DESENVOLVIMENTO DE APLICATIVOS MULTIMÍDIA USANDO JAVA MEDIA FRAMEWORK (JMF)

Leonardo BARRETO CAMPOS (1); Diego BOMFIM ANDRADE (2)
(1) Instituto Federal de Educação Tecnológica da Bahia – Campus de Vitória da Conquista, Av. Amazonas 3150 - Zabelê - Vitória da Conquista, e-mail: leonardobcampos@ifba.edu.br
(2) Instituto Federal de Educação Tecnológica da Bahia – Campus de Vitória da Conquista, Av. Amazonas 3150 - Zabelê - Vitória da Conquista, e-mail: diegoandrade.eng@gmail.com

RESUMO

As tecnologias que dão suporte ao processamento e transmissão de dados multimídia pelas redes de computadores permitem o desenvolvimento de sistemas cada vez mais poderosos e utilitários. Nesse sentido, este artigo tem como objetivo uma abordagem prática dos fundamentos do Paradigma Orientado a Objetos aplicados na Linguagem de Programação Java com o propósito de desenvolver aplicativos usando a API Java Media Framework (JMF) e explorar suas principais funcionalidades. Como objetivo específico o artigo prevê a implementação de um software capaz de executar mídias em formatos diferentes oriundas de diversas fontes de dados, entre elas: base de dados local, base de dados distribuída ou câmeras Web.

Palavras-chave: POO, Java, JMF e Sistemas Multimídia.

1 INTRODUÇÃO

A evolução dos sistemas de computação (capacidade de armazenamento, novas interfaces gráficas, velocidade das redes, etc) que dão suporte ao processamento ou troca de dados, permite cada vez mais o desenvolvimento de sistemas que processe ou transmita informações representadas em outras mídias além da textual como, por exemplo, áudio e vídeo (TANENBAUM 2003), (KUROSE 2003) e (TORRES 2001). Esse conjunto de hardware e software que possibilita: criar, manipular, armazenar, transmitir e exibir informações de diversas naturezas como: texto, gráficos, imagens estáticas, voz (áudio) e vídeo, é conhecido como Sistemas Multimídia (LU 1996).

As aplicações multimídia são comumente divididas em dois grandes grupos: multimídia distribuída e multimídia não-distribuída. As aplicações distribuídas são aquelas executas com o auxílio das redes de computadores. Dessa forma, as informações estão distribuídas em redes de computadores, podendo estar armazenada em diversos servidores com o objetivo de compor uma única apresentação multimídia, conforme Figura 1.

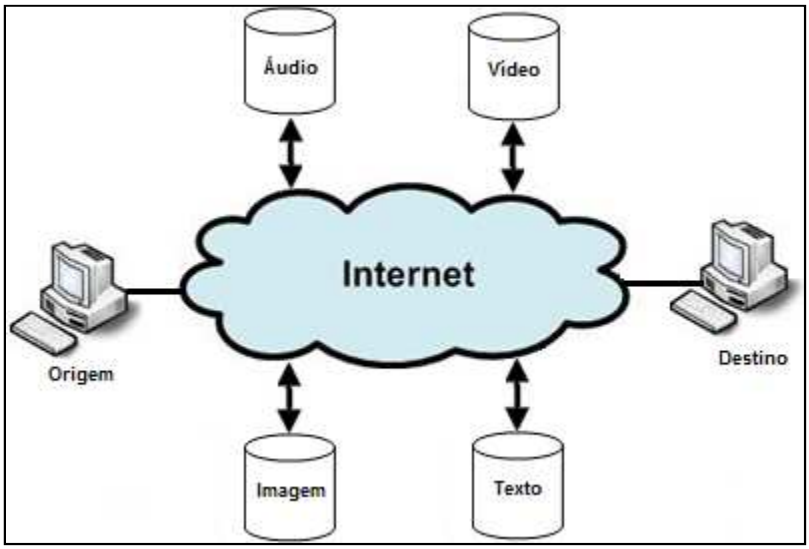


Figura 1 – Cenário clássico de aplicações multimídia distribuídas

Nas aplicações multimídia não-distribuídas, também conhecidas como *stand-alone*, todos os dados para a manipulação e apresentação dos dados multimídia encontram-se numa única máquina.

Dessa forma, diante disso do crescimento das aplicações multimídia nos sistemas de computação e as variedades de mídias processadas e transmitidas pelas redes de computadores, o presente trabalho apresenta uma abordagem para desenvolvimento de aplicações multimídia distribuídas e não-distribuídas utilizando uma API (*Applications Programming Interfaces*) da linguagem de programação Java, denominada JMF (*Java Media Framework).*

Segundo Horstmann et al. (2002), JMF "é uma coleção de classes que permite a captura e visualização de dados multimídias em aplicações e *applets* Java". Usando a API JMF é possível criar aplicativos Java que capturaram, produzem e editam os tipos de mídias de áudio e vídeo mais populares do mercado, por exemplo: MPEG, AVI, MP3, etc.

Para obter uma melhor compreensão do trabalho apresentado, o artigo está dividido da seguinte forma: a seção 2 faz uma análise dos principais fundamentos definidos pelo Paradigma Orientado a Objetos aplicados na linguagem de programação Java e que serão utilizados no desenvolvimento de aplicativos multimídia através da API JMF. Serão apresentados exemplos e aplicações práticas para facilitar a compreensão. Na seção 3 os conceitos básicos da API *Java Media Framework* (JMF). A seção 4 consiste na aplicação prática de todo o referencial teórico visto nas seções anteriores. Por fim, a seção 5 apresenta as considerações finais e agradecimentos.

2 FUNDAMENTOS DO PARADIGMA ORIENTADO A OBJETOS – POO

Segundo (SANTOS 2003) Programação Orientada a Objetos ou, abreviadamente, POO, é um paradigma de programação de computadores onde se usam classes e objetos, criados a partir de modelos para representar e processar dados usando programas de computadores. As seções a seguir apresentarão os principais fundamentos do paradigma orientado a objetos que serão utilizados no desenvolvimento de aplicativos multimídia.

2.1 Modelos

Modelos são representações simplificadas de pessoas, itens, tarefas, processos, etc usados comumente no seu dia-a-dia, independente do uso de computadores.

Para exemplificar o uso de modelos em tarefas comuns, considere uma Instituição de Ensino Superior que armazena dados acadêmicos dos seus alunos através de uma fica de matrícula. Este modelo pode ser considerado contendo as principais informações de identificação do aluno conforme mostra a Figura 2.

