4
Arquitetura Computacional do *codec* Open DVC

Uma das grandes dificuldades para a continuidade das pesquisas na área de Codificação Distribuída de Vídeo (DVC) é a indisponibilidade de softwares desenvolvidos que implementam as técnicas para manipular os quadros de vídeo, nos diversos processos aos quais esses quadros têm que ser submetidos.

Praticamente cada grupo de pesquisa ou aluno que inicia o estudo do assunto é obrigado a começar seu desenvolvimento "do zero", visto que há pouca disponibilidade desses softwares, pouca documentação e o pouco que se tem não se encontra adequadamente preparado, do ponto de vista da engenharia de software, para sofrer adições diretas em relação à sua modularidade.

Nesse panorama, os novos interessados que resolvem estudar o assunto perdem grande parte do tempo de pesquisa desenvolvendo o software para manipulação de processos já estudados e conhecidos por outros grupos, comprometendo com isso o esforço do estudo, pois o foco do desenvolvimento deveria estar concentrado unicamente na nova contribuição a ser implementada.

No presente trabalho, foi projetada e implementada uma versão do *codec* DVC em linguagem orientada a objetos, de forma modularizada, de tal maneira que futuros estudos com adições ou melhorias dos algoritmos implementados não necessitem desenvolver novos sistemas, mas unicamente programar o módulo a ser adicionado ou modificado. Assim, após os testes da implementação das técnicas utilizadas em linguagem interpretada em ferramenta de programação matemática (MATLAB), o código foi transcrito para linguagem orientada a objetos, com a respectiva documentação.

A programação em MATLAB tem alguns problemas. Primeiramente, a linguagem é completamente estruturada, não aproveitando as vantagens de novos paradigmas mais reutilizáveis como a orientação a objetos. Além disso, por ser uma linguagem interpretada, o tempo estimado de execução de um determinado algoritmo, se comparado a uma programação em linguagem nativa de alto nível, é da ordem de 10 para 1. Ou seja, o tempo será 10 vezes maior para a execução do mesmo algoritmo em MATLAB.

Porém, existe uma vantagem importante no desenvolvimento em MATLAB. O fato de muitos algoritmos, manipulações matemáticas (principalmente algébricas) e bibliotecas de várias áreas (inclusive as de processamento de imagens e vídeo) já estarem implementadas, faz com que o desenvolvimento seja muito mais rápido. Por exemplo, várias funções utilizadas no *codec* Open DVC e que em MATLAB eram escritas em uma única linha, ao se transformarem em linguagem orientada a objetos, tem seu código transformado em várias linhas e laços ou até várias classes para uma única função.

Num primeiro momento, para se ganhar velocidade de desenvolvimento e retorno rápido de resultados e correções, implementaram-se as técnicas a serem testadas em MATLAB e, após isso, depois de todas as modificações e correções, que também são mais simples em MATLAB, houve uma preocupação com a melhora do desempenho e organização do código do sistema, fazendo a tradução para uma linguagem moderna orientada a objetos. Outro detalhe importante é que, como o projeto e a documentação serão construídos seguindo as notações e padrões da orientação a objetos, o sistema pode ser implementado utilizando geradores de código em qualquer linguagem OO, seja C++, Java, Ruby, C#, etc.

Percebe-se assim que, escrevendo os algoritmos em linguagem orientada a objetos, além de se ganhar todas as vantagens das linguagens desse paradigma, tais como modularização, redução de manutenção, reutilização, entre outras, também se melhora significativamente o desempenho do sistema, reduzindo seu custo a partir do momento em que o mesmo poderá ser implementado em plataformas de programação gratuita e com outras possibilidades de otimizar seu desempenho como, por exemplo, programação com a utilização de *multi-threading*, programação distribuída e até clusterização.

Assim, foram utilizadas boas práticas de programação e técnicas de engenharia de software e orientação a objetos [38, 39, 40] para se criar o *framework* de implementação do *codec* Open DVC em linguagem orientada a objetos, no caso, em Java [41, 42].

Quanto à escolha da linguagem de programação Java ao invés de outras linguagens orientadas a objetos como C++, Ruby, C#, Python, Perl entre outras, isso se explica pela maior maturidade da arquitetura orientada a objetos da linguagem Java em relação às demais, por causa da utilização por um número

cada vez maior de usuários da linguagem Java, como pode ser visto nas tabelas 4.1 e 4.2, que consideram o índice TIOBE para definição deste ranking [49] e em particular, no caso da nossa aplicação de codificação de vídeo para dispositivos de baixa capacidade computacional, se justifica pela enorme quantidade de dispositivos móveis que hoje em dia dão suporte a esta linguagem, se tornando uma realidade muito viável de ser utilizada numa aplicação comercial de captura e codificação de vídeo para celulares, *tablets*, redes de sensores, rede de monitoramento de desastres e outras aplicações com tais características de processamento.

Tabela 4.1 – Ranking das linguagens de programação mais utilizadas nos últimos vinte e cinco anos

Linguagem de Programação	Posição em 2011	Posição em 2006	Posição em 1996	Posição em 1986
Java	1	1	5	-
C	2	2	1	1
C++	3	3	2	7
PHP	4	4	-	-
Python	5	8	22	-
C#	6	7	-	-
Visual Basic	7	5	3	5
Objective-C	8	44	-	-
Perl	9	6	6	-
Ruby	10	20	-	-
Lisp	13	14	13	3
Ada	20	17	12	2

Fonte: Ref [49]

Tabela 4.2 – *Ranking* das linguagens de programação mais utilizadas nos últimos dois anos

Posição em Jan 2011	Posição em Jan 2010	Linguagem de Programação
1	1	Java
2	2	C
3	4	C++
4	3	PHP
5	7	Python
6	6	C#
7	5	Visual Basic
8	12	Objective-C
9	8	Perl
10	10	Ruby
11	9	JavaScript
12	11	Delphi
13	18	Lisp
14	17	Pascal
15	-	Assembly
16	14	SAS
17	30	Transact-SQL
18	33	RPG (OS/400)
19	20	MATLAB
20	28	Ada

Fonte: Ref [49]

4.1
Projeto Orientado a Objetos do Open DVC

4.1.1
Diagrama de Pacotes

Em relação ao projeto orientado a objetos, o framework que implementa o *codec* Open DVC está dividido em pacotes correspondentes às caixas implementadas no esquema teórico da arquitetura, mostrado no capítulo anterior desta dissertação, com o acréscimo de mais algumas classes dando suporte às funcionalidades necessárias para a implementação de um software de simulação do *codec*.

Dessa forma, tais pacotes correspondem aos subsistemas implementados na arquitetura do Open DVC, conforme mostrado na figura 4.1, onde se representa

um diagrama de pacotes tradicional seguindo as notações definidas na UML 2.0 (*Unified Modeling Language*) [50].

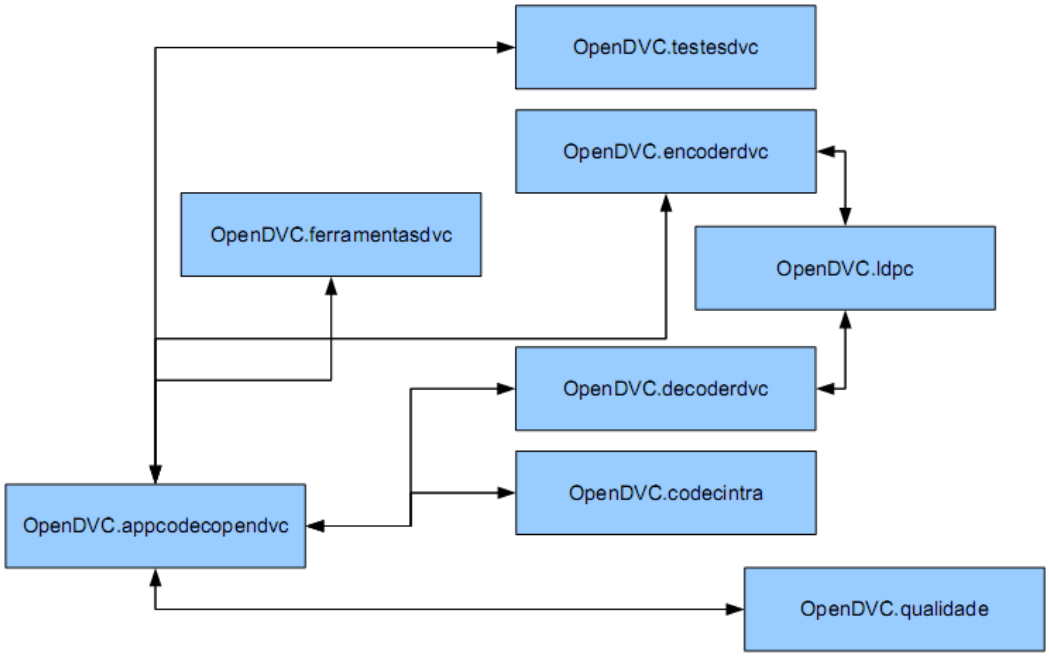

Figura 4.1 – Diagrama de pacotes do *codec* Open DVC

As figuras 4.2 e 4.3 mostram o mesmo diagrama de pacotes, mas, agora, com uma estrutura de blocos apresentando o sentido da colaboração das classes de cada pacote, no caso da figura 4.2, e as camadas com a utilização das classes de cada pacote ao longo do tempo, no caso da figura 4.3.

