de execução medido é o necessário para inicializar o aplicativo, uma vez que depois de inicializado o mesmo pode ser executado por tempo indeterminado. Essa medida visa mostrar não só o desempenho das linguagens em si, como o de suas APIs, que utilizam estruturas diferentes. Em C++ o tempo de inicialização é obtido da seguinte forma:

```
1 long int starttime = GetTickCount();
2 // Codigo a ser testado
3 //...
4 long int endtime = GetTickCount();
5 long int time_consumption = endtime - starttime;
```

Obtendo o tempo de execução do aplicativo C++.

No caso de Java, essa medida é obtida através do seguinte código:

```
1 long starttime = System.currentTimeMillis();
2 // Codigo a ser testado
3 // ...
4 long endtime = System.currentTimeMillis();
5 long time_consumption = endtime - starttime;
```

Obtendo o tempo de execução do aplicativo Java.

Embora as chamadas utilizadas para demarcar o instante de início e de fim do código a ser testado também consumam algum tempo para serem executadas, este pode ser ignorado, pois conforme demonstrado em Motorola(MOTOROLA, 2006), o tempo utilizado por essas funções é menor do que um milissegundo, o que está fora dos limites de precisão do aparelho.

Uma última consideração a ser feita sobre essas medidas é em relação à qualidade do vídeo capturado. Essa análise não será considerada neste trabalho, pois envolvem medidas mais complexas sobre processamento de imagens, fugindo do foco deste. Contudo, para efeitos de comparação dos aplicativos implementados basta que o aparelho, a câmera e os meios de transmissão sejam os mesmos.

# 4 IMPLEMENTAÇÃO DO APLICATIVO

Para poder avaliar questões de qualidade e desempenho das linguagens escolhidas neste trabalho, foi proposto um aplicativo capaz de capturar o vídeo diretamente da câmera do aparelho e transmiti-lo por *stream* para um servidor, o qual fica responsável apenas por receber os pacotes de dados do aparelho móvel (Figura 4.1).

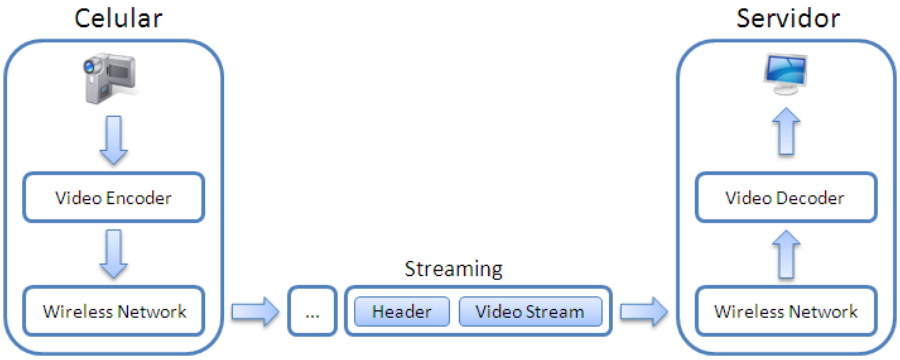


Figura 4.1: Estrutura geral do aplicativo proposto (criado pelo autor).

## 4.1 Descrição do Aplicativo

O objetivo dos aplicativos se resume em acessar o *buffer* (memória) de vídeo, onde os quadros estão sendo armazenados. Uma vez obtido esses dados, os mesmos devem ser enviados através da rede para o servidor por meio de *sockets*.

O meio utilizado para a transmissão escolhido foi o *wireless* (Seção 2.2). Embora o interesse maior em se distribuir vídeo em aparelhos celulares seja a partir da tecnologia 3G, esta é dependente da disponibilidade de planos de uma operadora, enquanto o *wireless* pode ser facilmente utilizado em testes locais. Outro fator que força essa escolha, é que para efeitos de comparação das linguagens o meio de transmissão não é relevante (desde que suporte a quantidade de dados a ser transmitida), pois o mesmo meio foi utilizado para todas as plataformas.

Nas próximas seções é descrito como o aplicativo foi implementado e as ferramentas utilizadas, visando as linguagens C++ (para Windows Mobile) e Java.

## 4.2 Aplicativo C++

Como descrito na seção 2.5.1, a API DirectShow manipula elementos multimídias através de uma estrutura de filtros de grafo, os quais nem todos estão disponíveis na sua

versão *mobile*. Um dos filtros não incorporados no Windows Mobile é o *Sample Grabber*, responsável por capturar os dados da câmera em tempo de execução, uma vez que os filtros de saída (renderização) da API possuem apenas pinos de entrada e só permitem acesso aos dados ao fim da gravação. Desse modo é necessário implementar um filtro próprio, capaz de exercer tal função. Porém, outro problema vem a tona, pois para se implementar um filtro (uma classe) no DirectShow, é necessário herdar algumas características da classe base da API, chamada de *BaseClass*, que só está disponível na SDK paga da versão componentizada Windows CE (vide seção 2.3.3), como mostrado na Figura 4.2. Embora seja possível utilizar a SDK do Windows CE em versão limitada (trial) para este caso, esse tipo de correlação pode aumentar consideravelmente os custos de um projeto, o que nem sempre está claro quando se começa a trabalhar com essa plataforma.

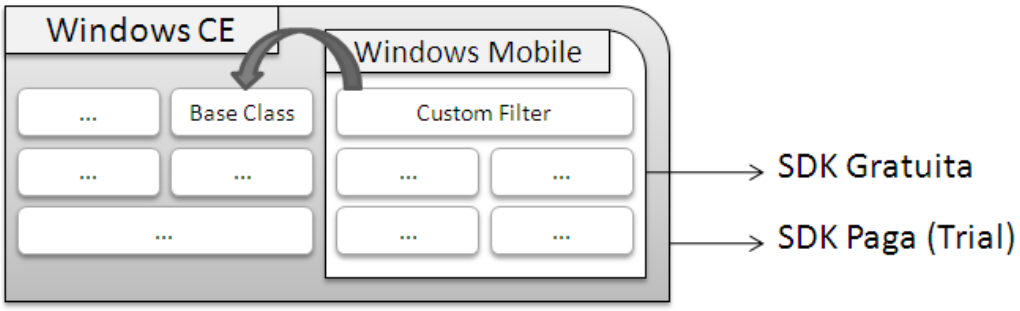


Figura 4.2: Hierarquia de classes da API DirectShow entre Windows CE e Windows Mobile (criado pelo autor).

Outro quesito a ser considerado sobre essa API é o fato de que os parâmetros de ajuste da câmera, como abertura e luminosidade, são dependentes da DLL (Dynamic-Link Library) que o fabricante oferece, as quais nem sempre possuem documentação aberta.

### 4.2.1 Aplicativo e Ferramentas Utilizadas

Em posse das informações sobre o funcionamento e características da API DirectShow na plataforma Windows Mobile, foi necessário ainda buscar as ferramentas a serem utilizadas na implementação do aplicativo:

- Visual Studio 2008[1]: O ambiente de programação Visual Studio (proprietário da Microsoft) facilita as etapas de compilação e implantação do aplicativo no aparelho, enquanto é possível acompanhar a compilação e a depuração do código diretamente pelo *desktop*.

- Microsoft Active Sync: Essa ferramenta (gratuita) é responsável por realizar a interface de comunicação entre o dispositivo e o *desktop*, e por conseguinte, com o ambiente Visual Studio.

- Windows Mobile 6 SDK: Essa SDK (gratuita), necessária para se ter acesso aos componentes do Windows Mobile (WMSDK, 2009), é dividida em duas partes. A primeira, *Standard SDK*, é destinada aos dispositivos com teclado alfanumérico, enquanto que a segunda, *Professional SDK*, é destinada aos dispositivos com tela sensível ao toque. É importante mencionar que essas SDKs não são excludentes,

ou seja, um aplicativo feito a partir da *Professional SDK* não necessariamente irá executar em uma plataforma que utilize a *Standard SDK*.

- Windows Embedded CE: A SDK do Windows CE (paga) foi utilizada em sua versão de amostra (*trial*) para se ter acesso às classes de base necessárias para implementar o filtro de captura da câmera.

Assim, para implementar o aplicativo foi utilizado uma estrutura lógica baseada na Figura 4.3. O primeiro passo é criar uma *thread* responsável por gerenciar o grafo de filtros, garantindo que o programa execute concorrentemente o aplicativo em si, e a captura do vídeo. Essa *thread* possui um modelo de comunicação através de mensagens que sinalizam eventos, assim, em seu laço principal (*main loop*) é executado uma chamada ao método *ProcessCommand*, que lê a mensagem enviada e realiza operações na *thread* internamente. Nos exemplos a seguir, as declarações e tratamento de erros serão omitidos para fins de clareza.

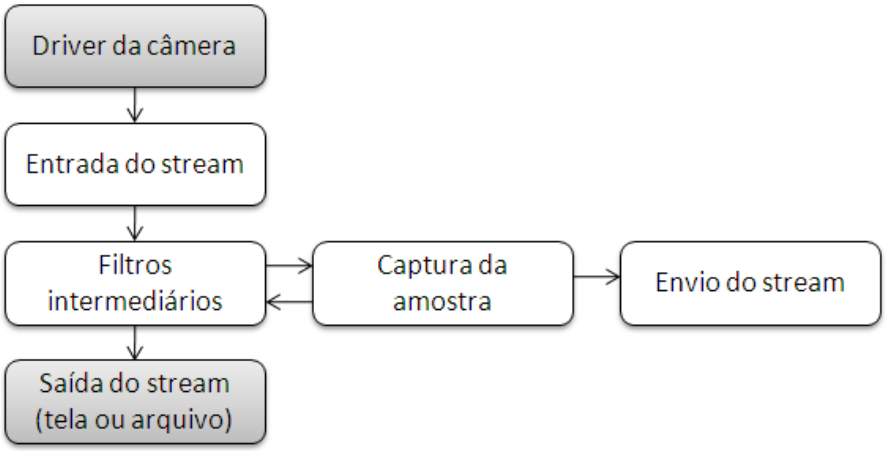


Figura 4.3: Estrutura lógica do aplicativo para a plataforma Windows Mobile (criado pelo autor).

```
1  HRESULT CGraphManager::ProcessCommand() {
2    //...
3    switch( m_currentCommand ) {
4      case COMMAND_BUILDFRAMEGRAPH:
5        hr = CreateCaptureGraphInternal();
6        SetEvent( m_hCommandCompleted );
7      break;
8      //...
9      case COMMAND_STARTFRAMECAPTURE:
10       hr = StartCaptureFramesInternal();
11       SetEvent( m_hCommandCompleted );
12     break;
13
14     case COMMAND_REGISTERCALLBACK:
15       hr = SetGrabberCallbackInternal();
16       SetEvent( m_hCommandCompleted );
17     break;
18
```

[1]Essa e outras ferramentas citadas podem ser encontradas em: http://www.microsoft.com/DownLoads/

```
19      default: break;
20    }
21    //...
22  }
```

Método utilizado para processar mensagens recebidas pela *thread*.

```
1  DWORD WINAPI CGraphManager::ThreadProc( LPVOID lpParameter ) {
2    //...
3    while(( command != COMMAND_SHUTDOWN ) && ( hr != S_FALSE )) {
4      dwReturnValue = WaitForMultipleObjects( 2, pThis->m_handle,
5                                              FALSE, INFINITE );
6      switch( dwReturnValue ) {
7        case WAIT_OBJECT_0:
8          command = pThis->m_currentCommand;
9          hrCommand = pThis->ProcessCommand();
10       break;
11
12       case WAIT_OBJECT_0 + 1:
13         pThis->ProcessDShowEvent();
14       break;
15
16       default: break;
17     }
18   }
19   //...
20 }
```

Laço principal da *thread* que espera por comandos.

Neste caso, a *thread* espera por dois tipos de notificações (*WaitForMultipleObjects*), onde a primeira está relacionada com eventos do aplicativo, e a segunda com eventos do DirectShow. Após as inicializações dos componentes da API do Windows, é necessário definir uma função de *callback* que será chamada sempre que um quadro for capturado, informando o seu endereço na memória para que o aplicativo possa usá-lo independentemente da captura. A função de *callback*, definida como *OnSampleCaptured*, é informada à *thread* de captura através da estrutura de mensagens:

```
1  HRESULT CGraphManager::SetCallback(MANAGED_SAMPLEPROCESSEDPROC callback)
2  {
3    //salva o ponteiro no objeto
4    m_ManagedCallback = callback;
5
6    //processa o registro na thread de captura
7    m_currentCommand = COMMAND_REGISTERCALLBACK;
8    SetEvent(m_handle[0]);
9    WaitForSingleObject(m_hCommandCompleted, INFINITE);
10
11   return S_OK;
12 }
```

Definindo a função de *callback* através de mensagens.

onde o comando atual é definido para COMMAND_REGISTERCALLBACK, o qual será lido pelo método *ProcessCommand* descrito acima. A chamada *SetEvent* sinaliza a *thread* do aplicativo (indicada por m_handle[0]) que há uma nova mensagem a ser processada, forçando-a a sair de seu estado de espera e, em seguida, através da função *Wait-*

*ForSingleObject*, o método espera a *thread* completar sua tarefa, a qual responde através do comando m_hCommandCompleted.

Uma vez informado a função de *callback* responsável por receber o endereço de memória que contem o quadro recém capturado, deve-se criar o filtro de grafos que realizará essa função. Novamente, através da estrutura de mensagens, é informado o comando COMMAND_BUILDFRAMEGRAPH, fazendo com que a *thread* do aplicativo construa o grafo de filtros.