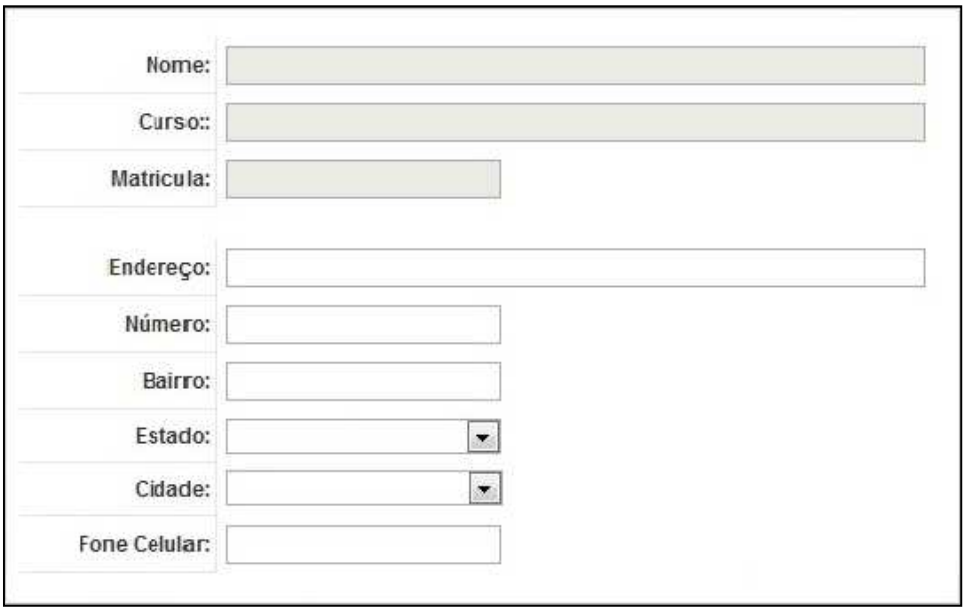


Figura 2 – Ficha cadastral de alunos

O modelo apresentado na Figura 2 representa dados ou informações relevantes à abstração do mundo real. Dessa forma, no registro acadêmico de um aluno, dados como altura, cor dos olhos e peso são irrelevantes e não devem ser representados pelo modelo (domínio) em questão.

Além dos dados associados aos alunos, um modelo comumente contém operações ou procedimentos associados a ele. Essas operações são listas de comandos que processarão os dados contidos no próprio modelo. Entre outras operações que podem ser feitas sobre os dados cadastrados dos alunos estão: alteração do turno, obtenção do telefone celular, atualização do endereço, etc. A Figura 3 mostra os dados e as operações deste modelo apenas para os três primeiros campos do formulário.

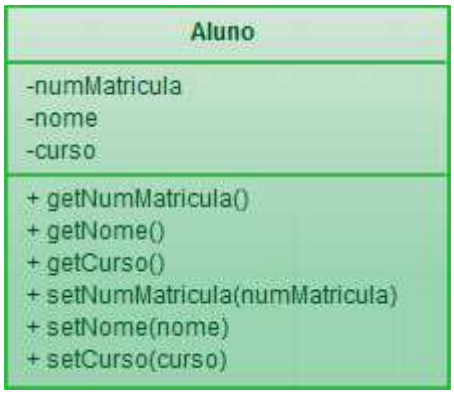


Figura 3 – Modelo Aluno, seus dados e métodos

Uma vez definido o modelo é possível codificá-lo na linguagem de programação Java através de classes, conforme apresentado na próxima seção.

2.2 Classes e Objetos

Em Java, a unidade de programação é a classe da qual eventualmente objetos são instanciados (criados) (DEITEL & DEITEL 2003). Os grupos de ações que realizam alguma tarefa são chamados de métodos. Portanto, cada classe contém dados, bem como o conjunto de métodos que manipulam esses dados. A Figura 4 mostra a implementação da classe Aluno na linguagem Java.

```
1   public class Aluno {
2       private String numMatricula;
3       private String nome;
4       private String curso;
5
6       public Aluno()
7       {
8           curso = "Bacharelado em Ciência da Computação";
9       }
10      public String getCurso() {
11          return curso;
12      }
13      public void setCurso(String curso) {
14          this.curso = curso;
15      }
16      public String getNome() {...}
19      public void setNome(String nome) {...}
22      public String getNumMatricula() {...}
25      public void setNumMatricula(String numMatricula) {...}
28  }
```

Figura 4 – Código-fonte da Classe Aluno

A Figura 4 apresenta uma simples definição da classe Aluno. Uma classe em Java é sempre declarada com a palavra-chave **`class`** seguida do nome da classe (linha 1). Nas linhas 2 a 4, são declarados três dados vinculados a classe Aluno capazes e armazenar sequências de caracteres (**`strings`**), são eles: `numMatricula`, `nome` e `curso`.

Em muitos casos será desejável que os dados não possam ser acessados ou usados diretamente, mas somente através das operações cuja especialidade será a manipulação dos dados. A capacidade de ocultar dados dentro das classes, permitindo que somente operações especializadas ou dedicadas manipulem os dados ocultos (privados) chama-se encapsulamento. A palavra-chave em Java que efetua essa modificação sobre os dados é **`private`**.

Porém, a maior parte do código-fonte que especifica a classe Aluno é composta por métodos. Conforme definição anterior, métodos são grupos de ações que realizam alguma tarefa sobre os dados de uma classe. Os métodos da classe Aluno estão definidos nas linhas 6 a 27, são eles: **`Aluno`**, **`getCurso`**, **`setNome`**, **`getCurso`**, **`setCurso`**, **`getNumMatricula`** e **`setNumMatricula`.**

O método que possui o mesmo nome da classe (linhas 6-9) é o método conhecido como *construtor* da classe. Um construtor é um método especial que inicializa as variáveis de um objeto dessa classe, ou seja, o método construtor é chamado quando um programa instancia um objeto da classe. Na classe Aluno, o construtor inicia o campo curso com a string “Bacharelado em Ciência da Computação”. Dessa forma, para cadastrar um aluno com outro curso será necessário alterar o campo curso deste objeto.

Os métodos iniciados em get e set terão a finalidade de obter o dado gravado na variável e alterar este dado, respectivamente. De acordo com essa premissa os métodos **`getCurso`**, **`getNome`** e **`getNumMtaricula`** terão comportamento semelhante: retornar os caracteres do campo correspondente. De forma análoga, os métodos **`setCurso`**, **`setNome`** e **`setNumMatricula`** terão o mesmo comportamento: receber um novo dado, passado como parâmetro do método, que atualizará o campo correspondente.