A ordem de execução da chamada das classes está definida sumariamente na figura 4.3, onde vemos o pacote inicial app*codec*opendvc, que tem a classe responsável por iniciar a aplicação, chamando as demais classes dos outros pacotes e controlando o ciclo de vida do *codec*.

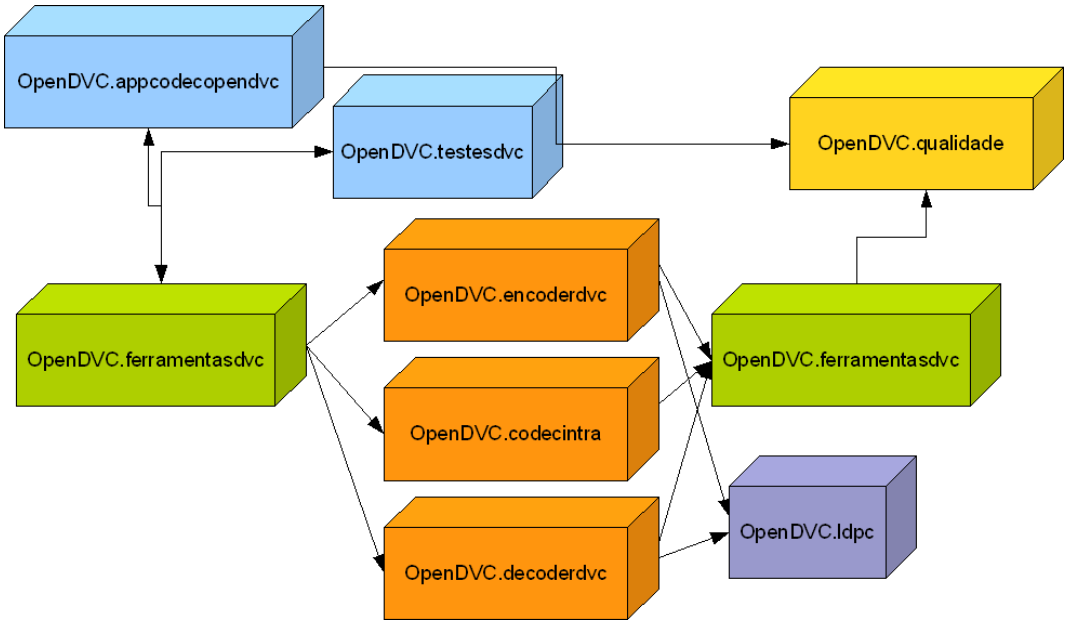

Figura 4.2 – Subsistemas e diagrama de colaboração dos pacotes

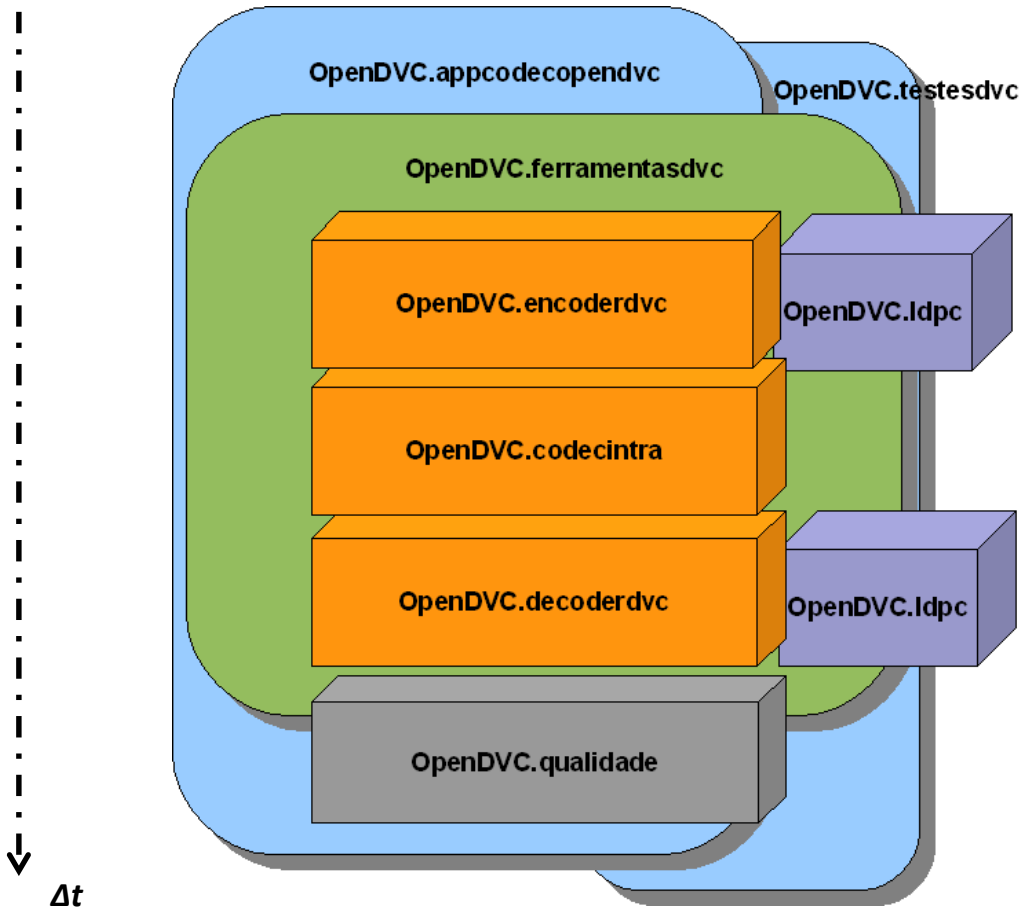

Figura 4.3 – Ciclo de vida e chamada dos pacotes

4.1.2
Diagrama de Classes

O próximo passo na construção do *framework*, à luz da orientação a objetos, é o detalhamento dos pacotes, ou seja, a abertura da caixa-preta ou objetos internos, mostrando a organização das classes do sistema. Essa visão é a mais importante do ponto de vista de documentação O.O., uma vez que define a arquitetura dos objetos atômicos que foram implementados para o funcionamento da ferramenta e é ilustrada de forma completa e integrada na figura 4.4.

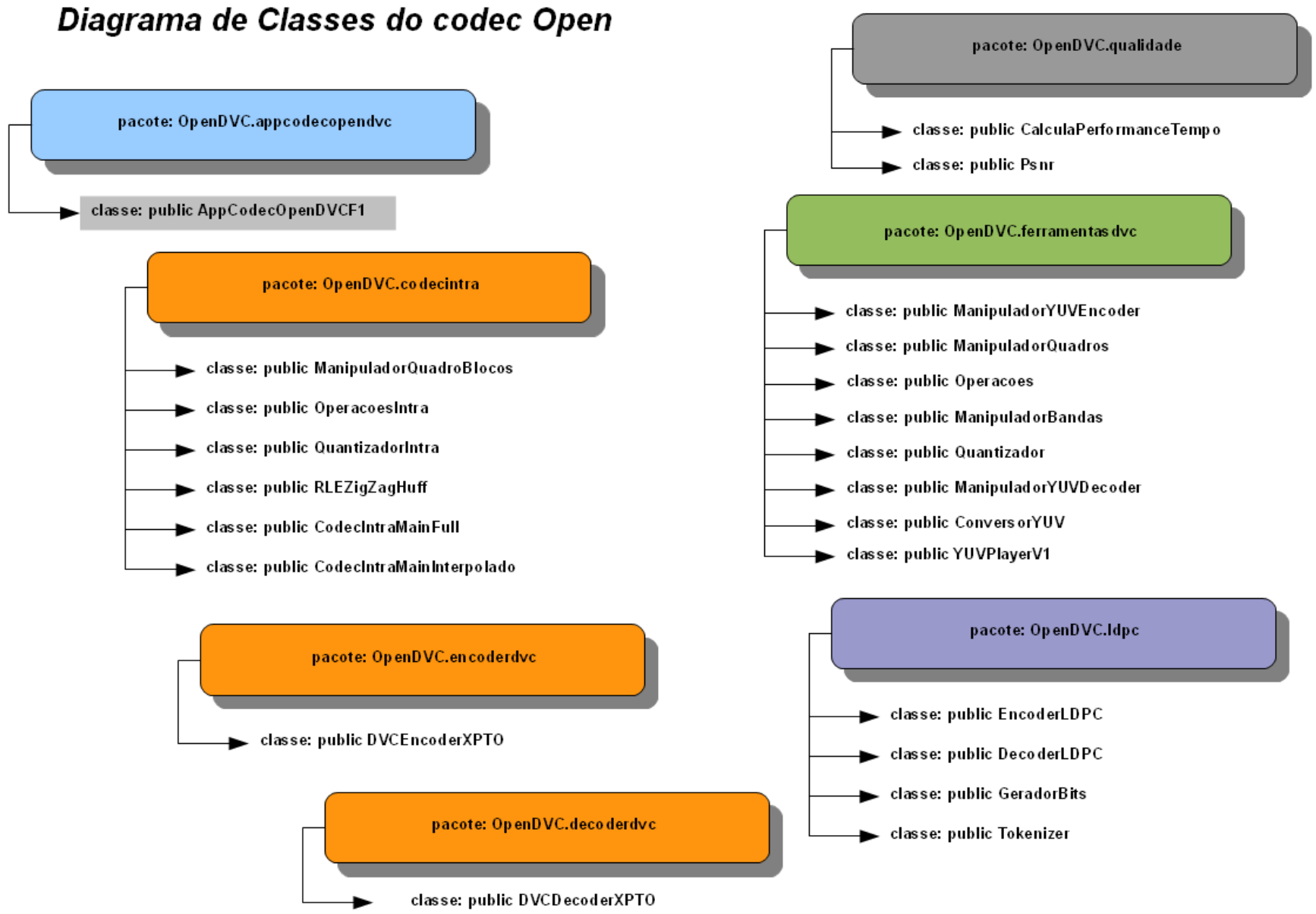

Figura 4.4 - Diagrama de Classes do codec Open DVC

A ordem de detalhamento dos pacotes será a mesma utilizada na figura 4.3, considerando então o ciclo de vida e chamada dos pacotes. De acordo com este critério, abaixo é detalhado o primeiro pacote:

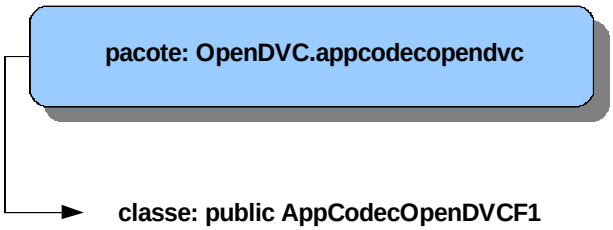

Figura 4.5 – Pacote OpenDVC.app*codec*opendvc e sua classe

Conforme mostrado na figura 4.5, o pacote **OpenDVC.app*codec*opendvc** possui apenas a classe pública **App*Codec*OpenDVCF1** e esta classe possui apenas o método ***main***, conforme mostrado abaixo. Isso ocorre porque, na verdade, a classe **App*Codec*OpenDCVF1** é um exemplo de execução completo do *codec* e é a classe aplicação, o executável, responsável por instanciar todas as demais classes do *framework*, bem como controlar o ciclo de vida de execução da codificação e decodificação, instanciando todas as demais classes utilizadas na ferramenta durante um processo de codificação e decodificação.

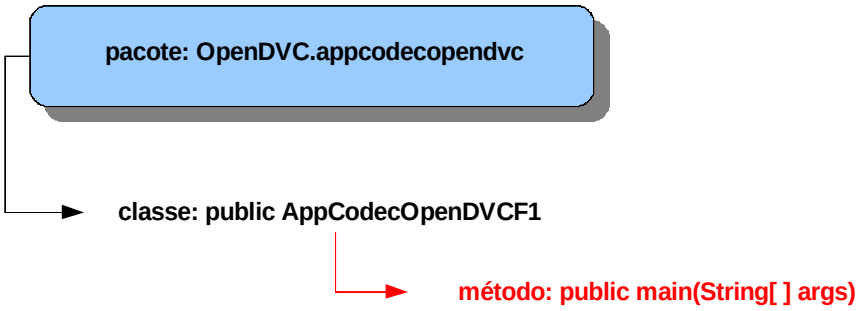

Figura 4.6 – Detalhamento da classe App*Codec*OpenDVCF1

O próximo pacote e consequentemente as próximas classes a serem utilizadas numa execução do *codec* Open DVC estão em **OpenDVC.ferramentasdvc**, onde, com elas, começamos a manipular o arquivo com a sequência de vídeo original e a realizar outras operações de apoio, até

quando, no fim, retornamos a esse pacote para criarmos a nova sequência de vídeo yuv reconstruída. Vejamos abaixo as classes deste importante pacote:

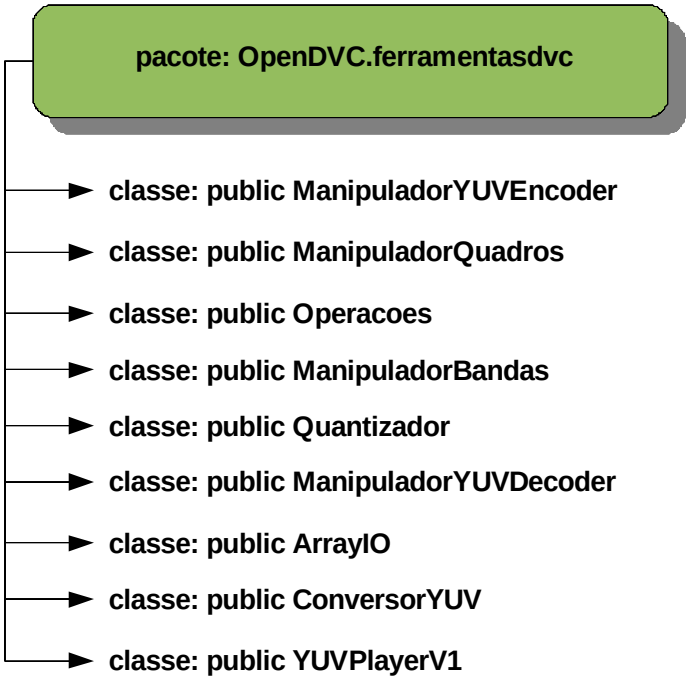

Figura 4.7 – Pacote OpenDVC.ferramentasdvc e suas classes

Agora serão mostrados os detalhes das principais classes do pacote **OpenDVC.ferramentasdvc**:

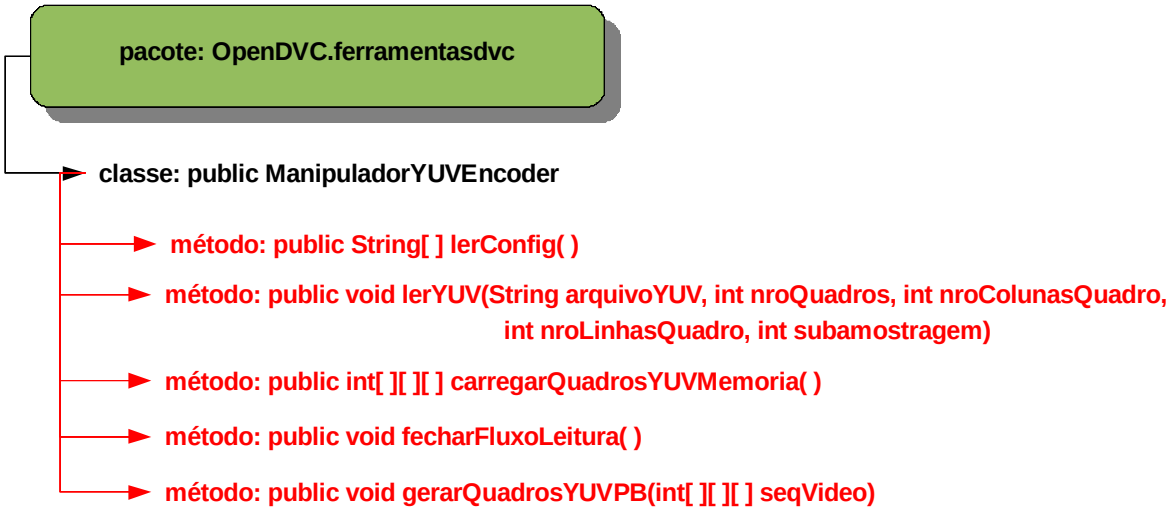

Figura 4.8 – Detalhamento da classe ManipuladorYUVEncoder

A primeira classe do pacote **OpenDVC.ferramentasdvc** a ser utilizada no ciclo de vida do *codec* é a classe pública **ManipuladorYUVEncoder**, que possui métodos de apoio para a leitura e manipulação do arquivo yuv que contém a sequência de vídeo original. Conforme apresentado na figura 4.8, esta classe possui os seguintes métodos com as respectivas funcionalidades:

- **lerConfig**: método responsável por ler o arquivo **DVCConfig.cfg**, que contém os parâmetros de codificação, decodificação, arquivos de entrada e saída, além de outras informações utilizadas por ocasião do funcionamento do *codec*. Posteriormente detalharemos este arquivo de configuração;
- **lerYUV**: método responsável por ler o arquivo yuv original, carregando-o no buffer da máquina virtual do sistema operacional, deixando-o disponível para utilização;
- **carregaQuadrosYUVMemoria**: método responsável por realizar a leitura do arquivo yuv carregado do buffer e fazer a divisão e carregamento quadro a quadro, operação necessária para manipulação e execução da codificação;
- **fecharFluxoLeitura**: método responsável por fazer o fechamento do fluxo de leitura do arquivo yuv que estava no buffer, descarregando-o e deixando-o livre para outras operações;
- **gerarQuadrosYUVPB**: método auxiliar para gerar uma figura no formato .PNG, .BMP ou .JPG de um ou vários quadros da sequência de vídeo, com o objetivo de se realizar algum tipo de análise posterior do quadro ou simplesmente para a geração das figuras, quando o usuário achar necessário.