Para construir um grafo de filtros, é utilizado um objeto do tipo *CaptureGraphBuilder*, responsável por ligar os filtros através de seus pinos. Como o DirectShow é baseado no modelo COM, é necessário utilizar algumas funções especiais para criar a instância da classe e definir-lhes uma interface. No primeiro caso, a função *CoCreateInstance* recebe um identificador global CLSID (Class ID), retornando uma instância do objeto para a linguagem C++. Para definir uma interface, ou seja, o que será visível a partir do objeto, é utilizado o método *QueryInterface*, que também recebe um identificador único chamado de IID (Interface Identifier) e o objeto que mostrará essa interface.

```
1  //cria instancia do construtor do grafo
2  m_pCaptureGraphBuilder.CoCreateInstance(CLSID_CaptureGraphBuilder);
3
4  //cria instancia de um grafo de filtros
5  //e define a interface de controle
6  m_pFilterGraph.CoCreateInstance(CLSID_FilterGraph);
7  m_pFilterGraph->QueryInterface(IID_IMediaControl,
8                                 (void**)&m_pMediaControl);
9
10 //adiciona a referencia do grafo de filtros ao construtor
11 m_pCaptureGraphBuilder->SetFiltergraph( m_pFilterGraph );
```

Iniciando a construção do grafo de filtros.

A interface *IID_IMediaControl* expõe os controles do grafo, usado para iniciá-lo ou pará-lo. Como ela foi adicionada ao objeto m_pMediaControl, este agora pode controlá-lo. Após essa sequência de comandos, o grafo de filtros é instanciado, restando agora o passo de criar os filtros e adicioná-los ao grafo. Essa etapa é feita de maneira semelhante, instanciando objetos e suas interfaces através da arquitetura COM. O primeiro filtro criado realiza a tarefa de capturar as imagens fornecidas pelo *driver* da câmera.

```
1  //cria instancia de um filtro de captura
2  //e obtem uma referencia de sua PropertyBag
3  m_pVideoCaptureFilter.CoCreateInstance(CLSID_VideoCapture);
4  m_pVideoCaptureFilter.QueryInterface(&pPropertyBag);
5
6  //obtem a referencia ao driver da camera
7  GetFirstCameraDriver(wzDeviceName);
8  varCamName = wzDeviceName;
9  ...
10 //adiciona a propriedade VCapName com a referencia
11 //ao driver e carrega as propriedades no filtro
12 PropBag.Write(L"VCapName", &varCamName);
13 pPropertyBag->Load(&PropBag, NULL);
14
15 //adiciona o filtro ao grafo
16 m_pFilterGraph->AddFilter(m_pVideoCaptureFilter,
17                           L"VideoCaptureFilterSource");
```

Criando o filtro de captura.

Um passo importante neste filtro, é obter a referência (nome) do *driver* da câmera, para que o mesmo possa receber os quadros capturados. Isso é feito utilizando-se uma *PropertyBag*, um objeto que pode conter propriedades dinâmicas. A vantagem de utilizar *PropertyBags* se deve ao fato de que, caso seja modificada uma propriedade em uma classe, a mudança é refletida apenas neste objeto, evitando que todo o código tenha de ser reestruturado. Assim, a propriedade *"VCapName"* guarda o nome do *driver* da câmera, que em seguida é carregado no filtro através de sua interface de *PropertyBag* pelo método *Load*. Configurado o filtro de captura, este é então adicionado ao grafo através do método *AddFilter*, que recebe como parâmetros o filtro em si, e um nome lógico.

Com o filtro de captura configurado, é possível agora adicionar filtros intermediários, que podem operar sobre o *stream* de dados. Como descrito anteriormente, o filtro necessário para acessar a memória de vídeo (*SampleGrabber*) não é implementado no DirectShow para Windows Mobile, o que força o programador a criar um filtro novo (uma biblioteca DLL). Por ser uma tarefa complexa, o código necessário para criar o filtro muitas vezes é feito de uma maneira automática (através de *templates* da IDE VisualStudio), motivo pelo qual grande parte do seu código não será mostrado neste trabalho, bastando saber que sua principal função é executar o método *Transform*:

```
1  HRESULT CSampleGrabber::Transform(IMediaSample *pMediaSample) {
2    //...
3    //obtem o ponteiro para a area de memoria que contem
4    //o quadro recebido pelo filtro de captura
5    pMediaSample->GetPointer(&pCurrentBits);
6    lSize = pMediaSample->GetSize();
7
8    //invoca a funcao de callback atravez de seu ponteiro
9    //informando o ponteiro para a area de memoria, o tamanho
10   //desta area, e parametros do quadro como tamanho e duracao
11   if ( m_Callback ) {
12     m_Callback( pCurrentBits, lSize, m_Height, m_Width, m_Stride );
13   }
14   //...
15 }
```

Configurando um filtro de transformação.

O filtro, uma vez criado, é compilado em forma de uma biblioteca DLL, a qual precisa ainda ser registrada no sistema operacional do aparelho para que este possua uma referência às suas chamadas. Essa operação é feita automaticamente pela IDE Visual Studio, que compila e registra a DLL diretamente no dispositivo. Feito o registro do filtro como uma DLL, este agora pode ser utilizado por qualquer aplicativo que suporte a API DirectShow, possuindo seu próprio *Class ID* (CLSID).

```
1    CComPtr<IBaseFilter>  pSampleCapFilter;
2    //...
3    //cria a instancia do filtro de transformacao
4    CoCreateInstance(CLSID_SampleGrabber, NULL, CLSCTX_INPROC,
5                     IID_IBaseFilter, (void**)&pSampleCapFilter);
6
7    //adiciona o filtro ao grafo
8    m_pFilterGraph->AddFilter(pSampleCapFilter, L"SampleGrabber");
9
10   //e obtem a interface do filtro de transformacao
```

[1]GraphEdit: disponível em http://msdn.microsoft.com/en-us/library/dd390950%28VS.85%29.aspx

```
11    pSampleCapFilter->QueryInterface( IID_ISampleGrabber,
12                                      (void**)&m_pISampleGrabber ) );
13    //...
```

Adicionando o filtro de transformação que lê os quadros na memória ao grafo.

O último passo a ser realizado é conectar os pinos dos filtros adicionados ao grafo. Essa tarefa é realizada pelo construtor através do método *RenderStream*, que recebe como parâmetro a categoria dos pinos (captura), o tipo do dado recebido ou enviado (video) e três filtros, onde o primeiro representa um filtro de captura (fonte), o segundo um filtro intermediário (transformação) e o terceiro um filtro de renderização que neste caso é nulo, pois não há necessidade de gravar o vídeo em disco, uma vez que o mesmo será enviado pela rede.

```
1    //...
2    //conecta os pinos dos filtros do grafo
3    m_pCaptureGraphBuilder->RenderStream(&PIN_CATEGORY_CAPTURE,
4                                         &MEDIATYPE_Video,
5                                         m_pVideoCaptureFilter,
6                                         pSampleCapFilter,
7                                         NULL);
8    //...
```

Conectando os filtros presentes no grafo.

A próxima etapa, após a construção do grafo, é controlá-lo. Como nos casos anteriores, esse evento é repassado para a *thread* de captura através do comando COMMAND_STARTFRAMECAPTURE, que basicamente invoca o método *StartCaptureFramesInternal*. Lembrando que a interface de controle está instanciada no objeto m_pMedia Control, basta chamar o método *Run* deste para inicializar a captura.

```
1  HRESULT CGraphManager::StartCaptureFramesInternal() {
2    //...
3    //inicia a execução do grafo de filtros
4        m_pMediaControl->Run();
5    //...
6  }
```

Iniciando a execução do grafo de filtros.

Assim, quando os dados chegam no filtro de transformação, este invoca a função de *callback OnSampleCaptured*, recebendo como parâmetros o endereço do quadro na memória, e o seu tamanho em *bytes*.

```
1  VOID VideoClient::OnSampleCaptured(BYTE* pdata, long len)
2  {
3    //...
4    //Envia os dados recebidos por UDP
5    VSSendUDPData(vcSocket, pdata, len);
6    //...
7  }
```

Acessando e enviando os dados na memória.

### 4.2.2 Transmissão Stream

Uma vez acessado diretamente os dados da câmera, ainda é necessário compactar o vídeo e disponibilizá-lo em forma de pacotes de *stream*. O formato de *stream* padrão da

Microsoft é o ASF (seção 2.2). Geralmente utilizado com o protocolo UDP (User Datagram Protocol), esse formato trabalha de forma serial, ou seja, garantindo que os pacotes cheguem ao seu destino em ordem. Embora não garanta que todos os pacotes cheguem ao receptor, em geral um quadro perdido não é suficiente para degradar significativamente a qualidade geral do vídeo.

Mas novamente a plataforma Windows Mobile sofre com os cortes em sua API, a qual não oferece suporte para a criação de pacotes no formato ASF, adicionando ainda mais complexidade ao desenvolvimento nesse ambiente. Embora seja possível criar uma classe para lidar com esse formato de *stream*, essa alternativa acaba se desviando do propósito geral deste trabalho, sendo assim, deixada como sugestão para trabalhos futuros.

Porém, para fins de testes de implementação, é possível enviar os pacotes de vídeo no formato nativo da câmera (RGB) diretamente, sem informações de ordem e resolução, para o servidor utilizando a API Winsock (*sockets)* e o protocolo UDP. O servidor, em seguida, pode rodar o vídeo uma vez informado sua resolução e formato manualmente. Por não se tratar do assunto principal deste trabalho e por haver pouca diferença entre o uso da API Winsock em celulares e *desktops*, sua implementação não será demonstrada.

Para execução, os primeiros testes foram realizados no smartphone Motorola Q11. A execução nesse aparelho apresentou problemas no momento em que o programa solicita acesso a câmera. O erro ocorre na chamada pPropertyBag->Load(&PropBag, NULL), onde a pPropertyBag é utilizado para informar ao filtro de captura a identificação do *driver* da câmera, sendo que, quando chamado esse método, o aparelho em questão permanece processando, sem retornar qualquer mensagem de erro. Por se tratar de um método interno à API (Load), não foi possível depurar em mais detalhes esse problema de execução. Em busca realizada em fóruns de discussão especializados, também não houve maiores informações para solucionar o problema.

Não sendo possível a execução nesse aparelho, que era o que estava disponível no momento, como alternativa, este trabalho utilizou o emulador específico do aparelho fornecido pela Motorola[1], até conseguir outro celular para realizar os testes. A desvantagem do uso desse simulador é que o mesmo não consegue acessar a *webcam* do computador. Dessa forma, ele apenas simula a câmera, escrevendo na memória de vídeo imagens em tons de cinza que variam do branco ao preto. A Figura 4.4 ilustra o aplicativo C++ em execução no simulador.

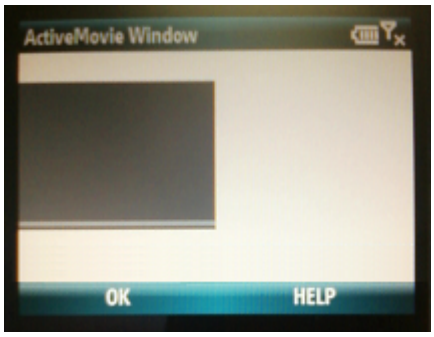


Figura 4.4: Aplicativo C++ em execução.

[1]Disponível em: http://developer.motorola.com/docstools/windows-mobile-plugins/MOTO_Q11_plugin.msi/

## 4.3 Aplicativo Java

A plataforma Java possui uma versão voltada especialmente para dispositivos móveis, a JME, a qual disponibiliza uma API exclusiva para a manipulação de elementos multimídias nesses dispositivos, a MMAPI (Mobile Media API). Essa API de alto nível oferece as interfaces necessárias para a manipulação dos recursos em um aparelho (como câmera, som e tela) através de chamadas internas à funções de mais baixo nível implementadas em geral pelos fabricantes.

### 4.3.1 Aplicativo e Ferramentas Utilizadas

As ferramentas disponíveis para a plataforma Java são, em sua grande maioria, gratuitas, o que facilita a sua disseminação e a criação de aplicativos por terceiros. Para implementar a solução em Java, as seguintes ferramentas foram utilizadas:

- Java JDK 6 Update 16: O JDK (Java Development Kit) contém os principais programas e ferramentas necessários para programar aplicativos em Java. Conta também com o JRE (Java Runtime Environment), responsável por executar código Java.
- Java ME SDK 3.0: O SDK (Software Development Kit)Java voltada para dispositivos móveis contém as ferramentas adicionais necessárias para o desenvolvimento nesses aparelhos.
- NetBeans IDE 6.7: Essa IDE (Integrated Development Environment, gratuito) oferece um ambiente de programação para a plataforma Java, facilitando a implementação e permitindo algumas correções do código em tempo de escrita.