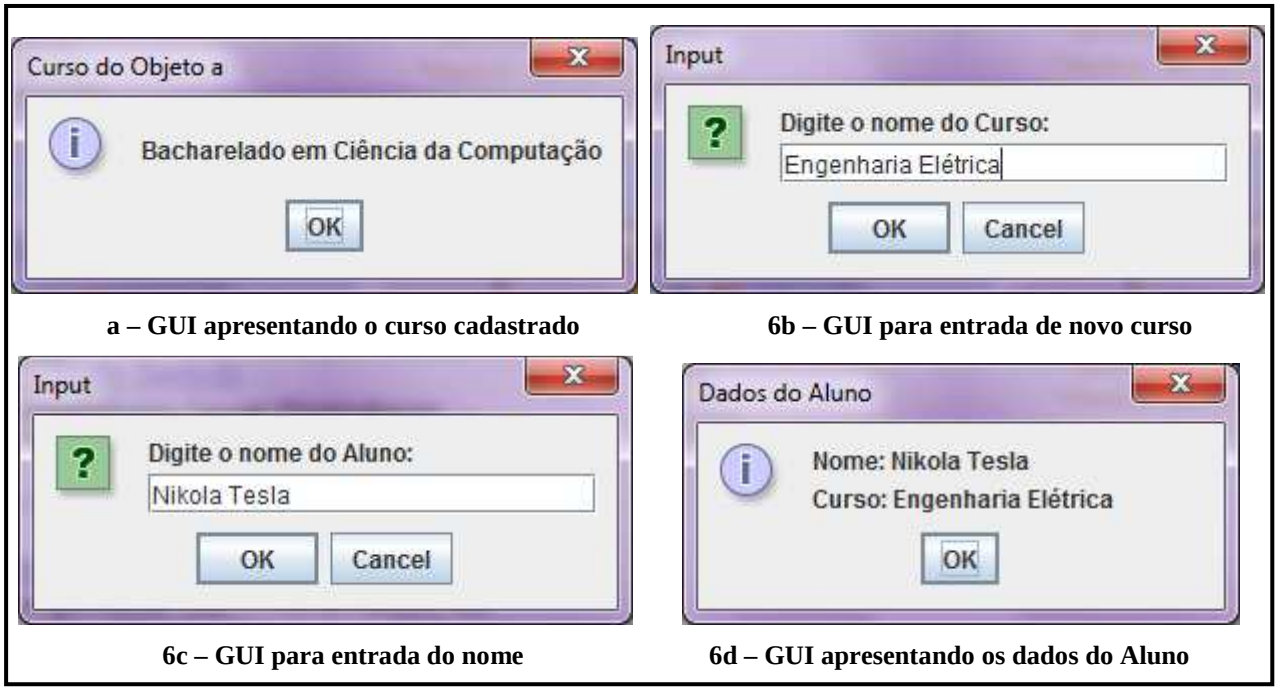
Por fim, para testar as funcionalidades da classe Aluno é necessária a criação de uma classe que contenha o método **`main`**. A Figura 5 mostra o código-fonte da classe **`Main`** que testará as funcionalidades da classe Aluno.

```
1  import javax.swing.JOptionPane;
2
3  public class Main {
4
5      public static void main(String[] args) {
6
7          Aluno a = new Aluno();
8
9          JOptionPane.showMessageDialog(null, a.getCurso(), "Curso do Objeto a", 1);
10
11         a.setCurso(JOptionPane.showInputDialog("Digite o nome do Curso:"));
12
13         a.setNome(JOptionPane.showInputDialog("Digite o nome do Aluno:"));
14
15         JOptionPane.showMessageDialog(null, "Nome: "+a.getNome()+
16                                             "\nCurso: "+a.getCurso(),
17                                             "Dados do Aluno", 1);
18     }
19
20 }
```

Figura 5 – Programa Main.java que testa a Classe Aluno

Toda aplicação Java deve conter o método **`main`** (linha 5), justamente, onde as aplicações Java iniciam sua execução. Dento do escopo do método **`main`** o operador **`new`** (linha 7) cria um objeto (instância) da classe

Aluno alocando a memória dinamicamente suficiente para armazenar os dados do objeto. No momento em que o *objeto a* é criado seu construtor inicializa a variável `curso` inserindo a string "Bacharelado em Ciência da Computação". Em seguida são usados dois métodos contidos na classe **`JOptionPane`**, são eles: **`showMessageDialog`** e **`showInputDialog`**; ambos abrirão GUI (*Graphical User Interface*), interfaces gráficas para saída e entrada de dados respectivamente, conforme Figura 6.



a – GUI apresentando o curso cadastrado

6b – GUI para entrada de novo curso

6c – GUI para entrada do nome

6d – GUI apresentando os dados do Aluno

Figura 6 – Teste da classe Aluno

A classe **`JOptionPane`** definida pela Java está inserida no pacote **`javax.swing`**, importado na linha 1 através do parâmetro **`import`**. Este parâmetro ajuda o compilador a localizar as classes usadas pelo programa. Para cada classe usada da API Java deve-se indicar (importar) o pacote no qual se encontra aquela classe, só assim, o compilador assume a forma correta de usar as classes e métodos do pacote importado.

Em suma, de acordo com o código-fonte do programa **`Main.java`** e as imagens da Figura 6, tem-se a seguinte sequência de execução: (i) O *objeto a* é criado e inicializado pelo construtor da classe Aluno, (ii) Um tela com a confirmação dessa inicialização é exibida, (iii) Uma caixa de texto para entrada de dados é exibida solicitando o nome do curso, (iv) Uma caixa de texto para entrada de dados é exibida solicitando o nome do aluno, (v) Uma caixa de texto para exibição de dados finaliza o programa mostrando os dados digitados.

2.3 Herança

Uma das características mais interessantes de linguagens de programação orientadas a objetos é a capacidade de facilitar a reutilização de código – o aproveitamento de classe e seus métodos que já estejam escritos e que já tenham o seu funcionamento testado e comprovado. Reutilização de código diminui a necessidade de escrever novos métodos e classes, economizando o trabalho do programador e diminuindo a possibilidade de erros.

Como exemplo, considere um aluno de um curso universitário, que pode ser modelado por uma instância da classe Aluno definido nas Figuras 3 e 4. Um aluno de pós-graduação na poderia ser modelado por uma instância da mesma classe, uma vez que para alunos de pós-graduação deve-se manter dados sobre o título da tese e o nome do orientador, por exemplo.