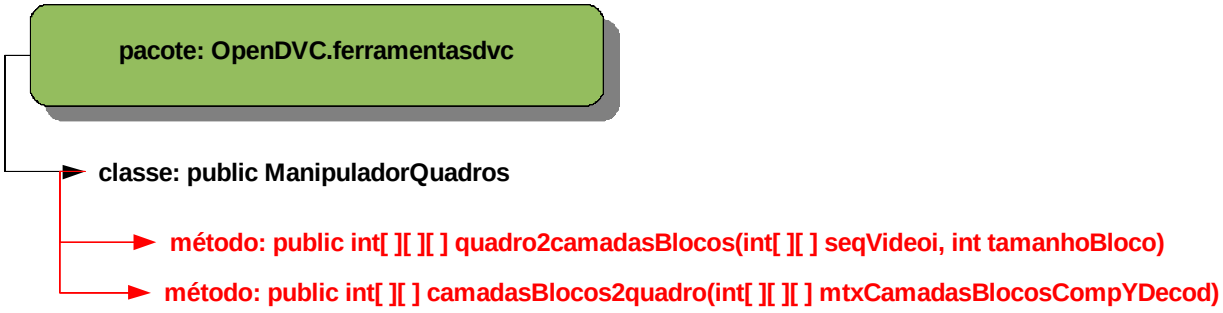

Figura 4.9 – Detalhamento da classe ManipuladorQuadros

A classe **ManipuladorQuadros** do pacote **OpenDVC.ferramentasdvc** é responsável por fazer a manipulação dos quadros, no que se refere à sua divisão em blocos e sua posterior reconstrução, através dos seguintes métodos:

- **quadro2camadasBlocos**: método que recebe um quadro de uma sequência de vídeo e, através do tamanho do bloco recebido como parâmetro, divide-o em blocos e empilha cada um deles, criando camadas indexadas de blocos durante a codificação;
- **camadasBlocos2quadro**: método que executa a operação inversa do método anterior, reconstruindo um quadro a partir de seus blocos depois de serem recuperados na decodificação.

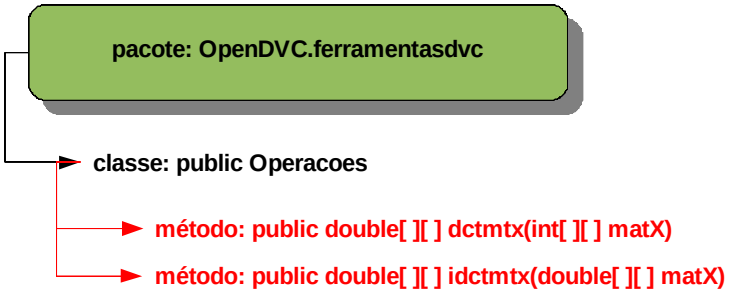

Figura 4.10 – Detalhamento da classe Operacoes

A classe **Operacoes** do pacote **OpenDVC.ferramentasdvc** é responsável por fazer as operações de transformada do *codec*, conforme os métodos abaixo definidos:

- **dctmtx**: método que executa a operação de Transformada Discreta Coseno (DCT) em cada bloco originado da divisão dos quadros da sequência de vídeo, gerando os coeficientes dct do *codec* por ocasião da codificação;
- **idctmtx**: método que executa a operação inversa do método anterior, ou seja, a IDCT, sobre os coeficientes reconstruídos durante a decodificação.

Figura 4.11 – Detalhamento da classe ManipuladorBandas

A classe **ManipuladorBandas** do pacote **OpenDVC.ferramentasdvc** é responsável por fazer as manipulações matriciais que transformam as camadas de blocos em bandas e vice-versa, ou seja, agrupam os coeficientes que estão indexados sob o mesmo índice num bloco, criando um vetor com todas as informações de uma mesma posição através das camadas dos blocos. Isso é feito através dos métodos:

- **gerarBandasDosBlocos**: método que recebe os coeficientes DCT dos blocos de cada quadro, agrupados em camadas indexadas, e une todos os coeficientes da mesma posição (mesmo índice), criando um vetor com os coeficientes de cada posição, as chamadas Bandas de Coeficientes DCT;
- **gerarBlocosDasBandas**: método que recebe as Bandas de Coeficientes DCT reconstruídas na decodificação e divide o vetor, reagrupando as informações nos blocos originais, recriando as camadas de blocos que darão origem ao quadro recuperado.

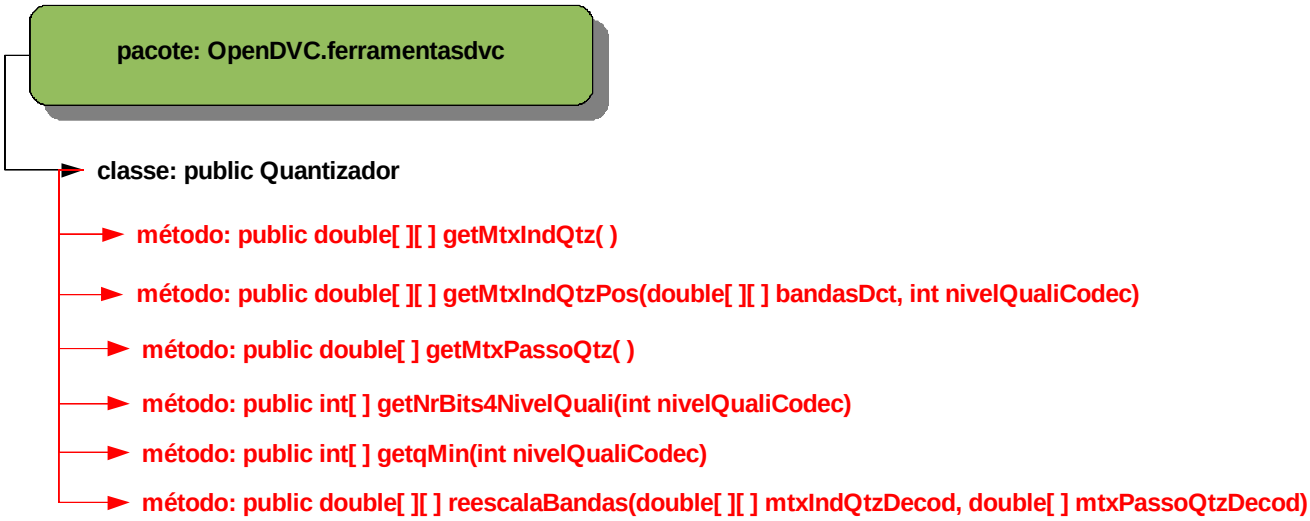

Figura 4.12 – Detalhamento da classe Quantizador

A classe **Quantizador** do pacote **OpenDVC.ferramentasdvc** é uma das classes mais complexas do *codec* e executa uma série de operações relativas à quantização durante a codificação e à reescala (operação inversa da quantização) por ocasião da decodificação. Os métodos utilizados para executar essas operações são:

- **getMtxIndQtz**: método que executa a operação de quantização uniforme para uma determinada banda, através do passo de quantização especificado para esta. Cada banda de coeficientes DCT terá um passo de quantização associado de acordo com o nível de qualidade escolhido;
- **getMtxIndQtzPos**: método que executa a mesma operação de quantização uniforme do método anterior, porém, normalizado para não obtermos índices de quantização negativos. Isso é feito através da soma de um q (índice) mínimo, que é calculado através de um outro método;
- **getMtxPassoQtz**: método que executa a operação para o cálculo do passo de quantização de cada banda, que é o denominador utilizado por ocasião da operação de quantização uniforme. O passo de quantização define a menor ou maior qualidade da informação e consequentemente da banda utilizada e é calculado a partir do nível de qualidade escolhido, que é feito através da

escolha de um dos Vetores de Quantização da Matriz de Qualidade, através de um índice numérico definido no arquivo de configuração DVCConfig.cfg;

- **getNrBits4NivelQuali**: método que calcula a quantidade de bits e que será utilizada para transmitir cada banda de informação de acordo com nível de qualidade escolhido, através da mesma técnica utilizada no método anterior, ou seja, em função do Vetor de Qualidade definido no arquivo DVCConfig.cfg;
- **getqMin**: método que calcula o q mínimo a ser utilizado em cada banda por ocasião do cálculo da matriz índice de quantização;
- **reescalaBandas**: método que executa a operação inversa da quantização, recuperando as bandas DCT reconstruídas, através do índice de quantização e passo de quantização no decodificador.