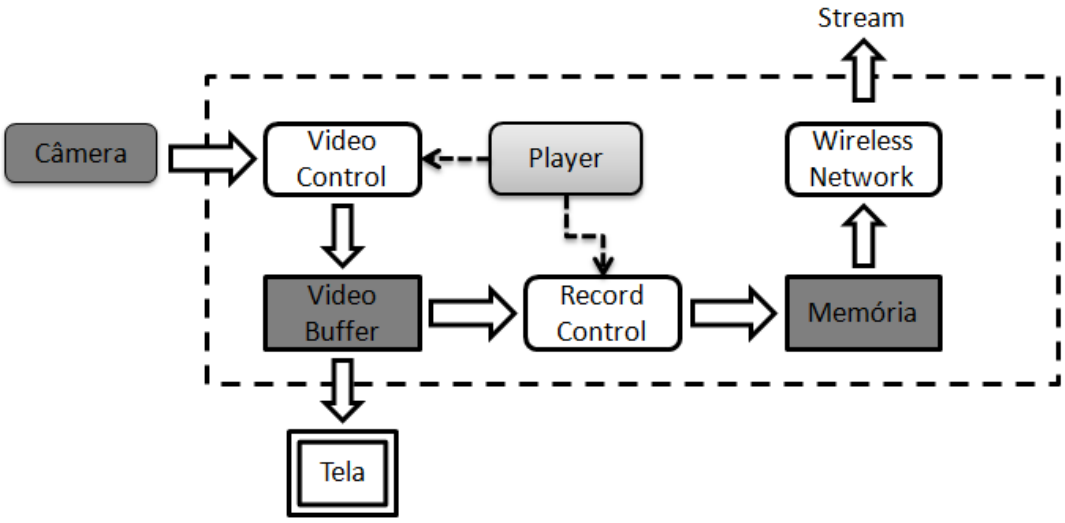


Figura 4.5: Estrutura do aplicativo em Java (criado pelo autor).

A Figura 4.5 demonstra a estrutura do aplicativo proposto. O objeto *Player* utiliza, através da interface *Control*, os controles de vídeo (*VideoControl*) e de gravação (*RecordControl*). O primeiro, é utilizado para obter a imagem da câmera e mostrá-la na tela do aparelho, enquanto que o segundo grava os dados obtidos pelo controle de vídeo. Neste momento, o *Player* precisa ser interrompido para se ter acesso aos dados obtidos pelo controle de gravação, os quais em seguidas são enviados pela rede para o servidor.

Como Java segue o paradigma de orientação a objetos, a sua API fornece uma classe especial para aplicativos voltados aos dispositivos móveis, chamada de MIDlet. A classe MIDlet visa otimizar o uso dos recursos do aparelho rodando sobre uma versão mais leve da máquina virtual Java (JVM) a KVM (SUN, 2008). Assim, o primeiro passo para implementar o aplicativo é criar uma classe que estenda a classe MIDlet. Nesta classe é possível adicionar classes formulários (*forms*), que são responsáveis por exibir elementos como imagens, textos, campos de entrada e saída, botões, entre outros. Logo, outra classe foi implementada para conter esses elementos, estendendo a classe *Form*. Para que este formulário possa receber os comandos externos (teclas), esta classe implementa também a interface *CommandListener*, que através do método *commandAction* permite receber tais comandos, pertencentes à classe *Command*. Nos exemplos a seguir, as declarações e tratamento de erros serão omitidos para fins de clareza.

```
1  public class CaptureForm extends Form implements CommandListener {
2    //...
3    //construtor da classe
4    public CaptureForm(String name, VideoClient parent) {
5      super(name);
6     //...
7    }
8
9    //metodo para tratar os comandos recebidos
10   public void commandAction(Command c, Displayable d) {
11
12     if (c == CMD_EXIT) {
13       //...
14       parentMidlet.destroyApp(true);
15       parentMidlet.notifyDestroyed();
16     }
17
18     else if (c == CMD_Capture) {
19       //...
20       recording = new RecordCamera(myPlayer);
21       recording.start();
22     }
23
24     else if (c == CMD_STOP) {
25       //...
26       recording.StopRecord();
27     }
28   }
29 }
```

Classe para exibir elementos e receber comandos.

Portanto, ao receber o comando CMD_Capture, é criado um objeto da classe *RecordCamera*, responsável por obter os quadros da câmera. Por questões de desempenho e escalabilidade, essa classe é executada como uma *thread*, estendendo então a classe homônima.

Como explicado na seção 2.5.3, essa API possui quatro classes básicas que são gerenciadas através de estados pelo *Player*. Essa abordagem facilita a abstração ao implementar o aplicativo, mas por outro lado, ela impede que os dados que estão sendo manipulados em um estado ativo sejam acessados. Assim, para poder ler o *buffer* (classe *ByteArrayOutputStream*) de imagens da câmera, por exemplo, é necessário interromper o estado de execução do *Player*, realizar as operações sobre o *buffer*, e iniciá-lo novamente. Obvia-

mente essa é uma operação onerosa, mas que demonstra claramente a falta de capacidade da linguagem Java em operar em níveis inferiores.

```
1  class RecordCamera extends Thread {
2
3    RecordControl rc;
4    Player rcPlayer;
5
6   //construtor da classe
7    public RecordCamera(Player p) {
8      rcPlayer = p;
9    }
10
11  //inicia a thread
12   public void run() {
13     RecordVideo();
14   }
15
16   //executa a captura dos quadros
17   public void RecordVideo() {
18    //...
19     try {
20      //obtem o controle de gravacao do Player
21       rc = (RecordControl) rcPlayer.getControl("RecordControl");
22
23    //define o buffer de saida da imagem
24    //e inicia a captura
25       output = new ByteArrayOutputStream();
26
27       rc.setRecordStream(output);
28       rc.startRecord();
29
30    //captura os quadros durante o tempo
31       Thread.sleep(200);
32
33    //encerra a captura para obter acesso
34    //ao buffer de imagens
35       rc.stopRecord();
36       rc.commit();
37
38    //inicia o envio do conteudo do buffer
39    //atraves da classe ConnectionClient
40       connection.setBuffer(output.toByteArray());
41       connection.setSendData(true);
42     }
43     catch (Exception e) {
44       e.printStackTrace();
45     }
46   }
47 }
```

Classe para capturar os quadros da câmera.

Por fim, a classe *ConnectionClient*, implementada neste trabalho, realiza a tarefa de enviar o conteúdo do *buffer* pela rede. Como esta não faz parte do objetivo principal deste trabalho, e por ser implementada de maneira semelhante em *desktops*, seu código não será mostrado. A Figura 4.6 demonstra o aplicativo Java em execução.

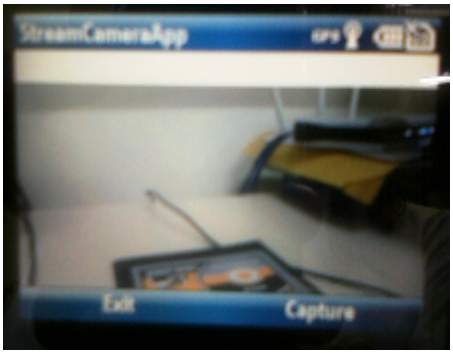


Figura 4.6: Aplicativo Java em execução.

### 4.3.2 Transmissão Stream

O modelo de rede utilizado para a transmissão do vídeo é semelhante à do aplicativo em C++, o qual envia o sinal para um servidor através do protocolo UDP. A diferença básica neste caso, reside no formato do *stream* a ser utilizado. Uma vez que em Java não é possível realizar um *stream* de vídeo diretamente da memória (como é feito em C++), a alternativa que mais se aproxima da solução proposta é um envio sequencial de imagens ou ainda de diversos vídeos pequenos. Assim, o formato de mídia utilizado depende das capacidades do aparelho, sendo que neste caso foi utilizado a codificação MPEG-4 do formato 3GP (descritos na seção 2.2).

# 5 TESTES E VALIDAÇÃO

A primeira questão a ser levantada ao analisar as soluções implementadas é se elas são comparáveis. Como foi descrito na seção anterior, a lógica das implementações não é a mesma, pois as plataformas e APIs disponíveis não oferecem uma solução para a captura de vídeo diretamente. Assim, foi necessário utilizar lógicas diferentes em cada aplicativo. Porém, como o objetivo final é o mesmo (capturar e transmitir vídeo por *stream*), esse fato pode ser considerado como uma análise adicional, referente a disponibilidade e facilidade de se utilizar os recursos presentes para um contexto que a princípio não foi considerado no projeto dessas plataformas.

A seguir, utilizando-se as métricas descritas no capítulo 3, é feito uma análise em cima das soluções implementadas.

## 5.1 Análise Qualitativa

A análise qualitativa foi realizada em cima dos aplicativos implementados e, mais especificamente, nas estruturas e funções que as APIs utilizam, visando o contexto deste trabalho. Obviamente essa análise pode ser diferente para outros contextos, visto que ela depende da relação entre o objetivo para o qual a linguagem foi projetada e o contexto no qual ela é utilizada.

### 5.1.1 Legibilidade

A começar pela simplicidade, a principal diferença perceptível no projeto das linguagens é o fato de que Java não implementa alguns recursos presentes em C++, como herança múltipla de classes e ponteiros.

Partindo para a análise das soluções implementadas, podemos citar como exemplo os métodos utilizados em Java e C++ para iniciar a captura de imagens da câmera. No primeiro código, escrito em Java, é utilizado três componentes (objetos) básicos (RecordControl, Player e ByteArrayOutputStream), enquanto que no segundo código, escrito em C++, são necessários sete componentes (VideoControl, StreamConfig, MediaControl, FilterGraph, FilterState, SampleGrabber e MediaTypeCallback) para iniciar a câmera. Isso se deve principalmente ao fato da arquitetura das APIs utilizadas, visto que as interfaces em Java mascaram chamadas em baixo nível internas do sistema, enquanto que em C++ é necessário construir a lógica em baixo nível através do grafo DirectShow (capítulo 4).

```
1  public void RecordVideo() {
2      try {
3          rc = (RecordControl) myPlayer.getControl("RecordControl");
4          if (rc == null) {
5              return;
6          }
7          output = new ByteArrayOutputStream();
8
9          rc.setRecordStream(output);
10         rc.startRecord();
11     } catch (Exception e) {
12         e.printStackTrace();
13     }
14 }
```

Iniciando a captura de imagens em Java.

Outro fator que reduz a legibilidade do código em C++ é o *overloading* de alguns identificadores, neste caso o **void**. Se utilizado como o tipo de uma função, indica que a mesma não retorna um valor. Caso seja utilizado como parâmetro de uma função (linha 1), indica que a função não recebe valores de entrada. Por último, se for utilizado na declaração de um ponteiro (como o *cast*[1] do endereço de pVideoWindow para **void**** na linha 16), indica que o ponteiro é universal, ou seja, pode apontar para qualquer variável (salvo se esta for do tipo **const**), inclusive funções.

```
1  HRESULT CGraphManager::StartCaptureFramesInternal(void)
2  {
3    HRESULT hr = S_OK;
4    CComPtr<IVideoWindow>  pVideoWindow;
5    CComPtr<IAMStreamConfig> pConfig;
6
7    if ( m_pMediaControl == NULL )
8    {
9      return E_FAIL;
10   }
11
12   try {
13     m_pMediaControl->Run();
14
15     m_pFilterGraph->QueryInterface( IID_IVideoWindow,
16                                     (void**)&pVideoWindow );
17
18     pVideoWindow->put_Owner( (OAHWND)m_hwnd );
19     pVideoWindow->put_WindowStyle( WS_CHILD|WS_CLIPSIBLINGS );
20     pVideoWindow->put_Visible(OATRUE);
21
22     OAFilterState filterstate;
23     m_pMediaControl->GetState(100,&filterstate);
24
25     if ( filterstate == State_Running ) {
26       if ( !m_pISampleGrabber )
27         return E_FAIL;
28
29       CSampleGrabber* pCodec = (CSampleGrabber*)m_pISampleGrabber.p;
30
```

[1]Cast: Mudança do tipo de uma variável. Neste caso, o endereço de pVideoWindow é convertido para um ponteiro de dupla indireção (ponteiro do ponteiro) que pode apontar variáveis de qualquer tipo.

```
31          int height = pCodec->m_Height;
32          int width = pCodec->m_Width;
33          int size = pCodec->m_SampleSize;
34          int stride = pCodec->m_Stride;
35
36          if ( m_MediatypeCallback ) {
37            m_MediatypeCallback(height, width, size, stride);
38          }
39        }
40        return hr;
41      }
42      catch ( ... ) {
43        hr = E_FAIL;
44      }
45
46      return hr;
47    }
```

Iniciando a captura de imagens em C++.

Ao analisar a ortogonalidade, é possível novamente observar diferenças nas linguagens em questão. Enquanto que em Java normalmente é mais difícil recombinar tipos semanticamente errados, a presença de ponteiros em C++ pode causar efeitos indesejáveis que não são acusados pelo compilador.

Já ao analisar o modo em que as APIs tratam a ortogonalidade, é necessário ter um cuidado especial com os tipos utilizados em C++. Por exemplo, o método *RenderStream*, que conecta os nodos de um grafo DirectShow possui a seguinte assinatura:

```
1  HRESULT RenderStream( const GUID*   pCategory,
2                        const GUID*   pType,
3                        IUnknown*     pSource,
4                        IBaseFilter* pIntermediate,
5                        IBaseFilter* pSink);
```

Assinatura do método **RenderStream**.

Observando os dois primeiros parâmetros, um ponteiro que identifica a categoria do filtro (pCategory) e um ponteiro que identifica o tipo do pino de saída (pType), é visto que ambos possuem o mesmo tipo GUID (Globally Unique Identifier), porem representam valores semânticos diferentes. O mesmo ocorre nos dois últimos parâmetros, que referenciam o filtro intermediário e o filtro de *sink* (final). Embora sejam filtros implementados de forma diferente, eles derivam da mesma classe base IBaseFilter, a qual é usada como parâmetro do método.

Assim, por descuido ou falta de conhecimento, é possível escrever a chamada de diversas maneiras. A primeira chamada mostrada abaixo, demonstra um dos possíveis meios corretos de se utilizar o método.

```
1  m_pCaptureGraphBuilder->RenderStream( &PIN_CATEGORY_PREVIEW,
2                                        &MEDIATYPE_Video,
3                                        m_pVideoCaptureFilter,
4                                        pSampleCapFilter,
5                                        NULL );
```

Método para conectar os nodos em um grafo DirectShow (certo).