A solução dada por linguagens de programação orientada a objetos é criar uma nova classe, por exemplo, **`AlunoPosGraduacao`**, que contém os campos e métodos da classe Aluno e os campos e métodos adicionais que diferenciam o uso e o comportamento das duas classes. Existem duas formas básicas de

reutilização de classes em Java: composição e herança. Como o objetivo deste trabalho não utiliza classes oriundas de composição, o direcionamento será apenas para a herança, conforme mostra a Figura 7.

```
1   public class AlunoPosGraduacao extends Aluno{
2
3       private String tituloDissertacao;
4       private String nomeOrientador;
5
6       public String getNomeOrientador() {
7           return nomeOrientador;
8       }
9       public void setNomeOrientador(String nomeOrientador) {
10          this.nomeOrientador = nomeOrientador;
11      }
12      public String getTituloDissertacao() {...}
15      public void setTituloDissertacao(String tituloDissertacao) {...}
18
19  }
```

Figura 7 – Classe AlunoPosGraduacao herdando a Classe Aluno

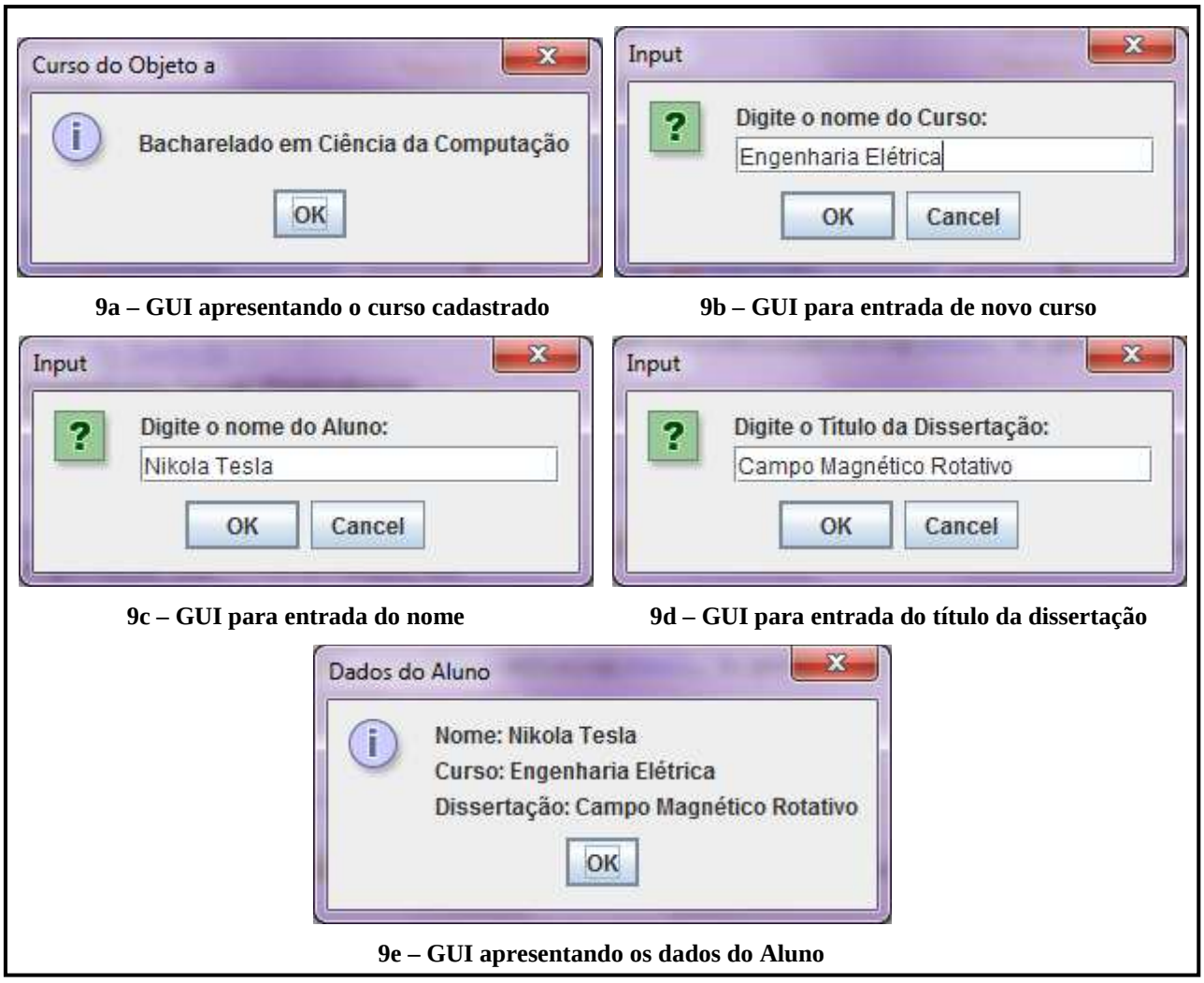
De acordo com o código-fonte apresentado na Figura 7, a linguagem de programação Java possibilita herança através da palavra-chave **`extends`**. Na relação de herança apresentada pelo código-fonte a classe **`Aluno`** aparece como superclasse ou classe base, enquanto que a classe **`AlunoPosGraduacao`** é chamada de subclasse ou classe derivada. Dessa forma, ao usar herança os dados e métodos da superclasse (classe base) são adicionados a subclasse (classe derivada), conforme apresentado na Figura 8.

```
1   import javax.swing.JOptionPane;
2
3   public class Main {
4
5       public static void main(String[] args) {
6
7           AlunoPosGraduacao b = new AlunoPosGraduacao();
8
9           JOptionPane.showMessageDialog(null, b.getCurso(), "Curso do Objeto b", 1);
10
11          b.setCurso(JOptionPane.showInputDialog("Digite o nome do Curso:"));
12
13          b.setNome(JOptionPane.showInputDialog("Digite o nome do Aluno:"));
14
15          b.setTituloDissertacao(JOptionPane.showInputDialog("Digite o Título da Dissertação:"));
16
17          JOptionPane.showMessageDialog(null, "Nome: "+b.getNome()+
18                                              "\nCurso: "+b.getCurso()+
19                                              "\nDissertação: "+b.getTituloDissertacao(),
20                                              "Dados do Aluno", 1);
21      }
22
23  }
```

Figura 8 – Programa Main.java que testa a Classe AlunoPosGraduacao

Ao criar o **`AlunoPosGraduacao`** b (linha 7), implicitamente, os dados e métodos para manipulação dos campos nome, curso e matrícula definidos na classe Aluno estarão disponíveis para a classe derivada. É

possível verificar a existência da herança na Figura 9 que apresenta o teste da classe **`AlunoPosGraduacao`**.