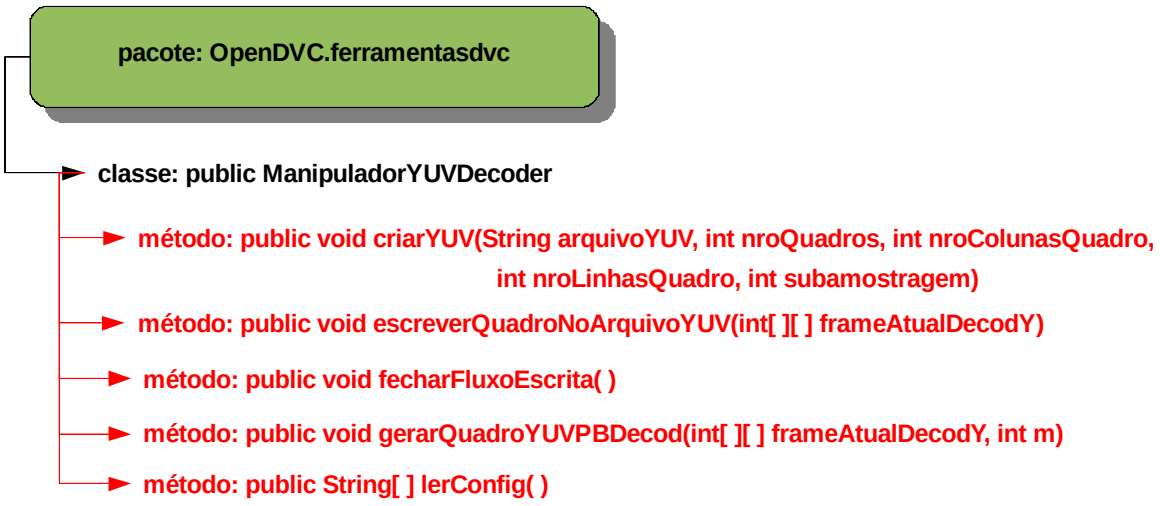

Figura 4.13 – Detalhamento da classe ManipuladorYUVDecoder

A classe **ManipuladorYUVDecoder** do pacote **OpenDVC.ferramentasdvc** faz o papel inverso de classe de mesmo nome para o encoder, servindo de classe auxiliar para a criação e fechamento do arquivo da sequência de vídeo reconstruída. Essa classe possui os métodos:

- **criarYUV**: classe responsável pela criação em buffer do arquivo yuv que irá receber as informações dos quadros reconstruídos,

além de armazenar os parâmetros da sequência de vídeo para serem utilizados quando solicitados;

- **escreverQuadroNoArquivoYUV**: método que cria, grava e organiza a informação binária dos quadros reconstruídos no arquivo yuv final, seguindo o padrão definido para os parâmetros do arquivo yuv em questão;
- **fecharFluxoLeitura**: método responsável por fazer o fechamento do fluxo de leitura do arquivo yuv que estava no buffer, descarregando-o e deixando-o livre para outras operações;
- **gerarQuadroYUVPBDecod**: método auxiliar para gerar uma figura no formato .PNG, .BMP ou .JPG de um ou vários quadros da sequência de vídeo reconstruída, com o objetivo de se realizar algum tipo de análise do quadro reconstruído, normalmente, comparação deste com o seu equivalente original;
- **lerConfig**: método responsável por ler o arquivo **DVCConfig.cfg**, que contém os parâmetros de codificação, decodificação, arquivos de entrada e saída, além de outras informações utilizadas por ocasião do funcionamento do *codec*. Da mesma forma que precisamos dessas informações na codificação, num cenário real, onde a codificação e decodificação são feitas por máquinas diferentes, esse mecanismo deve estar disponível dos dois lados, ou seja, tanto no codificador quanto no decodificador.

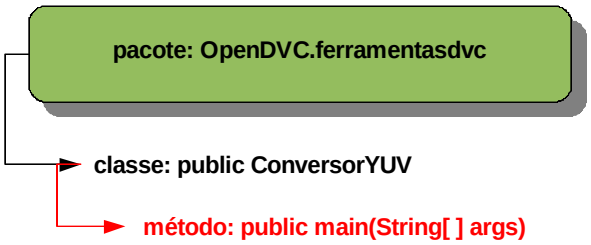

Figura 4.14 – Detalhamento da classe ConversorYUV

A classe **ConversosYUV** do pacote **OpenDVC.ferramentasdvc** é uma ferramenta auxiliar criada para fazer a conversão de uma sequência de vídeo

colorida em uma sequência de vídeo apenas com luminância, que será visualizada como um filme preto e branco. Por ser uma aplicação com apenas essa funcionalidade possui somente o método *main*. Essa operação é interessante quando se deseja fazer análise apenas da luminância de uma determinada sequência de vídeo.

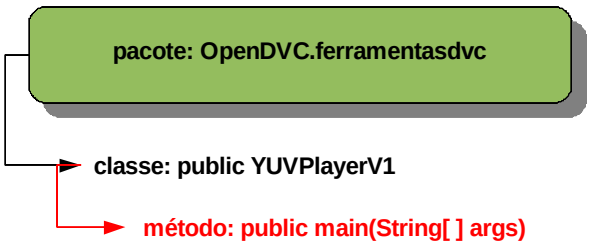

Figura 4.15 – Detalhamento da classe YUVPlayerV1

A classe **YUVPlayerV1** do pacote **OpenDVC.ferramentasdvc** é uma ferramenta auxiliar criada para assistir a uma sequência de vídeo yuv numa janela em qualquer formato, seja a sequência original a ser transformada seja a sequência reconstruída. Por ser uma aplicação com apenas essa funcionalidade possui somente o método main. Essa funcionalidade é importante para que possamos analisar o resulta e fazermos as comparações do funcionamento do *codec*.

Respeitando a ordem de execução do ciclo de vida de chamada dos pacotes do *codec* Open DVC, a próxima classe a ser utilizada é uma classe que "embrulha" as funcionalidades da codificação do módulo DVC do *codec*, a classe **DVCEncoderXPTO** (onde XPTO é um código relativo à versão do *codec* baseado em data), por isso a chamamos de classe *wrapper* em orientação a objetos (embrulhador em português), tendo sua importância por uma questão de organização, principalmente quando se está ainda na fase desenvolvimento do *codec*. Seu pacote, o pacote **OpenDVC.encoderdvc**, tem a finalidade de agrupar e guardar o histórico do desenvolvimento e evolução dessas classes *wrapper*, ao longo da programação do *codec*, servindo como forma de versionamento.

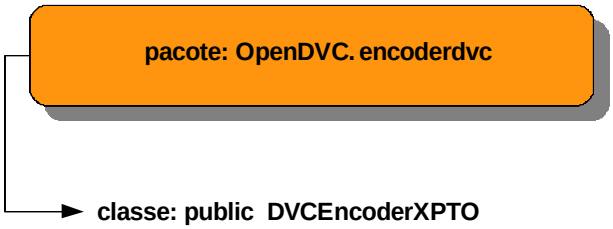

Figura 4.16 – Detalhamento da classe DVCEncoderXPTO no pacote OpenDVC.encoderdvc

Da mesma forma que temos a classe *wrapper* do codificador, que guarda as versões do codificador do módulo DVC, temos a classe *wrapper* do decodificador, a classe **DVCDecoderXPTO**, no pacote **OpenDVC.decoderdvc**, como está ilustrado na figura abaixo, que tem a mesma finalidade.

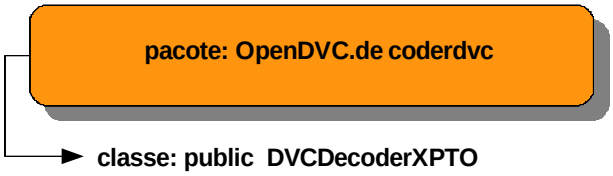

Figura 4.17 – Detalhamento da classe DVCDecoderXPTO no pacote OpenDVC.decoderdvc

Tanto o módulo de codificação DVC quanto o módulo de decodificação DVC, precisam utilizar uma técnica de codificação de canal por causa dos motivos apresentados no capítulo anterior desta dissertação. Devido à complexidade da técnica escolhida para a arquitetura do Open DVC nesta camada, que utiliza código LDPC na codificação e decodificação de canal, as funcionalidades que implementam tal código foram programadas num pacote em separado. Além disso, isso agrega modularidade ao framework, uma vez que podemos "desplugar" esse pacote codificador de canal e "plugar" um código que utiliza outra técnica, por exemplo, um pacote que implemente Códigos Turbo como codificador de canal, uma opção utilizada por algumas arquiteturas, com pouco custo de programação na integração ao *codec*. O pacote que implementa o LDPC tem a seguinte estrutura de classes:

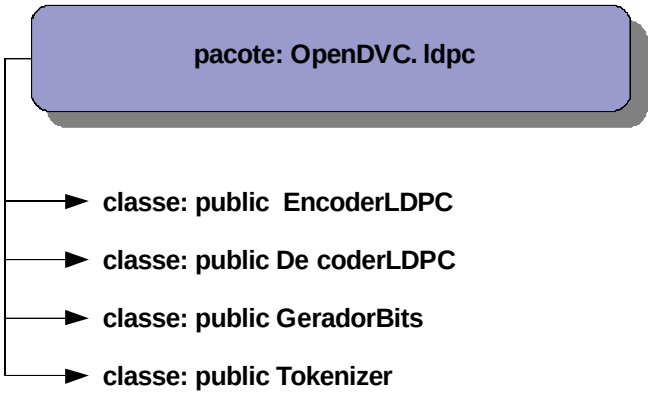

Figura 4.18 – Pacote OpenDVC.ldpc e suas classes

As classes **EncoderLDPC** e **DecoderLDPC** são as principais classes do pacote **OpenDVC.ldpc**, uma vez que são as que implementam os algoritmos de codificação e decodificação do LDPC propriamente ditos. Vamos na figura abaixo ver os métodos que implementam estas técnicas:

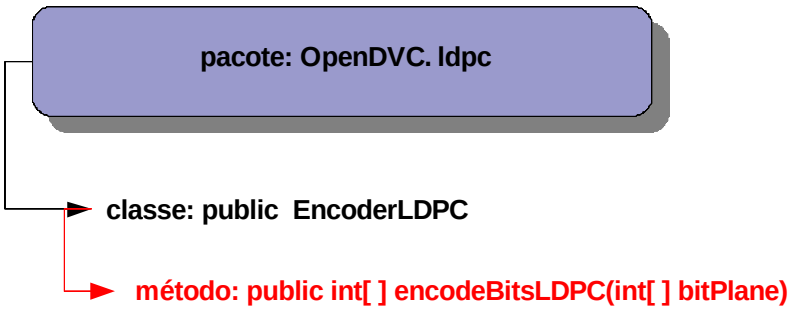

Figura 4.19 – Detalhamento da classe EncoderLDPC

O método **encodeBitsLDPC** da classe **EncoderLDPC** recebe a *bitplane* a ser codificada, executando sobre ela o processo de codificação LDPC e o acumulador, que na verdade classifica nosso código LDPC como acumulado e irregular (isso por conta da matriz geradora do código, que é irregular), gerando as síndromes a serem enviadas ao receptor pelo canal de comunicação.