Contudo, caso a chamada seja escrita da forma abaixo, nenhum erro é acusado pelo compilador, que verifica apenas os tipos utilizados na chamada do método. Desta forma,

o problema pode acabar sendo mascarado, tornando o erro mais difícil de ser detectado e prejudicando a legibilidade da linguagem.

```
1  m_pCaptureGraphBuilder->RenderStream( &MEDIATYPE_Video,
2                                        &PIN_CATEGORY_PREVIEW,
3                                        m_pVideoCaptureFilter,
4                                        NULL,
5                                        pSampleCapFilter );
```

Método para conectar os nodos em um grafo DirectShow (errado).

Partindo para as estruturas de controle, a diferença básica entre as linguagens é a presença do comando ***goto*** em C++. Embora seja uma palavra reservada em Java, o comando não é implementado, mas é possível ainda utilizar *labels* para definir pontos de saída de laços:

```
1  for (i = 0; i < N; i++) {
2      for (j = 0; j < N; j++) {
3          if (mat[i][j] == value) {
4              found = true;
5              break outer;
6          }
7      }
8  }
9  //ponto de saída
10 outer:
```

Definindo um ponto de saída através de *labels* em Java.

Embora esse comando facilite algumas operações, como definir um ponto de *clean up* na saída de uma função, seu uso em geral prejudica a legibilidade da linguagem, uma vez que a leitura pode deixar de ser sequencial. Logo, ambas as linguagens oferecem estruturas de controle mais rebuscadas, como laços iterativos e suporte a exceções. Contudo, vale ressaltar que, embora essa diferença exista, a mesma não teve impacto na implementação dos aplicativos de testes, pois ambas utilizam laços iterativos para realizar desvios mais controlados e de fácil entendimento.

No caso de tipos de dados e estruturas, ambas as linguagens suportam construções semelhantes, como classes. Mas novamente visando simplificar a linguagem em relação a C++, Java não possui o conceito de herança múltipla (grande parte devido ao problema do diamante, descrito no Apêndice 6), o que em geral melhora a sua legibilidade mas pode complicar a escrita caso seja necessário obter uma estrutura semelhante. Em contrapartida, o uso de herança múltipla em C++ ajuda a agregar funcionalidades em uma classe, vide o exemplo dos filtros criados com a API DirectShow.

```
1  class CSampleGrabber : public CTransInPlaceFilter,
2                         public ISampleGrabber
```

Herança múltipla na criação de filtros em C++.

No exemplo acima, é mostrado o caso do filtro de captura utilizado para obter os quadros do vídeo ao longo do grafo. Permitindo a herança múltipla das classes **CTransInPlaceFilter**, que define um filtro básico com uma entrada e uma saída, e a classe interface **ISampleGrabber**, que fornece os métodos necessários para acessar o *buffer* de quadros, é possível obter uma classe única e concisa, que neste caso não interfere fortemente na legibilidade.

Outra diferença na representação de dados entre as linguagens é o uso de ponteiros em C++. Ponteiros oferecem um grande poder de expressão, pois, como descrito na seção 3.1,

permitem manipular endereços de memória diretamente. Porém, no contexto de legibilidade, seu uso pode a dificultar a leitura do código, como no exemplo abaixo demonstrado por Henricson (HENRICSON; NYQUIST, 1992), onde se faz o uso de ponteiros para apontar funções:

```
1  void (*signal(int, void (*)(int)))(int);
```

Ponteiros para funções em C++.

o que representa uma função (signal) que possui como argumentos um inteiro (**int**) e um ponteiro sem tipo definido (**void**(*)(**int**)) e que retorna o ponteiro para uma função que possui como parâmetro um inteiro (ultimo **int**) e, que por fim, não retorna nenhum valor (**void**).

Finalmente, considerando a sintaxe no caso da legibilidade, praticamente não há diferenças entre as palavras reservadas, visto que Java foi fortemente baseado em C++. Contudo, se tratando de forma e significado, a linguagem C++ tem sua capacidade de leitura reduzida pelo fato de utilizar identificadores sensíveis ao contexto, como o **void** descrito anteriormente e o **static** que, de forma semelhante, tem o seu significado alterado dependendo de onde for declarado (seção 3.2.1.5).

Assim, o gráfico 5.1 mostra a relação dos fatores discutidos nesta seção em relação à legibilidade, onde quanto mais afastado do centro, mais o fator auxilia a legibilidade.

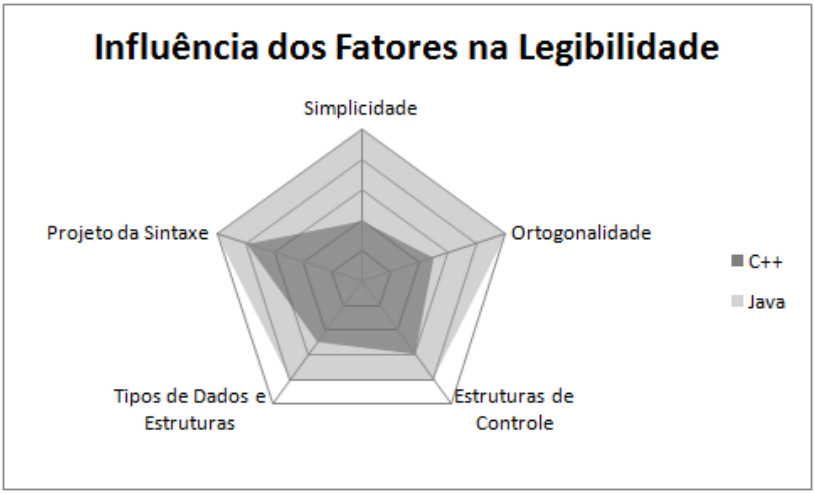


Figura 5.1: Gráfico radar da análise da legibilidade.

### 5.1.2 Capacidade de escrita

Passando para a análise da capacidade de escrita, o primeiro conceito analisado é a simplicidade aliada à ortogonalidade. Embora as linguagens ofereçam estruturas semelhantes, a falta de operações com ponteiros em Java, aplicado ao contexto deste trabalho, tende a dificultar o acesso e a configuração dos recursos em baixo nível. Esse fato acaba gerando a necessidade de se utilizar meios alternativos para realizar essas operações (quando não inviáveis), o que requer uma quantidade maior de estruturas. Contudo, vale ressaltar que a reduzida capacidade de escrita de Java (neste caso) é recompensada pela sua maior capacidade de leitura e pela maior confiabilidade das operações que não manipulam endereços diretamente (simplicidade). Por outro lado, C++ reduz a sua simplicidade em prol da ortogonalidade, fato representado na API utilizada que, através da estrutura de grafo de filtros, torna possível realizar diversas combinações e ter acesso a

uma vasta gama de configurações (como descrito na seção 2.5.1). Mesmo no caso dos dispositivos móveis, onde alguns filtros não são incorporados diretamente, a ortogonalidade da linguagem e da API permitem implementá-los.

O segundo conceito, suporte para abstração, é dividido entre abstração de processo e de dados. No primeiro caso, as linguagens utilizam meios externos e internos para atingir essa abstração. Como meio externo, ambas suportam o conceito de bibliotecas compartilhadas, permitindo que vários programas utilizem um mesmo código compartilhado, ao exemplo das DLLs (Dynamic Link Library) para o ambiente Windows e SO (Shared Object) para plataformas Unix. Como meio interno de abstração de processo, C++ e Java utilizam maneiras diferentes, em virtude do forte paradigma de orientação a objetos de Java em contraste com a possibilidade de utilizar construções procedurais em C++. Deste modo, em C++ é possível declarar procedimentos (ou sub-rotinas) independentes do contexto de uma classe, os quais podem ser compartilhados através de uma declaração presente em arquivos de cabeçalho (*headers*). No caso de Java, a linguagem impõe que esses procedimentos sejam implementados através de métodos de classes, que são gerenciadas por pacotes (*packages*) e visam criar um grupo lógico de código computável. Java permite ainda que tais pacotes sejam reunidos em um único arquivo JAR (Java Archive), o que facilita a sua distribuição e reuso por outros programas. C++ aproxima esse conceito de pacotes através de espaços de nomes (*namespaces*) que, embora sirvam para separar logicamente as classes e funções de uma biblioteca, não podem ser compostos em um arquivo único tal qual o JAR. Logo, Java pode ter uma maior capacidade de escrita neste ponto, pois a utilização de arquivos JAR é independente de plataforma, ao contrário das bibliotecas compartilhadas largamente utilizadas com a linguagem C++.

Por outro lado, a falta de ponteiros em Java, num primeiro instante, parece prejudicar a sua capacidade de escrita, uma vez que não é possível passar um parâmetro por referência (lembrando que Java possui apenas passagem por valor, como ressaltado na seção 3.1). Contudo, se considerarmos que os objetos em Java são passados internamente por referência, e que a linguagem é fortemente orientada a objetos, a falta de ponteiros não gera grande impacto em termos de abstração, pois seria de grande uso em tipos primitivos para poder alterar o seu valor dentro do escopo de uma função, fato que é contornado em Java incluindo esses parâmetros no contexto de uma classe em forma de atributos, os quais devidamente referenciados pelo objeto, podem ter o seu valor alterado em qualquer escopo.

No caso da abstração referente aos dados, o contexto é semelhante ao caso de tipos de dados e estruturas descrito na seção anterior, onde ambas as linguagens possuem construções semelhantes (orientadas a objetos), salvo para casos mais específicos como herança múltipla. Contudo, em termos de capacidade de escrita, é visto que C++ leva vantagem nesse quesito, pois pelo mesmo exemplo demonstrado anteriormente, em que uma classe herda características de mais de uma classe base, seria necessário mais linhas de código em Java para se ter o mesmo efeito obtido em C++.

Por fim, o conceito de expressividade traduz a quantidade de computação que é possível realizar utilizando as suas instruções. Ao analisar, por exemplo, os dois primeiros códigos na seção 5.1.1, que iniciam a captura de imagens da câmera (em Java e C++ respectivamente), pode-se dizer que a expressividade de Java é maior, porém, deve-se atentar ao fato de que as APIs utilizam estruturas diferentes para realizar essa operação e, conforme foi descrito anteriormente, C++ oferece diversos meios de configurações, as quais no caso de Java nem sempre estão disponíveis, pois dependem tanto da especificação comum JSR[1] (*Java Specification Request*) como da implementação de cada fabricante. As-

sim, C++ ao utilizar mais recursos, oferece também mais flexibilidade. Sintaticamente, não há grandes divergências na expressividade, visto que boa parte da sintaxe de java é derivada de C++.

O gráfico 5.2 demonstra a relação dos fatores discutidos nesta seção em relação à redigibilidade, onde quanto mais afastado do centro, mais o fator auxilia a capacidade de escrita.

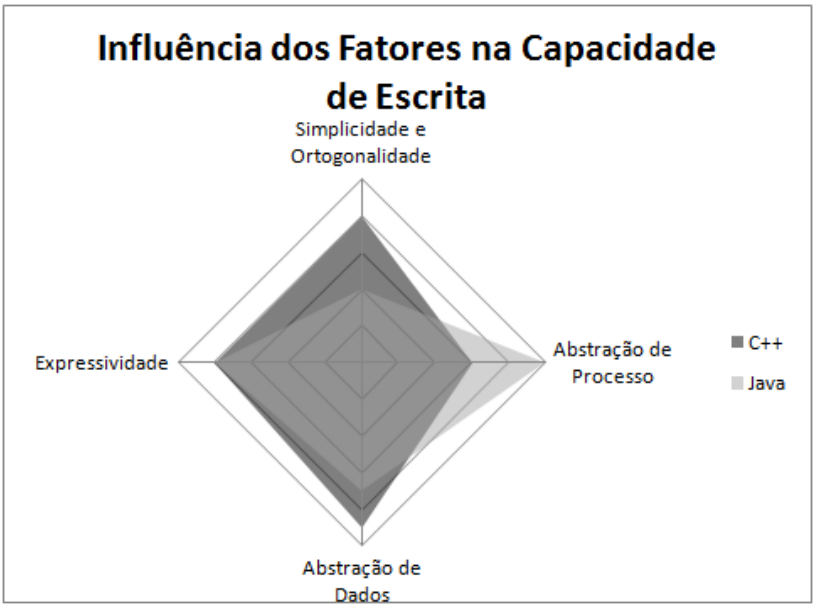


Figura 5.2: Gráfico radar da análise da capacidade de escrita.

### 5.1.3 Confiabilidade

A confiabilidade de uma linguagem visa garantir que um programa funcione de acordo com as suas especificações. A seguir, é feita uma análise dos recursos disponíveis em Java e C++ relacionados à sua confiabilidade, de acordo com os quesitos enunciados na seção 3.2.3.

A começar pela verificação de tipos, ambas as linguagens a realizam, quando possível, em tempo de compilação. No entanto, como C++ permite utilizar ponteiros de forma explícita, nem sempre essa verificação pode ser garantida, como demonstrado pelo trecho de código utilizado para conectar nodos em um grafo DirectShow, na seção anterior. Em situações mais extremas, C++ permite ainda utilizar ponteiros para os quais não se conhece o tipo, como é o caso do **void***, que utilizados de maneira incorreta, geralmente provocam erros em tempo de execução de difícil detecção, diminuindo assim a confiabilidade da linguagem. O mesmo exemplo se aplica aos vetores (**arrays**), que em C++ não fazem parte do tipo da linguagem e são construídos através de ponteiros, fato que impossibilita verificar o seu tamanho em tempo de compilação e que permite acesso a posições de memória fora de seus limites. Para contornar esse problema, é possível utilizar classes que representem vetores, tal como é feito em Java, onde é necessário conhecer o seu tamanho em tempo de compilação.