9a – GUI apresentando o curso cadastrado

9b – GUI para entrada de novo curso

9c – GUI para entrada do nome

9d – GUI para entrada do título da dissertação

9e – GUI apresentando os dados do Aluno

Figura 9 – Teste da classe AlunoPosGraduacao

Diante da sequência de execução apresentada pelas imagens contidas na Figura 9 é possível verificar a herança dos dados definidos na classe Aluno.

Considerando a compreensão e aplicação dos conceitos apresentados na seção 2, o leitor possuirá o fundamento para desenvolver aplicativos multimídia usando a API JMF. O objetivo do trabalho justifica a ausência de conteúdos fundamentais da POO, como por exemplo: classes abstratas e interfaces, polimorfismo, coleções de objetos, etc.

3 UTILIZAÇÃO DA API JMF

Os aplicativos multimídia têm passado por um crescimento considerável nos últimos anos, como pode ser observado pela enorme quantidade de sites disponíveis na Internet que apresentam este tipo de conteúdo. Os Sites da Web, que antes eram simples páginas HTML, foram se transformando em experiências intensas de multimídia, proporcionadas pelos avanços tecnológicos tanto de hardwares quanto de softwares.

Reconhecendo a necessidade de os aplicativos Java suportarem recursos de áudio e vídeo digitais, a Sun Microsystems, a Intel e a Silicon Graphics trabalharam juntas para produzir uma API para multimídia que é conhecida como Java Media Framework (JMF). Segundo (SUN MICROSYSTEMS 1998), “com o JMF pode-se facilmente criar *applets* e aplicações que apresentam, capturam, manipulam e armazenam diferentes tipos de mídias baseadas em tempo”.

Dispositivos como fitas e videocassetes fornecem um modelo equivalente ao JMF para gravação, processamento e apresentação de mídia baseada em tempo. Quando você reproduz um filme, um fluxo de mídia é fornecido para o videocassete através de uma fita de vídeo. O videocassete ler e interpreta os dados da fita e envia sinais adequados de áudio e vídeo para a televisão, como pode ser observado na Figura 10:

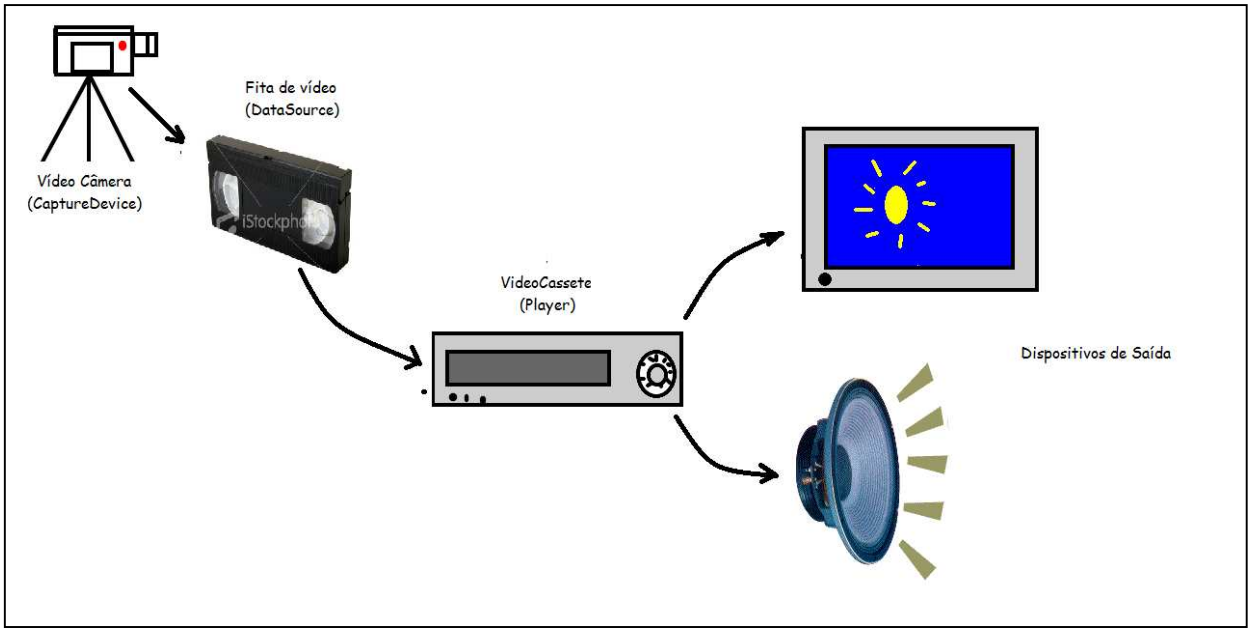


Figura 10: Gravação, processamento e apresentação de uma mídia baseada em tempo

Na API JMF, um *DataSource* encapsula um fluxo de mídia, assim como a fita de vídeo e um *Player* fornece mecanismos de apresentação e controle semelhantes a um Videocassete. Para reprodução e captura de áudio e vídeo com o JMF é necessário o uso de dispositivos apropriados para tais funções, como por exemplo, uma webcam (captura) e auto-falantes (reprodução).

Dentre outras características, JMF pode:

- Executar vários arquivos multimídia em um *applet* ou aplicativo Java. Os formatos suportados incluem AU, AVI, MIDI, MPEG, QuickTime, e WAV.
- Executar fluxo de mídia da Internet.
- Captação de áudio e vídeo com o microfone e a câmera de vídeo, em seguida, armazenar os dados em um formato compatível.
- Processo de mídia baseada no tempo e alterar o formato do tipo de conteúdo.
- Transmissão de áudio e vídeo em tempo real na Internet.
- Transmissão ao vivo de rádio ou programas de televisão.

3.1 Arquitetura da API JMF

A seguir serão comentadas as principais classes e interfaces contidas na arquitetura JMF:

Manager: Ainda comparando com o sistema de videocassete, a funcionalidade do *Manager* pode ser vista como os dispositivos eletrônicos que realiza o intermédio e integração entre a fonte de dados (fita de vídeo) e o player (videocassete). De modo geral, um Manager pode ser visto como um objeto versátil que realiza o intermédio entre duas classes diferentes. JMF oferece quatro tipos diferentes de *Managers:*

- *Manager*: Utilizado para criar *Players, Processors, DataSources e DataSinks.* A classe *Manager* fornece métodos *static* que permitem acessar a maioria dos recursos do JMF;

- *PackageManager*: Mantém um registro de pacotes que contém as classes JMF;
- *CaptureDeviceManager*: Mantém um registro dos dispositivos de captura que podem ser acessados por um na criação de um *Player*, como por exemplo, uma webcam;
- *PluginManager*: Mantém um registro dos componentes *plug-in* JMF para processamento.