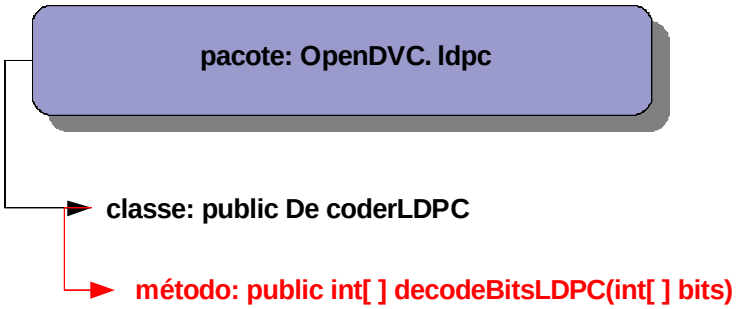

Figura 4.20 – Detalhamento da classe DecoderLDPC

O método **decodeBitsLDPC** da classe **DecoderLDPC** recebe a síndrome a ser decodificada, executando sobre ela o processo de decodificação LDPC, dando origem aos planos de bits reconstruídos, que serão associados à informação lateral gerada para reconstrução do quadro Wyner-Ziv, o mais próximo possível do original.

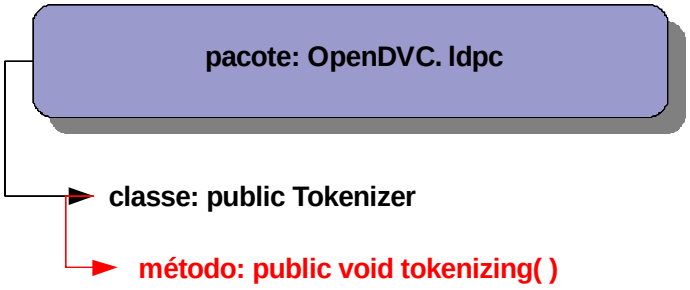

Figura 4.21 – Detalhamento da classe Tokenizer

Apesar das duas classes explicadas anteriormente, **EncoderLDPC** e **DecoderLDPC**, serem as mais importantes do pacote que trata da codificação de canal, por executarem o algoritmo dessas operações, a classe **Tokenizer** não pode ser considerada menos importante, pois é ela que, através do método **tokenizing**, faz a leitura, quebra e decodificação dos parâmetros de controle e da matriz de formação dos códigos, através da leitura dos arquivos de entrada de codificação e decodificação LDPC, os arquivos **EncodLDPC1584.lad** e **DecodLDPC1584.lad**, respectivamente.

O próximo pacote a ser explicado, é o pacote responsável pela codificação intra-frame, que está implementada nas classes do pacote **OpenDVC.*codec*intra**.

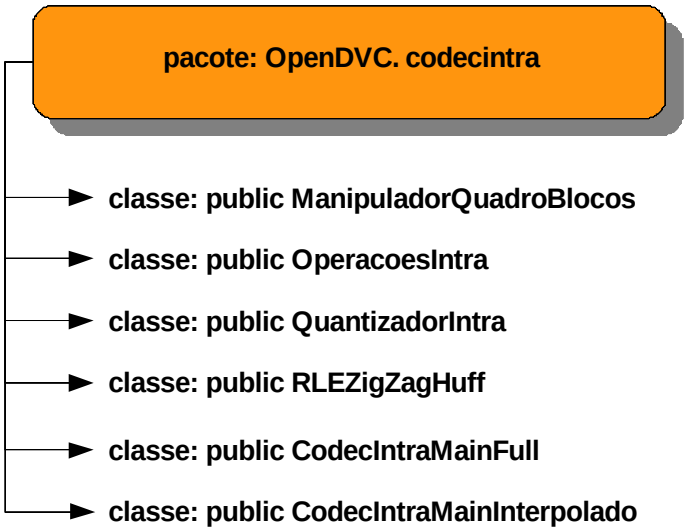

Figura 4.22 – Pacote OpenDVC.*codec*intra e suas classes

A primeira classe do pacote **OpenDVC.*codec*intra** a ser utilizada, por ocasião da codificação, é a classe **ManipuladorQuadroBlocos**, que possui a estrutura ilustrada na figura 4.23:

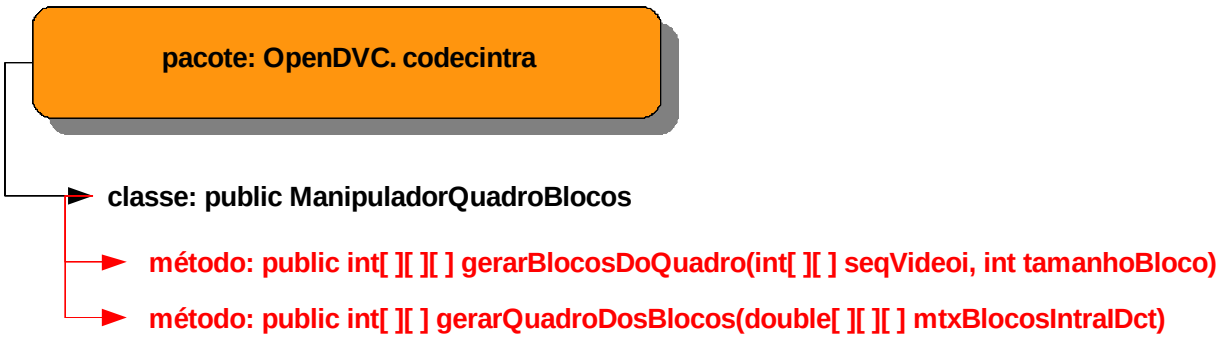

Figura 4.23 – Detalhamento da classe ManipuladorQuadroBlocos

Na classe **ManipuladorQuadroBlocos**, o método **gerarBlocosDoQuadro** tem a finalidade de dividir o quadro corrente em blocos, de acordo com os parâmetros definidos, para serem submetidos à DCT, sendo a primeira operação de codificação propriamente dita executada pelo *codec* intra-frame. O outro método da classe, o método **gerarQuadroDosBlocos**, executa a operação inversa

por ocasião da decodificação, ou seja, dispõe os blocos reconstruídos na ordem correta e os aglutina, gerando o quadro recuperado.

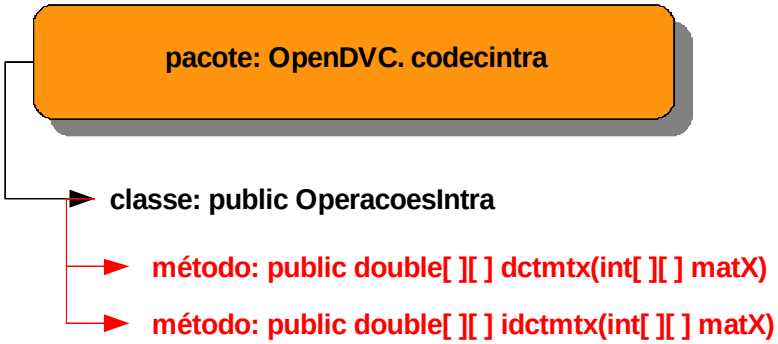

Figura 4.24 – Detalhamento da classe OperacoesIntra

A classe **OperacoesIntra** do pacote **OpenDVC.*codec*intra** é a responsável por receber os blocos divididos a partir dos quadros e submete-los à operação de DCT, por ocasião da codificação, operação executada pelo método **dctmtx**, fazendo também a operação inversa nos blocos de coeficientes reconstruídos, por ocasião da decodificação, através do método **idctmtx**.

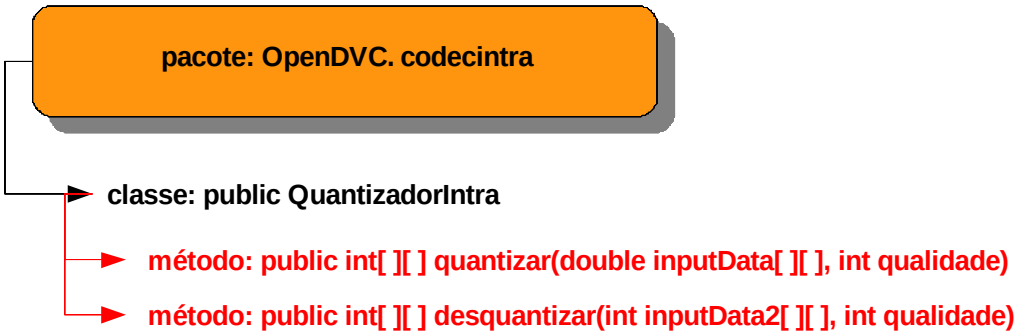

Figura 4.25 – Detalhamento da classe Quantizador

A classe **QuantizadorIntra** do pacote **OpenDVC.*codec*intra** é a responsável por executar as operações de quantização e desquantização do codificador intra-frame, essas operações são executadas atomicamente pelos métodos **quantizar** e **desquantizar**, uma vez que o foco principal do detalhamento da quantização foi feito para o codificador Wyner-Ziv.