A manipulação de exceções confere à linguagem a habilidade de detectar e tratar erros em tempo de execução, impedindo que o programa termine de forma inesperada e reto-

[1]JSR: Uma especificação JSR visa padronizar os recursos disponíveis para a plataforma Java. Por exemplo, um aparelho que siga a JSR 135 deve possuir implementado (normalmente pelo fabricante) suporte à API multimídia MMAPI.

mando um estado de computação válido. Esse é um dos pontos fortes da linguagem Java. Embora C++ também suporte o tratamento de exceções, Java aprofunda esse conceito tornando-as obrigatórias em pontos críticos, como manipulação de vetores dinâmicos, verificação de tipos em tempo de execução, operações de entrada e saída, manipulação de arquivos (*handlers*), entre outros. O mesmo efeito pode ser facilmente obtido em C++, porém sempre ao encargo do programador, que deve identificar e tratar esses pontos manualmente. Assim, mesmo ambas as linguagens suportando essa manipulação, a confiabilidade de Java tende a ser maior nesse ponto, visto a obrigatoriedade de se tratar certos tipos de exceções.

Na questão de *aliasing*, o emprego de ponteiros permite que uma mesma posição de memória seja apontada por duas ou mais variáveis. Assim, a confiabilidade de C++ pode ser drasticamente reduzida caso o programador não tenha consciência disso, uma vez que a mudança de valor no endereço apontado por uma variável, se reflete automaticamente na outra. Como não há ponteiros explícitos em Java, esse risco é mitigado, porém ainda presente, pois a declaração de um objeto nada mais é do que um endereço, possibilitando que uma mesma área de memória seja representada por nomes diferentes, como demonstrado no exemplo abaixo. Portanto, o *aliasing* utilizado de forma descriteriosa reduz a confiabilidade da linguagem, principalmente em C++, onde seu uso é facilitado.

```
1  int* y;
2  int  x;
3
4  //Valor de y é inicializado
5  x = 1;
6  y = &x;
7  print(y);
8
9  //Valor de y é alterado indiretamente
10 x = 2;
11 print(y);
12
13 //saída: 1
14 //       2
```

Exemplo de *aliasing* em C++.

```
1  myClass objA = new myClass();
2  myClass objB = new myClass();
3
4  //Valor de x de A é inicializado
5  objA.x = 1;
6  print(objA.x);
7
8  //Valor de x de A é alterado indiretamente
9  objB = objA;
10 objB.x = 2;
11 print(objA.x);
12
13 //saída: 1
14 //       2
```

Exemplo de *aliasing* em Java.

Finalmente, outro quesito que impacta na confiabilidade da linguagem é o conjunto de sua legibilidade com sua capacidade de escrita em função do domínio do problema sobre o qual ela é empregada. Visto que o contexto deste trabalho se refere à manipulação

de vídeo em dispositivos móveis, ambas as linguagens (e suas APIs) utilizam paradigmas semelhantes orientado a objetos. Porém, a linguagem Java não suporta níveis mais baixos de programação e, uma vez que o acesso aos recursos de *hardware* é implementado por terceiros (em geral pelos fabricantes), nem sempre é possível saber como o aplicativo irá se comportar em determinadas situações. Assim, um código Java que funciona corretamente em um aparelho, pode funcionar diferentemente ou até não funcionar em outro, mesmo que o código seja portável. Logo, a falta de capacidade de Java em implementar tais acessos em baixo nível diminui a sua confiabilidade, uma vez que estes estão fora do controle do programador.

O gráfico 5.3 representa a relação dos fatores discutidos nesta seção em relação à confiabilidade, onde quanto mais afastado do centro, mais o fator a auxilia.

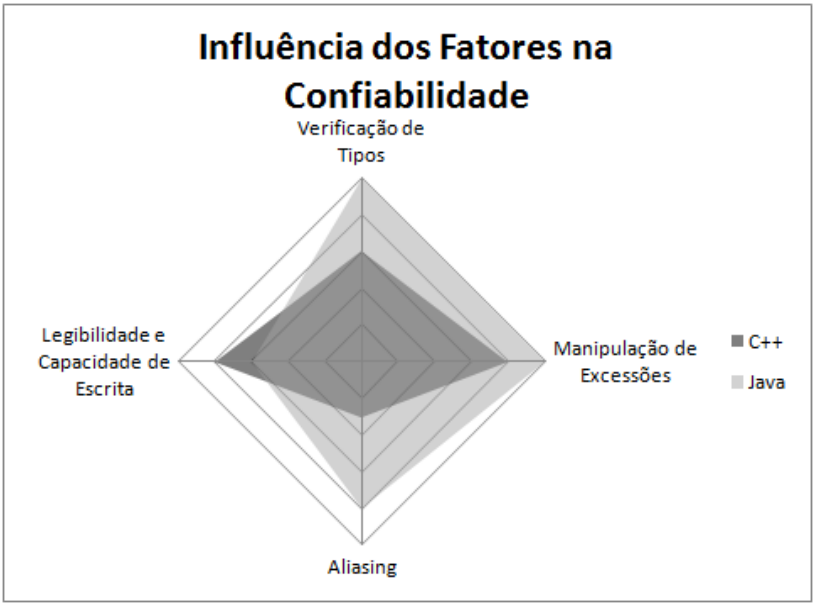


Figura 5.3: Gráfico radar da análise da confiabilidade.

### 5.1.4 Portabilidade

A análise da portabilidade das linguagens empregadas no contexto deste trabalho são de extrema importância, devido a vasta gama de plataformas e aparelhos existentes. Assim, esse fator tende a ser crucial no desenvolvimento de aplicativos para dispositivos móveis, uma vez que sendo necessário adaptar o código para cada um deles pode se tornar algo caro e dispendioso.

Uma das características mais fortes da linguagem Java é a sua alta portabilidade, uma vez que seu código intermediário, o Java bytecode, pode ser executado em qualquer máquina virtual (a JVM, vide seção 3.1). Em *desktops*, esse conceito é bastante difundido e por haver menos plataformas, é mais fácil manter a interoperabilidade entre as máquinas virtuais para cada ambiente. Contudo, se tratando de aparelhos celulares onde a variedade é muito maior, e considerando que cada fabricante deve implementar essa máquina virtual utilizando os recursos em baixo nível disponíveis no aparelho, a tarefa de manter um padrão de execução para o Java bytecode se torna complexa, fazendo com que o mesmo código seja executado de maneiras diferentes em cada plataforma. Logo, nesse ambiente a portabilidade de Java não é completa, pois algumas vezes é necessário utilizar códigos diferentes para se obter o mesmo fim.

A linguagem C++ para dispositivos móveis por sua vez, possui basicamente duas variantes. A primeira, utilizada em sistemas Windows Mobile da Microsoft, segue a convenção padrão do C++, enquanto que o Symbian C++, desenvolvido pela Nokia, possui alguns recursos a mais (descritos na seção 3.1) que visam a otimização de aplicativos para dispositivos móveis. Embora sejam semelhantes, seu código não é portável para outras plataformas, o que acaba limitando a abrangência de aplicativos em C++.

Mesmo não sendo possível portar um código C++ entre plataformas diferentes, a arquitetura COM (Component Object Model) da Microsoft permite que objetos possam ser portados entre linguagens diferentes, como por exemplo um filtro DirectShow que pode ser utilizado tanto em C++ como em C#. Embora ainda seja altamente dependente de plataforma, essa arquitetura permite que diversos programas utilizem os mesmos recursos, evitando o retrabalho.

Comparando a portabilidade de ambas as linguagens, é visto que nenhuma abrange esse conceito totalmente, porém, no caso da linguagem Java, há um grande esforço para que a portabilidade se torne viável facilmente, fato que não ocorre com C++ por ser uma linguagens totalmente compilada e nativa de plataformas distintas. Assim, analisando a portabilidade como a quantidade de código que precisa ser alterado para se passar de uma plataforma à outra, a linguagem C++, por possuir uma construção mais rígida, necessita de menos alterações. Por outro lado, se for considerado a portabilidade relativa ao número de plataformas abrangidas, é visto que Java, pelos motivos acima mencionados, pode ser executada em um número relativamente maior de plataformas.

### 5.1.5 Nível de Programação e Familiaridade

Outro fator importante na decisão da linguagem a ser utilizada em um projeto, é considerar se esta possui um nível de programação adequado para o problema e uma certa familiaridade por parte dos programadores.

Pelo nível de programação, é necessário saber se a linguagem oferece os recursos necessários aos requisitos de um projeto. Visto que há uma grande diferença entre o nível de programação de C++ e Java, esse fator merece especial atenção. Como foi visto ao longo das seções anteriores, a linguagem Java oferece um alto nível de programação, orientado a objetos. Embora isso facilite a abstração de diversos problemas, torna alguns outros difíceis ou até mesmo impossíveis de se resolver. Esse fato é claramente observado no contexto deste trabalho, onde mesmo que seja possível transmitir vídeo em tempo real utilizando Java, não é possível fazê-lo de uma maneira otimizada e nem altamente configurável, em virtude do alto nível de programação da linguagem. C++ por outro lado, permitindo o acesso direto ao *buffer* de memória de imagens da câmera, faz com que essa transmissão seja possível de uma maneira mais natural, permitindo ainda que uma variedade de configurações possam ser realizadas diretamente durante a captura (pré-processamento) como também na memória (pós-processamento).

É claro que a vantagem de C++ nesse contexto possui um custo, a familiaridade do programador. Embora ambas as linguagens sejam altamente difundidas atualmente, o uso das APIs de C++ para manipulação de vídeo pode se tornar uma tarefa custosa. Ao exemplo da API DirectShow, é necessário que o programador esteja familiarizado com as estruturas e convenções utilizadas pela plataforma Windows e sua API, como componentes COM (Component Object Model), programação orientada a eventos, troca de mensagens com o sistema operacional, entre outros. A API Symbian por sua vez, possui diversas convenções de programação (como descritos na seção 3.1) que visam obter o melhor desempenho de seus aplicativos em dispositivos com recursos limitados. Isso

requer que o programador esteja acostumado a utilizar construções que só existem praticamente nessa plataforma, o que exige um certo tempo de treinamento mesmo que este esteja familiarizado com a linguagem C++ padrão.

Java, ao contrário, possui uma API mais transparente e utiliza praticamente o mesmo estilo de programação usado em outros aplicativos da linguagem. Excluindo-se poucos parâmetros que são dependentes do aparelho utilizado, como as codificações e formatos disponíveis, a MMAPI possui estruturas familiares a qualquer programador Java, possibilitando um rápido aprendizado e desenvolvimento de programas para dispositivos móveis.

O gráfico 5.4 representa a relação dos fatores discutidos nas seções sobre portabilidade, nível de programação e familiaridade, onde quanto mais afastado do centro, mais presente é esse fator e mais auxilia o desenvolvimento dos aplicativos no contexto deste trabalho.

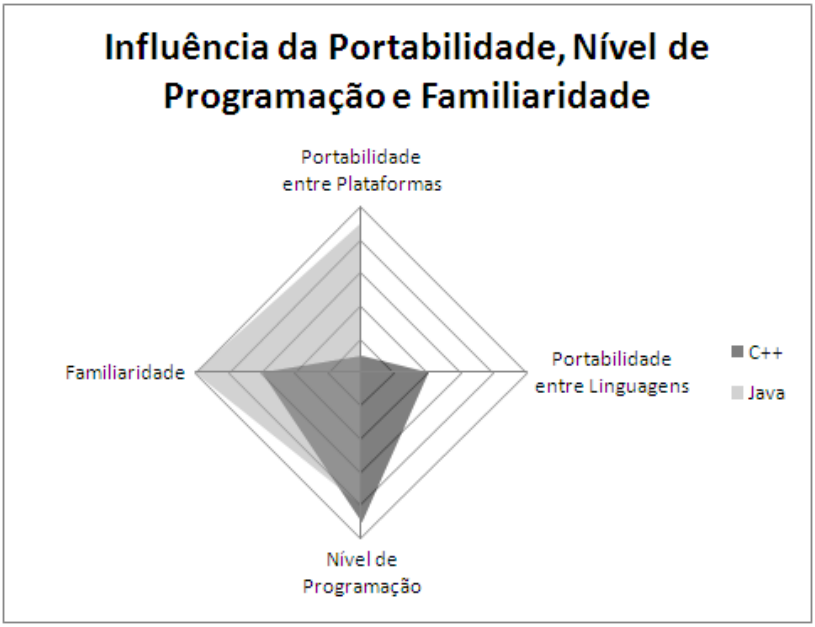


Figura 5.4: Gráfico radar da análise da confiabilidade.

### 5.1.6 Custo Final

Esclarecido as principais características e diferenças entre as linguagens C++ e Java, é possível agora realizar um estudo mais abrangente sobre os custos que mais influenciam a escolha de uma linguagem de programação.

Considerando o custo de treinamento de programadores, foi visto que Java possui uma maior familiaridade e simplicidade e uma menor ortogonalidade, assim como a MMAPI que utiliza basicamente os mesmos recursos de Java para *desktops*, fatores que facilitam o seu aprendizado. A linguagem C++, em contrapartida, tende a ter uma curva de aprendizado maior, bem como suas APIs que utilizam estruturas diferentes da linguagem padrão. Contudo, seu poder de programação neste contexto oferece mais flexibilidade ao programador, permitindo elaborar soluções mais complexas.