DataSource: A classe *DataSource* encapsula a localização da mídia e o protocolo de software usado para fornecer os meios de comunicação. Uma vez obtida, uma fonte de mídia não pode ser reutilizada para oferecer outras mídias, seria como utilizar um mesmo espaço em uma fita cassete para reproduzir dois vídeos diferentes. No JMF, um objeto *DataSource* pode ser um arquivo ou um fluxo de entrada da Internet. Uma vez criado, um *DataSource* pode ser inserido em um *Player*, para ser processado, sendo que o *Player* não se preocupa com a origem do *DataSource* ou qual era a sua forma original. Um *DataSource* pode ser identificado por um objeto da classe *MediaLocator*, ou uma URL *(Identificador Universal de recursos),* um *MediaLocator* é semelhante a um URL e pode ser construído a partir de um URL.

CaptureDevice: representa um hardware que é usado na captura de dados multimídia, como por exemplo, microfones e câmeras de vídeo. Os dados resgatados destes dispositivos podem ser diretamente enviados para um *Player*, processados, ou mesmo convertidos em outro formato mais adequado para armazenamento.

Player: tem como entrada um fluxo de áudio e/ou vídeo e torna a saída em um alto-falante ou uma tela, bem como um CD player que lê um CD de música e envia a saída para os alto-falantes. Um *Player* pode ter estados que existem naturalmente porque o *Player* tem que preparar a sua fonte de dados antes de começar a reproduzir a mídia. Isso ocorre também em aparelhos de DVD, por exemplo, quando você insere uma mídia de DVD onde um estado de leitura é apresentado no *display* de modo que o aparelho possa identificar a fonte de dados, bem como o formato ou mesmo se é compatível com os tipos de mídia que o mesmo pode executar. Da mesma forma um *Player* JMF deve fazer algumas preparações antes que uma fonte de dados multimídia possa ser ouvida (áudio) ou assistida (vídeo). JMF define seis estados para um *Player*, são eles:

- *Unrealized*: Neste estado, o objeto *Player* foi instanciado. Como um bebê recém-nascido que ainda não reconhece o seu ambiente, um *Player* recém instanciado ainda não sabe nada sobre sua mídia.
- *Realizing*: O objeto *Player* muda do estado *Unrealized* para *Realizing* quando o método realize() for chamado. No estado *Realizing* o *Player* está em processo de determinação de suas necessidades em recursos. Durante o *Realizing* o *Player* adquire recursos que ele só precisa adquirir uma vez, bem como os recursos de processamento, além de recursos de uso exclusivo como acontece, por exemplo, em webcams que só podem ser utilizadas por um *Player* a cada momento.
- *Realized*: Quando um Player termina o estado *Realizing*, ele se move para o estado *Realized.* Neste estado o *Player* sabe dos recursos que precisa e tem informações do tipo de mídia que será processada.
- *Prefetching*: Quando o método *prefetch* é chamado, um *Player* muda de estado *realized* para o estado de *prefetching*. Um *Player* no estado *prefetching* está se preparando para apresentar a sua mídia. Durante esta fase, o *Player* pré-executa os dados de mídia, obtém recursos de uso exclusivo, e qualquer outra coisa necessária para rodar os dados da mídia.
- *Prefeched*: O *Player* está pronto para iniciar a execução dos dados da mídia. Nesse estado o Player aguarda que o método *start* seja chamado para entrar no estado de *started.*
- *Started*: O player inicia a apresentação dos dados de mídia.

A Figura11 apresenta um modelo de estados conveniente para a execução de um *Player*:

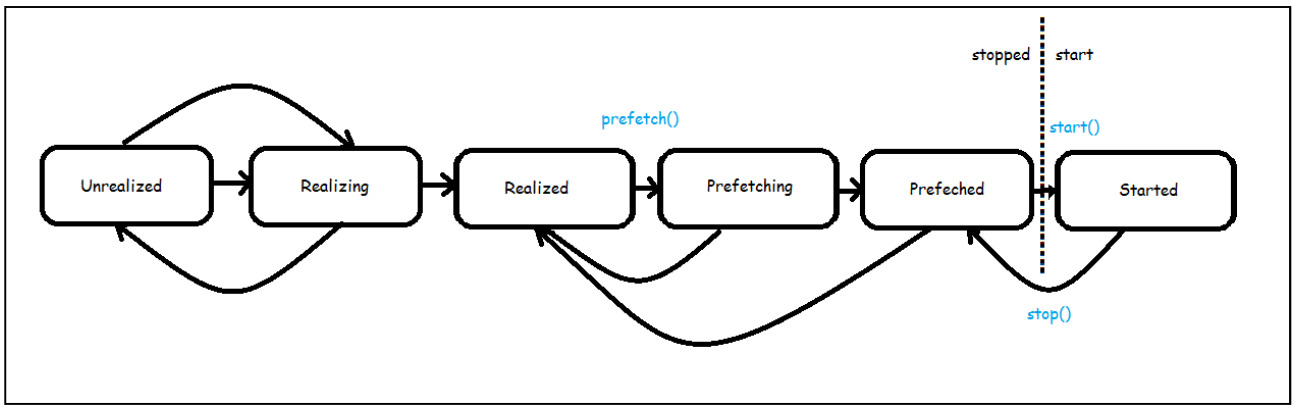


Figura 11: Estados de um Player em execução

Em JMF, o processo de apresentação é modelado pela interface do *Controller*. *Controller* define o mecanismo básico de estado e controle para um objeto que controla, apresenta ou captura mídia baseada em tempo. Ele define as fases que um controlador de mídia atravessa e fornece um mecanismo para controlar as transições entre as fases. Um número de operações que devem ser realizadas antes da apresentação de dados de mídia pode ser demorado, portanto JMF permite controle programático sobre quando eles ocorrem.

Um *Controller* registra uma variedade de específicos do controlador *MediaEvents* para fornecer notificação de alterações no seu estado. Para receber eventos de um *Controller* como um Player, deverá ser implementado a interface *ControllerListener*.