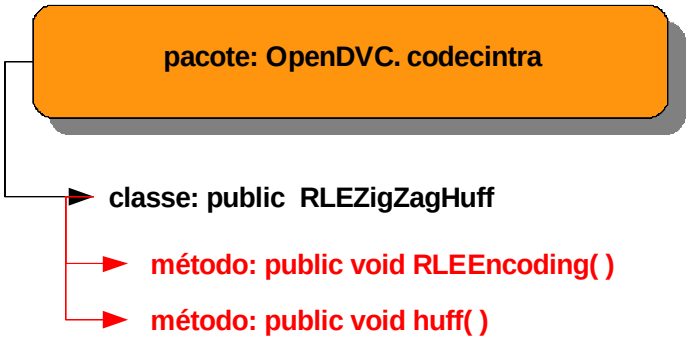

Figura 4.26 – Detalhamento da classe RLEZigZagHuff

A classe **RLEZigZagHuff** do pacote **OpenDVC.*codec*intra** é a responsável por executar as demais operações de codificação intra-frame, ou seja, fazer a codificação e a transmissão pelo canal de comunicações através da técnica de *Run-Length Encoding*, que é uma codificação simples mas eficiente para a compressão deste tipo de informação, uma vez que essa codificação funciona muito bem para dados com grande quantidade de repetição ou redundância, o que é comum entre os quadros de uma sequência de vídeo. Além disso, a forma variante da RLE utilizada na codificação intra implementada, através da técnica de codificação em zig-zag (*zig-zag encoding* em inglês), valoriza as técnicas utilizadas nos passos anteriores, transformada DCT e quantização. Por fim, é associada à técnica RLE uma codificação de Huffman, que otimiza a banda, através de uma compressão baseada na probabilidade de ocorrência dos símbolos.

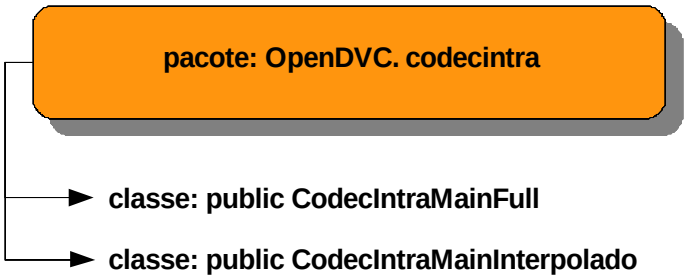

Figura 4.27 – Classes *Codec*IntraMainFull e *Codec*IntraMainInterpolado

As classes ***Codec*IntraMainFull** e ***Codec*IntraMainInterpolado** do pacote **OpenDVC.*codec*intra**, apesar de não implementarem nenhuma funcionalidade da

codificação intra-frame propriamente dita, são importantes porque agregam todas as funcionalidades do referido *codec*, servindo como classe *wrapper*, no caso da ***Codec*IntraMainFull**, podendo servir como aplicação de teste para a codificação unicamente intra do *codec*, além de implementar a função de interpolação, importante na geração da informação lateral, através da classe ***Codec*IntraMainInterpolado**.

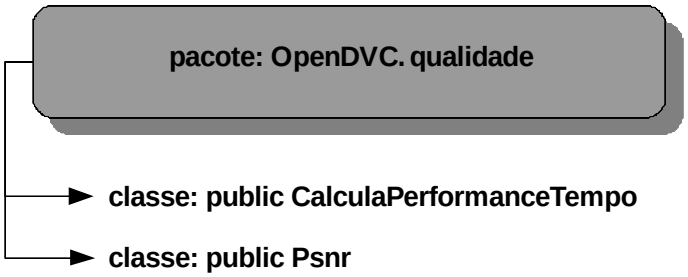

Figura 4.28 – Pacote OpenDVC.qualidade e suas classes

Por fim, as duas últimas classes que iremos explicar, mas nem por isso menos importantes, são as classes **CalculaTempo** e **Psnr** do pacote **OpenDVC.qualidade**, que são responsáveis pelos cálculos de qualidade e estatísticas de desempenho do *codec*. A classe **CalculaTempo** provê os marcadores e faz os cálculos do tempo do processamento de cada parte do *codec*, fornecendo suporte para as estatísticas temporais de . A classe **Psnr** é a responsável por efetuar o cálculo do *mean square error* ou erro médio quadrático em português, entre os pixels dos quadros originais e os dos quadros reconstruídos. Esse índice é utilizado para calcular o PSNR (*peak signal-to-noise ratio*), que é o principal fator de qualidade objetiva utilizado para gerar todas as estatísticas do *codec* Open DVC.

Cabe fazermos uma observação que nesta dissertação não foram apresentadas 100% das classes nem 100% dos métodos das classes apresentadas foram explicados, centralizamos no detalhamento das classes e métodos considerados mais importantes para o framework e para a arquitetura do *codec*, assim, podem ser encontradas no código fonte classes e métodos que não foram explicados.

4.2
Configuração do *codec* Open DVC: Arquivo DVCConfig.cfg

Muito se falou nas seções anteriores e continuaremos a falar nas futuras, das configurações do *codec* Open DVC. Como já foi explicado, essa configuração é feita através de um arquivo texto chamado **DVCConfig.cfg**, o qual iremos explicar.

Uma das vantagens de se utilizar um arquivo de configuração textual, retirando os parâmetros do código fonte, é justamente abrir a possibilidade de modificação destes parâmetros sem a necessidade do ambiente de programação ou gerar uma nova compilação. Se, por um lado, esta abordagem aumenta consideravelmente o trabalho de programação, trás a vantagem de flexibilizar a utilização da ferramenta para o usuário final, uma vez que este pode fazer a qualquer momento e em tempo de execução as modificações desejadas na configuração da ferramenta, podendo alterar a sequência de video a ser codificada, os arquivos e diretórios de saída e até os parâmetros da própria codificação como qualidade, quantidade de quadros e outros.

O arquivo DVCConfig.cfg tem o conteúdo como o exemplo extrato abaixo e iremos explicá-lo posteriormente:

```
300
176
144
18
foreman.yuv
420
15
16
PB
foreman_qcif_m5.dvc
foreman_qcif_m5.intra
foreman_qcif_m5.yuv
DVCEstatisticas.txt
//Parâmetros para o codec (acima listados):
//Linha 1: número de quadros da sequencia de video
//Linha 2: número de colunas por quadro
//Linha 3: número de linhas por quadro
//Linha 4: nível de qualidade do codec Wyner-Ziv (Matriz de qualidade
//daquantização: de 0 a 17), 18 para Matriz Customizada
//para utilizar a Matriz de Quantização Customizada, inserir código 18
//Linha 5: arquivo .yuv com a sequência de vídeo original a ser
//codificada (caminho absoluto ou relativo)
//Linha 6: subamostragem (4:2:0/4:1:1/4:4:4) OBS: por enquanto só está
//implementado para subamostragem 4:2:0
//Linha 7: taxa de quadro (em quadros por segundo)
//Linha 8: tamanho do bloco quadrado da DCT (16 para bloco 4x4/64 para
//bloco 8x8/256 para bloco 16x16) OBS: por enquanto só está implementado
//para bloco 4x4
//Linha 9: padrão de cores da codificação (PB/Color)
//Linha 10: arquivo .dvc com a sequência de vídeo codificada (codificacao
//Wyner-Ziv, quadros WZ - pares) para ser transmitida (caminho absoluto ou
//relativo)
//Linha 11: arquivo .intra com a sequência de vídeo codificada (codificacao
//Intra, quadros chave - impares) para ser transmitida (caminho absoluto ou
//relativo)
//Linha 12: arquivo .yuv com a sequência de vídeo decodificada (caminho
//absoluto ou relativo)
//Linha 13: arquivo .txt com as estatísticas do processo de codificacao e
//decodificacao (caminho absoluto ou relativo)
```

Figura 4.29 – Conteúdo do arquivo de configuração DVCConfig.cfg

Como pode ser visto, o arquivo de configuração possui treze linhas de parâmetros que são lidos e carregados no *codec* no momento de sua execução (em tempo de execução) e as explicações sobre o que significa cada parâmetro é feita logo abaixo, sendo de fácil entendimento para o usuário.