Em seguida, analisando o custo para escrever programas na linguagem, pode-se observar que programas em Java são prototipados e implementados mais rapidamente, em virtude da sua simplicidade. Por outro lado, C++ oferece recursos que facilitam a implementação de soluções mais complexas que em algumas ocasiões não podem ser feitos em Java de uma maneira ótima, como é o caso do envio de vídeo por *stream* apresentado

neste trabalho (vide capítulo 4).

O custo para compilar a linguagem em geral tem pouca influência em sua escolha. Contudo, se tratando de dispositivos móveis, esse quesito possui um peso maior, uma vez que nem todas as plataformas oferecem uma maneira simples de compilar, executar e depurar o código diretamente no aparelho. Outro fator inerente à programação em dispositivos móveis é a necessidade (por segurança) de alguns aplicativos possuírem assinaturas digitais (caso do Symbian C++) e permissões especiais (caso de Java) para ser executado na plataforma. No caso do Windows Mobile, a Microsoft oferece ferramentas (seção 4.2.1) que podem ser integradas com seu ambiente de programação, compilando e executando o programa diretamente no aparelho, enquanto que a depuração pode ser feita por *desktop*. Symbian C++ por sua vez, possui ferramentas disponibilizadas pela Nokia, similares às da Microsoft e capazes de assinar digitalmente os aplicativos para testes (assinatura de desenvolvedor válida para o aparelho). Já a linguagem Java não possui ferramentas padrões para desenvolvimento, sendo que estas em geral são oferecidas de forma independente pelos fabricantes. Contudo, quando não oferecidas, é necessário que o programador instale o programa manualmente e o conceda as permissões necessárias para executar, como acesso à rede e ao sistema de arquivos.

Se tratando do custo de execução do programa, é esperado que aplicações em C++ possuam um desempenho maior em relação a Java, por serem desenvolvidas na linguagem nativa do aparelho, o que possibilita sua execução direta, e por serem totalmente compiladas, ao contrário de Java que possui uma etapa de interpretação em sua máquina virtual. Contudo, essa etapa adicional confere à linguagem uma maior portabilidade de seu código, característica importante no contexto dos dispositivos móveis em virtude de sua grande variedade, visto que C++ não abrange tantas plataformas como Java.

Analisando o custo do sistema de implementação, Java oferece diversas ferramentas de forma gratuita e que podem ser utilizadas em praticamente qualquer plataforma (como seus SDKs e IDEs). Já a linguagem C++ possui ferramentas específicas para cada ambiente, que em alguns casos são pagas (como demonstrado na seção 4.2). Embora seja possível desenvolver aplicativos experimentais utilizando versões gratuitas dessas ferramentas, seu uso profissional geralmente requer a versão completa, adicionando custos ao projeto.

Considerando a confiabilidade da linguagem no contexto deste trabalho, é observado que Java é muito menos suscetível a erros em relação a C++, pelos motivos descritos nas seções anteriores. Porém, se considerarmos que esse contexto não apresenta fatores de risco críticos (como de vida ou financeiros), o custo da confiabilidade possui um peso menor, podendo ser trocada, caso seja necessário, pela maior flexibilidade de uma linguagem como C++.

Por fim, analisando o custo da manutenção da linguagem, é perceptível que esta está ligada à sua legibilidade. Como enunciado anteriormente, Java possui uma maior legibilidade em termos gerais do que C++, visto que grande parte de sua sintaxe e semântica foram implementadas visando simplificar a linguagem C++. Disso, decorre que um código em Java tende a ser mais claro e fácil de ler, auxiliando a sua manutenção. Por outro lado, corrigir ou adicionar código em um aplicativo C++ costuma ser mais oneroso, pois como foi demonstrado, há uma maior probabilidade de se cometer erros que não são acusados em tempo de compilação. Outro fator que pode auxiliar a manutenção de um código são as ferramentas disponíveis, ao exemplo das IDEs Eclipse (Java) e Carbide.c++ que podem detectar certos erros mesmo em tempo de escrita.

Assim, considerando as características descritas acima, percebe-se que a definição do

custo final e a escolha de uma linguagem de programação depende de diversos fatores, como abrangência do aplicativo em termos de plataforma, domínios do problema, nível de experiência dos programadores, entre outros. Aplicando então esses conceitos a este trabalho, conclui-se que a linguagem Java é uma boa escolha para aplicativos que façam uso dos recursos multimídias de um aparelho localmente, que não necessitem de grande flexibilidade e desempenho e que abranja o maior número possível de plataformas com um único código de fácil manutenção. A linguagem C++, por outro lado, confere uma ótima flexibilidade, permitindo incorporar novas funcionalidades às suas APIs, aliado a um bom desempenho. Contudo, isso vem em troca de sua maior complexidade em relação a Java, tornando o projeto mais longo, custoso e de manutenção mais complexa.

O gráfico 5.5 mostra de uma maneira geral os fatores discutidos nesta seção, aplicados ao contexto do trabalho. Como descrito, a legibilidade geral de Java é maior que a de C++, devido a sua simplicidade. A capacidade de escrita de Java em um contexto geral também é maior, porém neste trabalho, o seu alto nível de programação impediu que o aplicativo fosse implementado da maneira proposta, prejudicando a sua escrita. Em termos de confiabilidade, foi visto que Java possui mais características para tal, como ausência de ponteiros e um sistema de tratamento de exceções mais eficiente do que em C++. Contudo, essa confiabilidade é prejudicada quando se trata de multiplataformas, pois a implementação da sua máquina virtual é feita, em geral, por cada fabricante e, embora haja padrões que a definam, nem sempre eles são implementados ou seguidos. Sobre a portabilidade, Java oferece muito mais flexibilidade através de sua máquina virtual, enquanto que C++ requer suporte nativo do sistema. Contudo, através da arquitetura COM da Microsoft, é possível atingir certa portabilidade entre outras linguagens, porém na mesma plataforma. Analisando o nível de programação, foi visto que Java possui um nível mais alto, totalmente orientado a objetos, enquanto que C++ permite construções procedurais e acessos a baixo nível. Por fim, a familiaridade de Java é maior por ser uma linguagem mais simples que C++, ao mesmo tempo que mantém essa estrutura no caso dos dispositivos móveis. C++ por outro lado, utiliza mais estruturas e recursos exclusivos ao desenvolvimento nessas plataformas, diminuindo a sua familiaridade mesmo para programadores acostumados à linguagem.

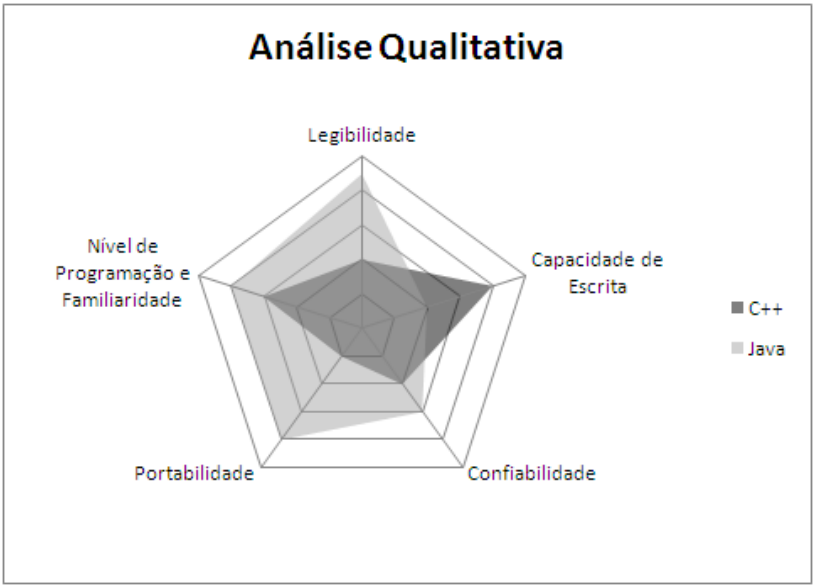


Figura 5.5: Gráfico radar da análise qualitativa.

## 5.2 Análise Quantitativa

Passando para a análise quantitativa, é esperado encontrar na prática os conceitos acima descritos. Essa análise é feita sobre os códigos fonte e a execução dos aplicativos desenvolvidos, visando demonstrar as características do uso das linguagens no contexto deste trabalho.

### 5.2.1 Análise do Código

Para analisar o código, foi utilizado a ferramenta RSM Wizard[1], capaz de contabilizar as informações de um projeto em ambas as linguagens. A partir da contagem mostrada na tabela 5.1 percebe-se que a solução feita em C++ é em torno de quatro a cinco vezes maior, em virtude da maior complexidade da API utilizada (DirectShow) em relação a API Java (MMAPI), atentando para o fato de que as medidas de eLOC (Effective Lines of Code) e lLOC (Logical Lines of Code) são as que realmente expressam código computável. Outro fator que favorece essa diferença é a necessidade de se utilizar arquivos de cabeçalho (*headers*) em C++, fazendo com que parte do código (declaração de classes e funções) seja replicada, ao contrário de Java, onde as classes são declaradas e implementadas no mesmo arquivo. Esses fatores fazem com que um código em C++ em geral leve mais tempo para ser escrito do que em Java.

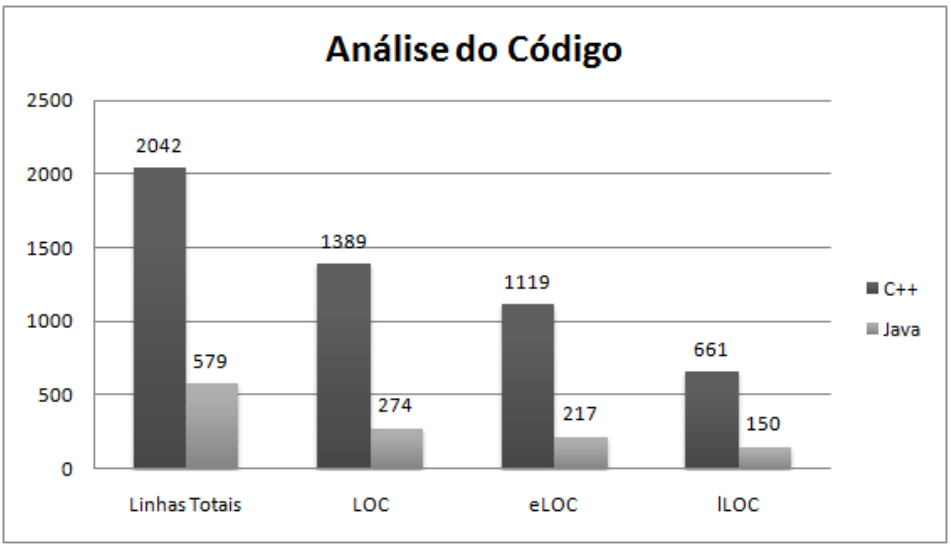


Figura 5.6: Gráfico da análise do código.

Tabela 5.1: Análise quantitativa do código das implementações.

| Quesito | C++ | Java | Relação C++/Java |
|---|---|---|---|
| Arquivos | 20 | 4 | 5 |
| Linhas totais | 2042 | 579 | 3.52 |
| LOC | 1389 | 274 | 5.06 |
| eLOC | 1119 | 217 | 5.15 |
| lLOC | 661 | 150 | 4.40 |

[1]RSM Wizard: http://msquaredtechnologies.com/

A análise das funções (métodos em Java) está representada na tabela 5.2. Novamente é possível perceber que, em virtude da complexidade das APIs, C++ utiliza cerca de duas vezes mais funções e métodos do que Java. Essas mesmas funções utilizam quase cinco vezes mais parâmetros em C++, dificultando seu uso e diminuindo sua capacidade de escrita e leitura, bem como sua confiabilidade, uma vez que estão mais suscetíveis a erros de ordenamento de parâmetros. Outro fator complicante é o uso excessivo de pontos de retorno, muito utilizados em C++ para tratar erros limpando estruturas pré-alocadas dentro de uma função. Note que em Java é mantido o conceito de pontos de entrada e saída únicos (pontos de retornos equivalem ao número de funções), o que facilita a leitura e a depuração de erros. Essa análise é refletira diretamente na complexidade de interface, que contabiliza os parâmetros de entrada com os pontos de retornos, demonstrando que as interfaces das funções em C++ são mais elaboradas em comparação com as de Java. Por fim, a complexidade ciclomática demostra a quantidade de instruções de desvios condicionais ou não no código. Como esperado, pelo maior número de linhas e funções em C++, existe em seu código cerca de quatro vezes mais instruções de laços e desvios, que também diminuem a capacidade de leitura e torna o código mais propenso a erros lógicos por parte do programador.

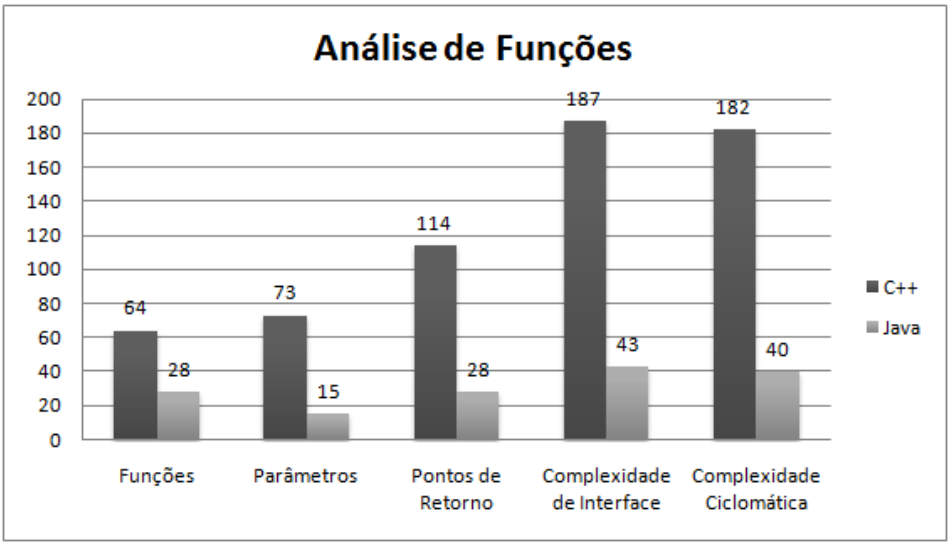


Figura 5.7: Gráfico da análise funcional.