Processor: É um tipo de *Player*. Na API JMF uma interface *Processor* estende *Player*, suportando os controles de apresentação como um Player, porém com a diferença que o *Processor* tem controle sobre o processamento que é realizado sobre o fluxo de mídia de entrada. Além disso, um através de um *Processor*, os dados da mídia de saída podem ser reaproveitados, podendo ser processados em outro *Player* ou *Processor*, ou convertido em algum outro formato.

Além dos seis estados já referidos Player, um *Processor* inclui dois estados adicionais que ocorrem antes que o *Processor* entre no estado *realizing*, mas depois de perceber o estado *unrealized*, que são:

- *Configuring:* Um processador entra no estado de configuração do estado latente, quando o método *configure()* é chamado. Um P*rocessor* entra no estado de *Configuring* quando ele entra no *DataSource* e reconhece o formato da mídia de entrada.
- *Configured:* Depois de ligado ao *DataSource* é configurado os formatos de entrada.

DataSink: É uma interface de base para objetos que ler conteúdos de mídia entregue por um *DataSource* e tornar a mídia para algum destino. Um *DataSink* pode ser comparado com um gravador que armazena os dados de uma mídia para ser gravado em outra mídia ou arquivo.

Format: Um objeto *Format* representa o formato de um objeto de mídia exata. O formato em si não traz parâmetros de codificação específico ou informação de tempo global, que descreve o nome do formato de codificação eo tipo de dados que o formato exige. subclasses formato incluem AudioFormat e *VideoFormat*. Por sua vez, *VideoFormat* contém seis subclasses: (i) H261Format, (ii) H263Format, (iii) IndexedColorFormat, (iv) JPEGFormat, (v) RGBFormat e (vi) YUVFormat.

4 DESENVOLVIMENTO DE APLICATIVOS API JAVA MEDIA FRAMEWORK – JMF

Nesta sessão será apresentado, como objeto de estudo, um aplicativo multimídia que utiliza os principais conceitos de POO e da API JMF vistos até aqui. Neste aplicativo será visto como se dá a implementação de um *Player* de áudio e vídeo utilizando objetos da classe *Manager* para criação de um *Player* a partir de um *DataSource*. O aplicativo será capaz de apresentar, bem como capturar um fluxo de vídeo, seja de um arquivo de mídia local compatível, através do botão "Abrir Arquivo", ou especificando um *URL* de mídia que pode também acessar um dispositivo de captura (webcam ou microfone) clicando no botão "Inserir Local".

As seguintes etapas serão seguidas para reproduzir um clipe de mídia:

1- Especificar a fonte de mídia;
2- Criar um Player para a mídia;
3- Obter a mídia de saída e os controles do *Player;*
4- Exibir a mídia e os controles.

Percorreremos o código fonte da classe nomeada *AppMinicurso* de modo a rever o que está acontecendo em cada linha, bem como as etapas descritas acima, com o objetivo de implementar um aplicativo semelhante ao da Figura 12:

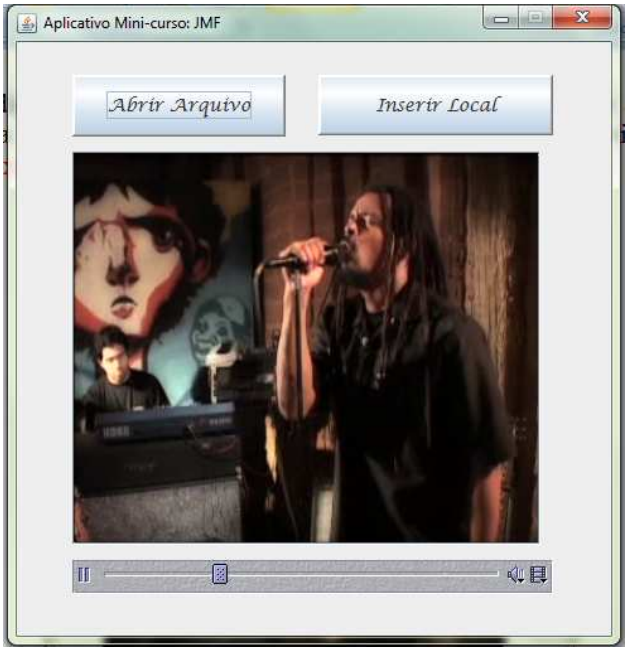


Figura 12: Aplicativo que usa a API JMF

4.1 ETAPA 1: ESPECIFICAR A FONTE DE MÍDIA

O clipe de mídia precisa ser processado antes de ser reproduzido. Para processar um clipe de mídia, o programa precisa acessar uma fonte de mídia. Em nível de linguagem de programação, a fonte de mídia que é inserida em um *Player* pode ser tanto um objeto que implementa a interface *DataSource*, quanto um *MediaLocator*. Para o nosso objeto de estudo, a fonte de mídia pode ser obtida através dos métodos *openFile* e *openFile2* que são chamados nas escutas dos botões e inseridas em um *MediaLocator*, como pode ser observado na Figura 13, que mostra uma parte do código onde a fonte de mídia é especificada.

```
private void openFile() {
JFileChooser fileChooser = new JFileChooser();
File arquivo;

if(player != null){
    stop();
}
fileChooser.setFileSelectionMode(
     JFileChooser.FILES_ONLY );
int result = fileChooser.showOpenDialog( this );

// para o caso de o usuário clicar no botão cancelar
if ( result == JFileChooser.CANCEL_OPTION )
    arquivo = null;
else
    arquivo = fileChooser.getSelectedFile();
try {
    media = new MediaLocator(arquivo.toURL());
    } catch (MalformedURLException ex) {
        mensagensErro("arquivo corrompido");
    }
}

private void openFile2(){

String endereco =
   JOptionPane.showInputDialog(this, "Digite o endereço da mídia");

if(endereco==null || endereco.length()==0){
    return;
    }
media = new MediaLocator(endereco);
}
```

Figura 13: Especificação da fonte de mídia através de um MediaLocator

Conforme mostra a Figura 13, no método *openFile*, a URL do arquivo selecionado através do método *getSelectedFile* de *JFileChooser* (classe do que modela uma janela de seleção de arquivos) é recuperada através do método *toURL* e passado como argumento no construtor do *MediaLocator*. Já o método *openFile2*, uma String digitada pelo usuário na janela criada pelo método *static showInputDialog* da classe *JOptionPane* é repassado diretamente como argumento da classe no construtor do *MediaLocator*.