Vamos explicar cada uma das doze linhas do arquivo DVCConfig.cfg:

- linha 1: número inteiro com a quantidade total de quadros da sequência de vídeo yuv a ser codificada;
- linha 2: número inteiro com a quantidade total de colunas de pixels por quadro;
- linha 3: número inteiro com a quantidade total de linhas de pixels por quadro;
- linha 4: número inteiro de 0 a 18 que define o nível de qualidade da codificação Wyner-Ziv, de acordo com os Vetores de Qualidade da Matriz de Quantização, conforme foi explicado no capítulo anterior desta dissertação. Caso insira 18, o usuário está optando por

utilizar uma matriz customizada, que é lida através do arquivo MtxQualiCustom.txt;

- linha 5: arquivo yuv com a sequência de vídeo original a ser codificada, podendo ser utilizado o caminho absoluto ou relativo do arquivo;
- linha 6: código inteiro com três dígitos definindo a subamostragem de croma, sendo 420 para a subamostragem 4:2:0, 411 para a subamostragem 4:1:1 e 444 para a subamostragem 4:4:4. Para saber mais sobre subamostragem, leia o Apêndice A;
- linha 7: número inteiro representando a taxa de quadro em quadros por segundo;
- linha 8: código inteiro representando o tamanho do bloco quadrado da DCT na codificação Wyner-Ziv, sendo 16 para bloco 4x4, 64 para bloco 8x8 e 256 para bloco 16x16;
- linha 9: padrão de cores da codificação, sendo PB para fonte preto e branco e Color para fonte colorida;
- linha 10: arquivo .dvc com a sequência de vídeo (*bitstream*) codificada (codificação Wyner-Ziv, quadros WZ - pares) para ser transmitida (caminho absoluto ou relativo);
- linha 11: arquivo .intra com a sequência de vídeo (*bitstream*) codificada (codificação Intra, quadros chave - ímpares) para ser transmitida (caminho absoluto ou relativo);
- linha 12: arquivo yuv com a sequência de vídeo original decodificada (caminho absoluto ou relativo).
- linha 13: arquivo .txt com as estatísticas do processo de codificação e decodificação (caminho absoluto ou relativo).

4.3
Interface Humano-Computador da Ferramenta de Simulação

As classes que foram implementadas e explicadas nas seções anteriores deste capítulo dão origem à biblioteca orientada a objetos ou *framework* que disponibilizará as funcionalidades para a codificação e decodificação do DVC, bastando para isso, instanciar as classes da biblioteca que contém as funcionalidades desejadas numa aplicação (classe) Java com um método *main*, como fizemos com a classe App*Codec*OpenDVCF1 que foi apresentada como exemplo de aplicação DVC.

A este conjunto de classes com tais funcionalidades implementadas do *codec* Open DVC, dá-se o nome de **Open DVC Lib** e estará contida num pacote chamado **opendvclib.jar**, que pode ser importado para qualquer projeto em Java para dar suporte à codificação DVC.

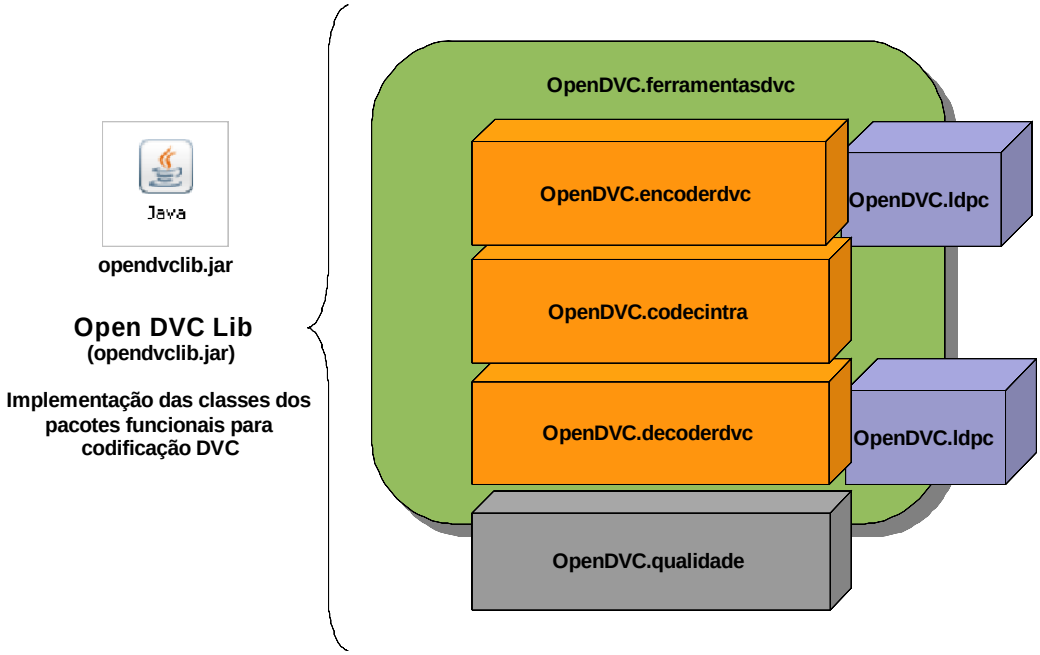

Figura 4.30 – Pacotes funcionais do *codec*, contidos na biblioteca opendvclib.jar

Expandindo a nossa arquitetura, como dissemos, o opendvclib.jar contém somente as classes com as funcionalidades e sozinho não serve para nada, pois precisa de uma aplicação para fazê-lo funcionar (uma classe com método *main* que instancie cada uma das classes com seus métodos, chamando as

funcionalidades desejadas) além de um arquivo yuv contendo a sequência de vídeo original a ser codificada e o arquivo de configuração, importante para ser utilizado pela biblioteca tanto na codificação quanto na decodificação e que também deve conter uma referencia ao arquivo yuv citado. A este conjunto da opendvclib.jar mais a aplicação que instancia as funcionalidades (nossa *App* com um método *main*), mais o arquivo yuv original, mais o arquivo de configuração DVCConfig.cfg daremos o nome de **Open DVC Lab v1.0**, que é a versão completa da ferramenta de simulação e avaliação de codificação DVC.

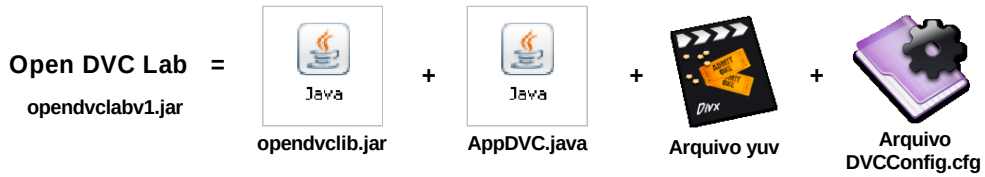

Figura 4.31 – Conteúdo do aplicativo opendvclabv1.jar

Nesta versão 1.0 da ferramenta, todas as operações são feitas via linha de comando ou pela chamada ao pacote executável **opendvclabv1.jar**, que nada mais é do que um diretório compactado com todos os componentes do Open DVC Lab v1.0, mas já configurado para funcionar como um executável, podendo o usuário alterar o arquivo yuv original e os parâmetros de configuração como queira. Além disso, esse pacote completo da ferramenta v1.0 possui um arquivo readme.txt, com a finalidade de orientar o usuário sobre a utilização da ferramenta de simulação e avaliação.

O objetivo final desta dissertação é chegar à versão 2.0 da ferramenta, o que se concretizará com a criação da ferramenta **Open DVC Lab v2.0**, que executa as mesmas operações da versão 1.0 através do pacote **opendvclab2.jar**, porém com as escolhas dos arquivos e definições dos parâmetros através de uma interface gráfica, facilitando a interação e experiência do usuário.

Figura 4.32 – Conteúdo do aplicativo opendvclabv2.jar, que possui interface gráfica

Figura 4.33 – Interface gráfica da ferramenta opendvclabv2.jar

Como será apresentado no capítulo que trata de trabalhos futuros, a ideia é expandir futuramente a ferramentas para versões 3.0 e 4.0, onde o DVC será expandido para outras possibilidades tanto em relação aos parâmetros, quanto às funcionalidades, podendo o usuário alterar, por exemplo, o tipo de codificação de canal, de LDPC para Códigos Turbo. Outros exemplos interessantes para se fazer o chaveamento e comparação de desempenho e qualidade, seriam a possibilidade de alterar a transformada DCT para uma transformada Wavelet ou trocar o tipo de codificação intra-frame, entre outras operações, tornando-se a ferramenta muito mais do que uma implementação de DVC, mas uma ferramenta inicial para o estudo de processamento de vídeo, quando então passará a se chamar **Open ViP Lab** ao invés de **Open DVC Lab**, que seria um acrônimo para **Open Video Processing Lab**.

4.4 Conclusões

Este capítulo apresentou a arquitetura computacional do *codec* Open DVC, baseada nas notações e conceitos de Orientação a Objetos.

Atualmente, é praticamente unanimidade a escolha por arquiteturas orientada a objetos, uma vez que esse esquema traz muitas vantagens para a programação da aplicação, seja do ponto de vista de implementação, de expansão

ou de manutenção. Assim, concluiu-se que é um fator positivo a implementação ter sido feita seguindo este paradigma.

Uma constatação importante é que a escolha da linguagem Java como linguagem para implementação se mostra promissora, uma vez que os dispositivos candidatos a utilizarem esse *codec* possuem bom suporte à linguagem. Além disso, o número cada vez maior de programadores desta linguagem em detrimento de outras, pode estimular e alavancar o uso da ferramenta como forma de aprender e expandir conhecimento sobre *codecs*.

Outro ponto observado importante é a forma modular como o projeto e implementação foram feitos, permitindo a fácil troca, expansão ou manutenção dos módulos da ferramenta.

Por último, cabe ressaltar as funcionalidades implementadas pelo programa, que pode se tornar uma importante ferramenta tanto de estudo e implementação do DVC como de codificação de vídeo propriamente dita, uma vez que vários de seus módulos são implementados por vários tipos de esquemas de *codecs*, como os módulos das transformadas, da quantização e da codificação de canal, possibilitando fácil expansão e utilização por parte dos usuários, como foi mostrado na seção anterior.