Tabela 5.2: Análise quantitativa funcional.

| Quesito | C++ | Java | Relação C++/Java |
|---|---|---|---|
| Funções | 64 | 28 | 2.28 |
| Parâmetros | 73 | 15 | 4.86 |
| Pontos de retorno | 114 | 28 | 4.07 |
| Complexidade de interface | 187 | 43 | 4.34 |
| Complexidade ciclomática | 182 | 40 | 4.55 |

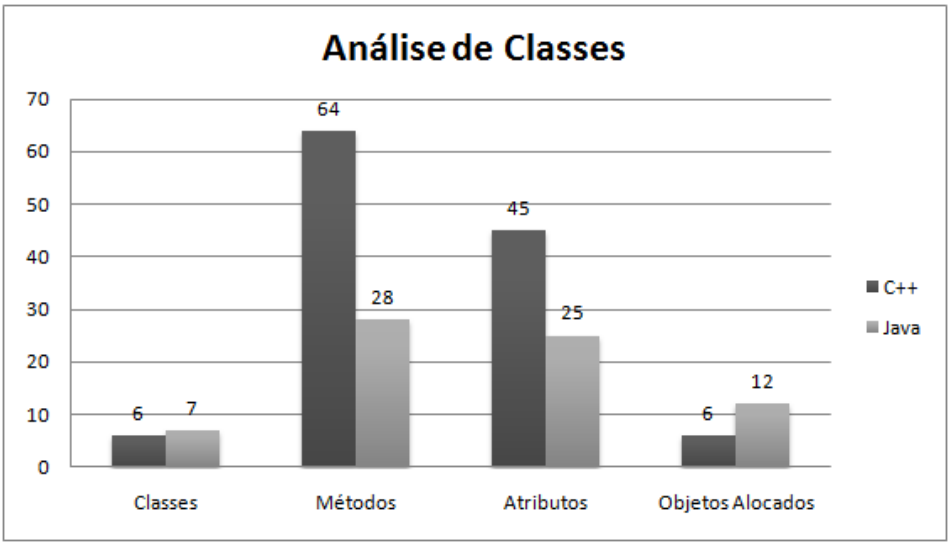


Figura 5.8: Gráfico da análise de classes.

A próxima tabela, 5.3, relaciona as métricas referentes as classes, fazendo uma melhor análise sobre a orientação a objetos e abstração das implementações. Mesmo o aplicativo em C++ tendo cerca de cinco vezes mais código, o de Java possui mais classes, refletindo seu forte paradigma orientado a objetos. Semelhante ao caso das funções, o código C++ possui cerca de duas vezes mais métodos e atributos, mostrando que suas classes são maiores e mais complexas, o que dificulta o reuso e diminui a coesão das mesmas, além da já citada capacidade de leitura. Contabilizando os objetos alocados, novamente percebe-se o forte paradigma de Java, sendo que este número é duas vezes maior do que no caso de C++.

Tabela 5.3: Análise quantitativa de classes.

| Quesito | C++ | Java | Relação C++/Java |
|---|---|---|---|
| Classes | 6 | 7 | 0.85 |
| Métodos | 64 | 28 | 2.28 |
| Atributos | 45 | 25 | 1.8 |
| Objetos alocados | 6 | 12 | 0.5 |

### 5.2.2 Análise de Desempenho

Nesta seção, é realizada a análise de desempenho dos aplicativos propostos. Como enunciado na seção 3.3.2, o aparelho selecionado para realizar estes testes é o Motorola Q11 (especificações técnicas no apêndice 6), em virtude da sua capacidade de executar ambas as implementações.

O primeiro quesito analisado é o tamanho do arquivo executável. Embora nestes casos de testes seu tamanho não seja tão relevante, essa medida reflete a proporção da quantidade de código a mais necessária no aplicativo em C++ (cerca de sete vezes) em relação ao aplicativo em Java.

O segundo quesito, tempo de inicialização, é obtido pelo código, conforme demostrado na seção 3.3.2. O valor calculado é referente à média de cinco execuções, onde o maior e menor tempo são descartados. Neste ponto é possível perceber a contrapartida de C++ que, apesar de possuir um código maior e relativamente mais complexo, consegue

inicializar o aplicativo cerca de quatro vezes mais rápido do que Java. Essa diferença provém basicamente do fato de Java rodar sobre uma máquina virtual (KVM), onde seu código é interpretado, ao contrário de C++ que, por ser a linguagem nativa do sistema, é executado diretamente, conforme explicado na seção 3.1 sobre o projeto das linguagens.

Finalmente, ao analisarmos a quantidade de memória volátil utilizada durante a execução, percebe-se mais uma vez o reduzido desempenho do aplicativo Java em comparação com o aplicativo C++, o qual utiliza cerca de oitenta e seis vezes menos memória. Novamente isso se deve ao fato de que Java utiliza uma máquina virtual, além de seu paradigma totalmente orientado a objetos, que utiliza mais memória do que estruturas de dados mais simples. O uso da memória é um dos requisitos mais importantes quando se desenvolve programas para dispositivos móveis pois, além de ser um recurso limitado na maioria dos aparelhos (conforme explicitado na seção 2.1, sobre limitações), é um dos componentes que mais consomem energia, diminuindo drasticamente a vida útil do aparelho, como demonstrado por Kato (KATO; LO, 2007). Assim, pode-se concluir que a simplicidade de Java em relação a C++ vem ao custo de seu desempenho, lembrando também que a maior justificativa desta troca é a grande portabilidade dos aplicativos Java.

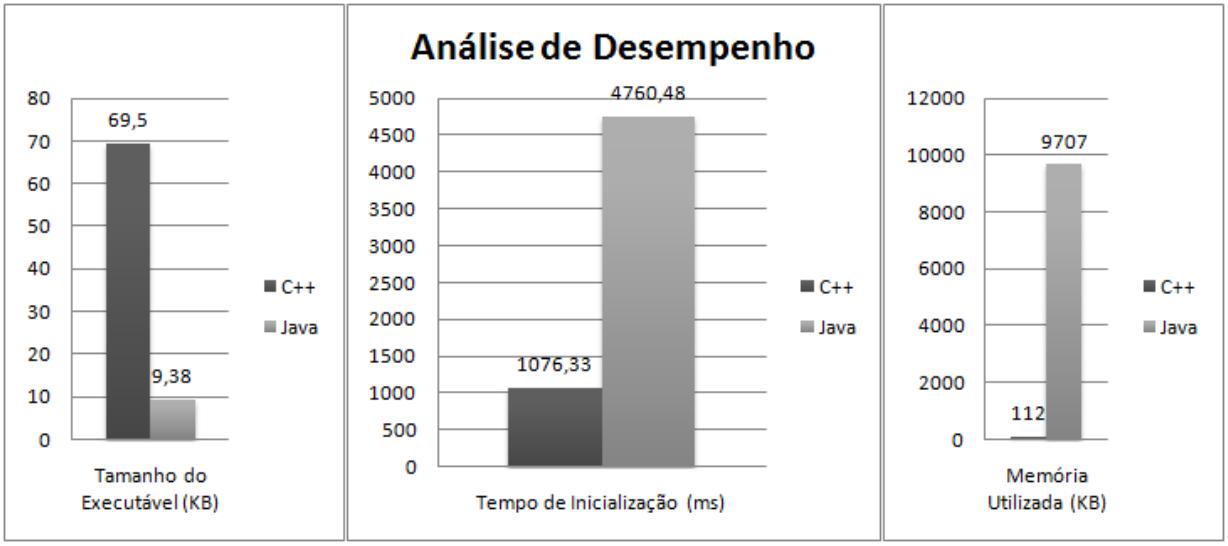


Figura 5.9: Gráfico da análise de desempenho

Tabela 5.4: Análise de desempenho das implementações.

| Quesito | C++ | Java | Relação C++/Java |
|---|---|---|---|
| Tamanho do Executável (KB) | 69.5 | 9.38 | 7.4 |
| Tempo de inicialização (ms) | 1076.33 | 4760.48 | 0.22 |
| Memória utilizada (KB) | 112 | 9707 | 0.01 |

### 5.2.3 Conclusões Sobre a Análise Quantitativa

Nas seções anteriores, foi demonstrado que o código do aplicativo proposto em C++ é cerca de quatro vezes maior, em virtude de sua maior complexidade. Esse fato se reflete também em sua análise funcional, onde C++ utiliza cerca de duas vezes mais funções, as quais também são mais complexas que as de Java, possuindo mais código por função, mais

desvios e interfaces com mais parâmetros. Em termos de classes, Java as implementa em maior número e aloca mais objetos, porém com menos código, mostrando que as classes de C++ são mais complexas. Essas informações são confirmadas considerando-se o tempo de desenvolvimento das soluções, onde no aplicativo em C++ foram empregados cerca de três meses, desde o aprendizado sobre suas estruturas até a implementação final, enquanto que em Java foi necessário cerca de três semanas. Sobre o desempenho, foi medido que, embora o aplicativo em C++ seja maior, este executa de maneira mais eficiente, inicializando em torno de quatro vezes mais rápido e ocupando oitenta e seis vezes menos memória. Deve-se ressaltar, porém, que o aplicativo em Java possui uma boa parte do consumo de memória fixo, dedicado à sua máquina virtual, razão também pela qual o seu código é executado mais lentamente, ou seja, interpretado. Essas informações são mostradas no gráfico 5.10, construído a partir da relação C++/Java.

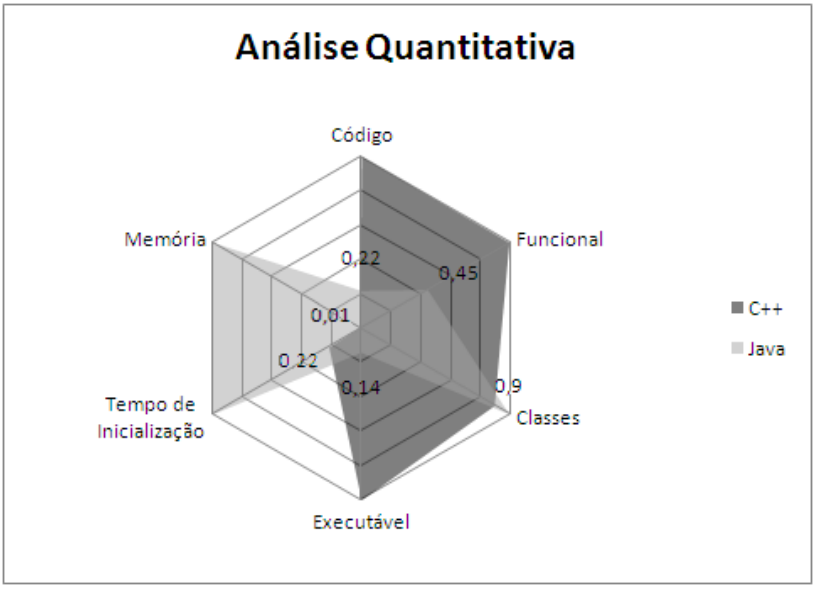


Figura 5.10: Gráfico radar da análise quantitativa.

# 6 CONCLUSÕES

Atualmente, presencia-se uma grande expansão do mercado de telefonia móvel e, por questões de mercado, é esperado que os mesmos procurem agregar cada vez mais serviços, ao exemplo do acesso à *internet*. Novos recursos também são constantemente adicionados, como câmeras, telas de alta resolução, conexões *bluetooth* e redes *wireless*. Baseado nisso, é esperado que a transmissão de vídeo por meio de dispositivos móveis, antes inviável, ganhe novamente a atenção, em vista da crescente necessidade de comunicação e do valor que tais serviços podem agregar às operadoras.

Assim, este trabalho apresentou o estado da arte do desenvolvimento de aplicativos capazes de manipular vídeo em dispositivos móveis. Como visto, há uma vasta gama de plataformas e linguagens disponíveis, cada qual com APIs e ferramentas diferentes, onde uma análise mais profunda é essencial para se tomar uma decisão em um projeto. Além da grande diversidade de plataformas, deve-se considerar também as limitações inerentes aos dispositivos portáteis, tais como memória, processamento, bateria e transmissão, bem como fatores mercadológicos referentes à portabilidade e abrangência do aplicativo.

Logo, para avaliar esses quesitos, foi proposto a implementação de aplicativos testes, utilizando-se as linguagens mais representativas atualmente: C++ e Java. Para validar o estudo, foram empregadas métricas qualitativas e quantitativas presentes na literatura atual, as quais oferecem uma visão sobre a construção e o uso das linguagens, desde aspectos referentes à escrita e leitura, bem como complexidade e desempenho.