4.2 ETAPA 2: CRIAR UM PLAYER PARA MÍDIA

Quando o usuário clicar em um dos botões para especificar a fonte de mídia, a respectiva escuta do botão irá chamar o método *openFile* correspondente e logo em seguida o método *criarPlayer*, já que uma fonte de mídia já terá sido especificada. A Figura 14 abaixo mostra a escuta para o botão "Abrir Arquivo":

```
    private void botAbrirVideoActionPerformed(java.awt.event.ActionEvent evt) {

openFile();
criarPlayer();

    }
```

Figura 14: Método associado botão "Abrir Arquivo"

A criação de um objeto *Player* para reproduzir uma mídia a partir de uma fonte de mídia resgatada em *openFile* é feita no método *criarPlayer*. Um objeto *Player* é criado a partir do método *static createPlayer* de *Manager* que tem como parâmetro um *MediaLocator* como pode ser observado na Figura 15:

4.3 ETAPA 3: OBTER MÍDIA DE SAÍDA E CONTROLES DO PLAYER

Como foi comentado na sessão anterior, um *Player* passa por vários estados até poder iniciar a execução a partir de uma fonte de mídia, porém, esses estados devem ser identificados durante a execução do programa, essa identificação é feita através da implementação dos *ControllerListeners* convenientes para escuta dos estados do *Player*.

Os *ControllerListeners* esperam os *ControllerEvents* que os Players geram, para monitorar o progresso de um *Player* no processo de tratamento da mídia. Ainda no médoto *criarPlayer*, o objeto player registra uma instancia da classe interna *ManipulaEventos* para esperar certos eventos que o player gera (ver Figura 6). A classe *ManipulaEventos* estende a classe *ControllerAdapter*, que oferece implementações vazias dos métodos da interface *ControllerListener*, facilitando assim a implementação de *ControllerListener* para as classes que precisam tratar de apenas alguns tipos de *ControllerEvent*.

```
//método criar Player, obtém um fluxo de mídia a partir de um média locator
// e cria um novo player
private void criarPlayer() {

  if(player != null){
        removePreviousPlayer();
  }
  try {
      // cria um novo player
      player = Manager.createPlayer( media );

      //escuta dos eventos que ocorrem em tempo de execução com o Player
      player.addControllerListener( new ManipulaEventos() );

      //inicializa o player
      play();
  }
  catch ( Exception e ){
      e.printStackTrace();
      mensagensErro("Arquivo ou local invalido");
  }
}
```

Figura 15: Método criarPlayer da classe AppMinicurso

4.4 ETAPA 4: EXIBIR A MÍDIA E OS CONTROLES

Para exibir a mídia e os controles do player, é necessário adicionar os componentes, visual e painel de controle, através dos métodos *getVisualComponent* e *getControlPanelComponent* de player em objetos da classe Container. Quando o *Player* completa a carga antecipada, ele passa para o estado *Prefetched* e está pronto para reproduzir a mídia. Durante esta transição, o Player gera um *ControllerEvent* do tipo *PrefetchCompleteEvent*, para indicar que está pronto para exibir a mídia. O Player invoca o método *prefetchComplete* de *ManipulaEventos*, que exibe a GUI do *Player* na frame. Após obter os recursos de hardware, o programa pode obter os componentes de mídia que ele exige. A Figura 16 apresenta a classe interna *ManipulaEventos* que observa os respectivos estados em que o player se encontra e implementa os métodos para apresentação da componente visual e de controle para o usuário.

Quando o clipe de mídia terminar, o *Player* gera um *ControllerEvent* do tipo *EndOfMediaEvent*. A maioria dos reprodutores de mídia "reenrolam" o clipe de mídia depois de chegar ao fim, de modo que os usuários possam ver ou ouvir novamente a partir do início. O método *endOfMedia* trata o *EndOfMediaEvent* e restaura o clipe de mídia para sua posição inicial invocando o método *setMediaTime* de *Player* com um novo Time (pacote javax.media) de 0. O método *setMediaTime* ajusta a posição da mídia para uma posição específica de tempo e é útil para "pular" para uma parte diferente da mídia. Logo após o método *stop* de *Player* é invocado, que termina o processamento da mídia e coloca o Player no estado *Stopped*. É válido ressaltar que é necessário invocar o método start para um *Player Stopped* que não tenha sido fechado retoma a reprodução da mídia

```
private class ManipulaEventos extends ControllerAdapter{
    //método realizeComplete garante o objeto paleyr no estado prefeched
    public void realizeComplete(RealizeCompleteEvent e){
        player.prefetch();
    }

    public void prefetchComplete(PrefetchCompleteEvent e){

        // componetes visual
        visual = player.getVisualComponent();
        if ( visual != null )
            c.add( visual);

        // componente de controle do player
        control = player.getControlPanelComponent();
        if(control != null)
           d.add(control);

        validate();
    }

    public void endOfMedia(EndOfMediaEvent e){
        player.setMediaTime(new Time(0));
        player.stop();
    }
}
```

Figura 16: Classe interna que implementa um Controller para o player

5 CONSIDERAÇÕES FINAIS

O trabalho apresentado neste artigo mostrou a importância e projeção que os Sistemas Multimídia têm alcançado nos Sistemas de Computação. Como trabalhos futuros as pesquisas avançam na especificação de projetos que dêem suporte a novos formatos de áudio e vídeo, sobretudo mais leves para processados e transmitidos. Finalmente, os autores agradecem à Fundação de Amparo à Pesquisa do Estado da Bahia (FAPESB) pela bolsa fornecida ao segundo autor.

REFERÊNCIAS

DEITEL, H.M.,DEITEL, P.J. **Java, como programar.** 4. ed. Porto Alegre: Bookman, 2003.

HORSTMANN. C. S.; CORNELL G., **Core Java 2 – Recursos Avançados**. São Paulo: Makron Books, 2002.

KUROSE, J. F; KEITH W., Ross, **Redes de Computadores E A Internet - Uma Nova Abordagem**, Editora Addison Wesley, 2003.

LU, G., **Communication and Computing for Distributed Multimedia Systems,** Editora Artech House, 1996.

SANTOS, R. **Introdução orientada a objetos usando Java, Rio de Janeiro:** Editora Elsevier, 2003.

SUN MICROSYSTEMS, Inc. **Guia do desenvolvedor Java Media Framework.** 1998, Disponível em: <http://java.sun.com/javase/technologies/desktop/media/jmf/2.1.1/guide/ > Acesso em: 18 julho 2010.

TANENBAUM, Andrew S. **Redes de computadores**. Rio de Janeiro: Campus, 4ª Edição, 2003.

TORRES, Gabriel, **Redes de Computadores: curso completo**. Rio de Janeiro: Axcel Books, 2001.