Qualitativamente, neste contexto, foi visto que Java é mais simples de usar, pois possui uma maior legibilidade, confiabilidade e portabilidade em relação à C++, adotando um paradigma orientado a objetos. A linguagem C++, por outro lado, adota tanto o paradigma procedural como orientação a objetos, o que a permite construir soluções mais complexas e flexíveis e, embora a linguagem Java em geral tenha uma maior capacidade de escrita, a mesma não permite acessos diretos à memória, fato que confere à C++ uma maior redigibilidade das implementações propostas.

Analisando as métricas quantitativas, foi mostrado que a solução em C++ é em média cinco vezes maior, utilizando funções e classes mais complexas, em geral devido a estrutura de sua API. Java entretanto, mesmo possuindo um código menor, implementa e aloca mais classes e objetos, em virtude do seu paradigma. Passando para a análise de desempenho, foi mostrado que a solução em Java é cerca de quatro vezes mais lenta que em C++ e utiliza em torno de oitenta e cinco vezes mais memória, uma vez que o aplicativo Java é interpretado em uma máquina virtual (KVM), a qual é carregada juntamente.

Logo, conclui-se através deste estudo que, ao escolher uma linguagem de programação para dispositivos móveis, existem diversos fatores que devem ser analisados e que em grande parte são dependentes do projeto. Do lado de Java, tem-se a seu favor a maior facilidade de aprender e utilizar a linguagem e suas APIs, além da sua portabilidade, que é um

dos fatores mais influenciantes na área de móveis. Fatores estes que são trocados pelo seu reduzido desempenho e falta de capacidade em implementar soluções de mais baixo nível, o que prejudicou a sua implementação neste contexto. Do lado de C++, destaca-se a sua flexibilidade, permitindo implementar soluções altamente configuráveis, bem como seu desempenho, por ser executado nativamente pelo sistema. Em contrapartida, seu aprendizado e uso é mais complexo, pois requer maiores cuidados por parte do programador. Uma última análise revela que as ferramentas e linguagens avaliadas ainda não oferecem maneiras diretas para se manipular vídeo em dispositivos móveis da maneira proposta neste trabalho. Contudo, é esperado que em breve suas APIs forneçam formas mais claras e eficientes de realizar tais operações, tento em vista a projeção para esse mercado.

Por fim, como trabalhos futuros, foi sugerido ao longo deste uma melhor análise e implementação do envio de vídeo por *stream* utilizando formatos mais eficientes, como o ASF. Outra sugestão seria analisar mais linguagens de programação voltadas a dispositivos móveis, tanto neste como em outros contextos. Trabalhos complementares também são sugeridos na área de qualidade de serviços (QoS ou *Quality of Service*) que garantam os padrões aceitáveis de qualidade de imagem, transmissão, dentre outros.

# REFERÊNCIAS


3GPP. **3rd Generation Partnership Project**. Acesso em: junho 2009. Disponível em: <http://www.3gpp.org/ftp/Specs/html-info/26244.htm>.

ALBRECHT, A. J. **Measuring Application Development Productivity**. [S.l.]: IBM Corporation, 1977. Disponível em: <http://www.bfpug.com.br/Artigos/Albrecht/MeasuringApplicationDevelopmentProductivity.pdf>.

APPLE. **iPhone Site**. Disponível em: <http://www.apple.com/iphone/>.

APPLEOS. **Developing for iPhone OS 2.2.1**. Acesso em: maio 2009. Disponível em: <http://developer.apple.com/iphone/index.action>.

CAMPO, C.; NAVARRETE, C.; GARCIA-RUBIO, C. **Performance Evaluation of J2ME And Symbian Applications in Smart Camera Phones**. [S.l.: s.n.], 2007.

CLEMENTE, R. G. **Uma Solução de Streaming de Vídeo para Celulares**: conceitos, protocolos e aplicativos. [S.l.]: Universidade Federal do Rio de Janeiro, 2006.

CRUZ, V. M.; MORENO, M. F.; SOARES, L. F. G. **TV Digital para Dispositivos Portáteis - Middlewares**. [S.l.]: PUC - Rio de Janeiro, 2008.

DERSHEM, H. L.; JIPPING, M. J. **Programming Languages**: structures and models. Belmont, CA, USA: Wadsworth Publishing Company, 1990.

DHIR, A. **The Digital Consumer Technology Handbook**. 1st.ed. Burlington, MA, USA: Elsevier, 2004.

DIRECTSHOW, M. **DirectShow Video Capture**. Acesso em: agosto 2009. Disponível em: <http://msdn.microsoft.com/en-us/library/ms940077.aspx>.

FáBIO S. CORIOLANO, T. A. B. **Uma estratégia para recepção de streaming de vídeo em dispositivos móveis**. [S.l.]: Universidade de Salvador, 2008.

FITZEK, F. H. P.; REICHERT, F. **Mobile Phone Programming and its Application to Wireless Networking**. 1st.ed. [S.l.]: Springer, 2007.

GARTNER. **Gartner Says Worldwide Smartphone Sales Grew 16 Per Cent in Second Quarter of 2008.** Disponível em: <http://www.gartner.com/it/page.jsp?id=754112>. Acesso em: abril 2009.

HENRICSON, M.; NYQUIST, E. **Programming in C++, Rules and Recommendations**. [S.l.: s.n.], 1992. Disponível em: <http://www.doc.ic.ac.uk/lab/cplus/c++.rules/>.

JODE, M. de; TURFUS, C. **Symbian OS System Definition**. Acesso em: maio 2009. Disponível em <http://developer.symbian.com/main/downloads/papers/SymbOS_def/symbian_os_sysdef.pdf>.

KATO, M.; LO, C. **Power Consumption Reduction in Java-enabled, Battery-powered Handheld Devices through Memory Compression**. [S.l.]: Univ. of Texas at San Antonio, 2007.

KIRKUP, M. **Introduction to BlackBerry® Development.** Disponível em: <http://www.blackberry.com/DevMediaLibrary/view.do?name=introblackberrydev>. Acesso em: maio 2009.

MICROSOFT. **Getting Started in Developing Applications for Windows Mobile 6**. Disponível em: <http://msdn.microsoft.com/pt-br/library/ bb158522.aspx>. Acesso em: maio 2009.

MOTOROLA. **Optimizing a Java ME Application**. [S.l.: s.n.], 2006. Disponível em: <http://developer.motorola.com/docstools/articles/Optimize_20061001.pdf>.

MSQUARED. **Resource Standard Metrics - Metrics Definitions**. [S.l.]: MSquared, 2009.

NOKIA. **S60 Platform**: image and video capturing. Acesso em: agosto 2009. Disponível em <http://tinyurl.com/SymbianOSMultimedia>.

NYTIMES. **Apple Profit Rises 15%, Driven by iPhone Sales**. Acesso em: maio 2009. Disponível em: <http://www.nytimes.com/2009/04/23/technology/ companies/23apple.html>.

O'HARA, K.; MITCHELL, A. S.; VORBAU, A. Consuming video on mobile devices. In: CHI '07: PROCEEDINGS OF THE SIGCHI CONFERENCE ON HUMAN FACTORS IN COMPUTING SYSTEMS, 2007, New York, NY, USA. **Anais...** ACM, 2007. p.857–866.

PRECHELT, L. **An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program**. TH, Germany: Universität Karlsruhe, 2000.

SCOTT, M. L. **Programming Language Pragmatics**. 1st.ed. San Francisco, CA, USA: Morgan Kaufmann, 2000.

SEBESTA, R. W. **Conceitos de Linguagens de Programação**. 8.ed. Porto Alegre: Bookman, 2008.

STEFFEN, J. B.; CUNHA, P. M. da. **Build your own Camera Application in Java ME, using JSR-135 and JSR-234 and explore your mobile resources.** [S.l.: s.n.], 2007.

SUN, M. **J2ME Building Blocks for Mobile Devices - White Paper on KVM and the Connected, Limited Device Configuration (CLDC)**. Acesso em: outubro 2009. Disponível em: <http://java.sun.com/products/cldc/wp/>.

SYMBIAN. **Symbian OS**: active objects and the active scheduler. 2004.

TELECO. **Serviços 3G**. Acesso em: maio 2009. Disponívem em: <http://www.teleco.com.br/3g_servicos.asp>.

WATT, D. A. **Programming Language Design Concepts**. 1st.ed. England: John Wiley, 2004.

WMSDK. **Windows Mobile 6 Documentation**. Acesso em: setembro 2009. Disponível em: <http://msdn.microsoft.com/en-us/library/bb158532.aspx>.

YUAN, M. J.; SHARP, K. **Developing Scalable Series 40 Applications**: a guide for java developers. [S.l.]: Addison-Wesley, 2004.

# ANEXO ESPECIFICAÇÕES TÉCNICAS

Segue abaixo as especificações técnicas do aparelho Motorola Q11 utilizado para testes, retirado de http://www.gsmarena.com/motorola_q_11-2541.php.

**Motorola Q 11**


| | | |
|---|---|---|
| GENERAL | 2G Network | GSM 850 / 900 / 1800 / 1900 |
| | Announced | 2008, October |
| | Status | Available. Released 2008, December |
| SIZE | Dimensions | 117 x 64 x 11.7 mm, 85 cc |
| | Weight | 115 g |
| DISPLAY | Type | TFT, 65K colors |
| | Size | 320 x 240 pixels, 2.4 inches |
| | | - Full QWERTY keyboard<br>- 5-way navigation button<br>- Downloadable wallpaper and screensavers |
| SOUND | Alert types | Vibration; Polyphonic, MP3 ringtones |
| | Speakerphone | Yes |
| MEMORY | Phonebook | Practically unlimited entries and fields, Photocall |
| | Call records | Practically unlimited |
| | Internal | 64 MB RAM, 128 MB Flash |
| | Card slot | microSD (TransFlash), up to 16GB |
| DATA | GPRS | Class 10 (4+1/3+2 slots), 32 - 48 kbps |
| | HSCSD | Yes |
| | EDGE | Class 10, 236.8 kbps |
| | 3G | No |
| | WLAN | Wi-Fi 802.11 b/g |
| | Bluetooth | Yes, v2.1 with A2DP |
| | Infrared port | No |
| | USB | Yes, v1.1 microUSB |
| CAMERA | Primary | 3.15 MP, 2048x1536 pixels, LED flash |
| | Video | Yes, 15fps |
| | Secondary | No |
| FEATURES | OS | Microsoft Windows Mobile 6.1 Standard |
| | CPU | Freescale ARM 7 LTE processor |
| | Messaging | SMS (threaded view), MMS, Email, Push Email, IM |
| | Browser | WAP 2.2/HTML (Pocket IE6) |
| | Radio | No |
| | Games | Yes + downloadable |
| | Colors | Black |
| | GPS | Yes, with A-GPS support |
| | Java | Yes, MIDP 2.0 |
| | | - MP3/AAC+/WAV/WMA music player<br>- MP4/3gp/MWV/H.264 video player<br>- Voice memo/dial<br>- T9 |
| BATTERY | | Standard battery, Li-Ion 1170 mAh |
| | Stand-by | Up to 195 h |
| | Talk time | Up to 7 h 30 min |

# APÊNDICE O PROBLEMA DO DIAMANTE

O problema do diamante, no contexto de herança múltipla, consiste da ambiguidade que surge em uma estrutura mostrada na Figura 1.

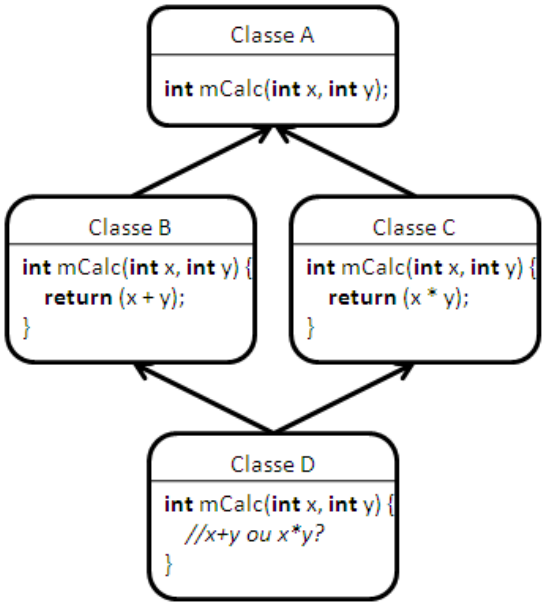


Figura 1: Estrutura do problema do diamante.

Neste caso, duas classes **B** e **C** herdam de uma mesma classe **A**, cada qual implementando o método *mCalc*. Em sequencia, uma classe **D** que herda de ambas as classes **B** e **C** também possui o método *mCalc*, porém de forma ambígua, pois não é possível saber se esse método se refere ao da classe **B** ou da classe **C**.

As linguagens C++ e Java resolvem esse problema de forma diferente. No primeiro caso (C++), é possível declarar as classes **B** e **C** como uma herança virtual da classe **A**:

```
class A {
  int mCalc(int x, int y) {
    return (x+y);
  }
}

class B: virtual public A { }

class C: virtual public A { }

class D: public B, public C { }
```

Herança virtual em C++.

Assim, ambas **B** e **C** compartilham a mesma referência da classe **A**, contudo o método *mCalc* é implementado apenas em **A**. Outra alternativa, que possibilita múltiplas implementações de *mCalc*, é nomear a chamada do método em relação à sua classe, através dos identificadores B::mCalc e C::mCalc:

```
D objD;
objD.B::mCalc();
```

Deste modo, o compilador pode agora decidir qual dos métodos será chamado. Já no caso de Java, não é possível implementar herança múltipla diretamente, por decisão do projeto da linguagem. Contudo, a alternativa oferecida é a herança múltipla de classes interface, pois como não implementam os métodos, mostrando apenas as suas assinaturas, não há ambiguidade.