UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

FERRAMENTA PARA DESENVOLVIMENTO DE PADRÕES DE IMAGEM 2D E SEPARAÇÃO DE CORES

CHARLES THEISS

BLUMENAU
2008

2008/1-08

CHARLES THEISS

FERRAMENTA PARA DESENVOLVIMENTO DE PADRÕES DE IMAGEM 2D E SEPARAÇÃO DE CORES

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. Dalton Solano dos Reis - Orientador

BLUMENAU
2008

2008/1-08

FERRAMENTA PARA DESENVOLVIMENTO DE PADRÕES DE IMAGEM 2D E SEPARAÇÃO DE CORES

Por

CHARLES THEISS

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Dalton Solano dos Reis - Orientador
Membro:	Paulo César Rodacki Gomes
Membro:	José Roque V. da Silva

Blumenau, 08 de julho de 2008

Dedico este trabalho aos amigos de trabalho, e empresa pelo apoio e entendimento pela minha ausência em muitos momentos.

AGRADECIMENTOS

Ao meu filho, por entender minha ausência na sua infância.

À Cia Hering, empresa onde trabalho, pelo apoio financeiro com a bolsa de estudos, e flexibilidade no horário de trabalho.

Ao meu orientador, Dalton Solano dos Reis, pela ajuda e esclarecimento de dúvidas.

RESUMO

Este trabalho apresenta um protótipo de uma ferramenta para desenvolvimento de padrões de imagens e separação de cores. A ferramenta auxilia no desenvolvimento de padrões de imagens e automatiza o processo de separação de cores utilizado na técnica de serigrafia. São apresentados também conceitos sobre manipulação de imagens, tipos de repetições que auxiliam numa melhor construção do padrão, definição de sistema de cores, apresentando alguns destes sistemas e formatos de arquivos, apresentando características de alguns dos formatos mais conhecidos. Também são apresentados vários recursos gráficos da linguagem Java os quais foram utilizados no desenvolvimento deste protótipo.

Palavras-chave: Serigrafia. Processamento de imagens. Padrões de imagens.

ABSTRACT

This work presents an prototype of a tool for pattern images development and color separation. The tool assists in the development of image patterns and automatizes the process of color separation used in the serigraphy technique. Concepts about image manipulation are also presented, repetitions types that assist in a better construction of a pattern, definition of color system, presenting some in these ways, formats of archives, also presenting characteristic of some of the most known formats. Some graphical resources of the Java language are also presented which had been used in the development this prototype.

Key-words: Serigraphy. Image process. Patterns images.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API – *Application Programming Interface*

ARGB – *Alpha, Red, Green, Blue*

AWT – *Abstract Windowing Toolkit*

BMP – *Windows Bit Map*

CMYK – *Cyan, Magenta, Yellow e Black*

GIF – *Graphics Interchange Format*

IBM – *International Business machines*

JPG – *Join Photographic Expert Group*

LZW – *Lempel Ziv Welch*

RF – Requisito Funcional

RGB – *Red, Green, Blue*

RNF – Requisito não Funcional

TIFF – *Tagged Image File Format*

UML – *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

Com o constante desenvolvimento da indústria serigráfica, surge o aumento de competitividade no mercado, ou seja, cada vez mais é necessária maior rapidez no processo de desenvolvimento de artes gráficas e separação de cores. O processo de separação de cores consiste em gerar um fotolito para cada cor da imagem que se pretende imprimir em algum tipo de superfície. Atualmente existem ferramentas que podem ser utilizadas para produção de artes e separação de cores. Algumas das mais conhecidas são: Corel Draw, Adobe Photoshop e Adobe Illustrator. Estas ferramentas podem ser utilizadas para criar as artes gráficas e também no processo de separação de cores, porém são de uso geral, podendo ser usadas em diversas áreas e por este motivo não possuem funcionalidades específicas para área de serigrafia.

Segundo Caza (1967, p. 10), a palavra serigrafia origina-se do latin *sericum* (seda) e do grego *graphe* (escrever), que é um processo de impressão no qual a tinta é vazada através de uma tela preparada. A tela, normalmente de seda, náilon ou poliéster, é esticada em um bastidor de madeira, alumínio ou aço. A gravação da tela se dá pelo processo de fotosensibilidade, onde ela é preparada com uma emulsão fotossensível é colocada sobre um fotolito, sendo este conjunto tela e fotolito colocados por sua vez sobre uma mesa de luz. Os pontos escuros do fotolito correspondem aos locais que ficarão vazados na tela, permitindo a passagem da tinta pela trama do tecido, e os pontos claros, onde a luz passará pelo fotolito atingindo a emulsão, são impermeabilizados pelo endurecimento da emulsão fotossensível que foi exposta a luz.

Esta técnica é utilizada na impressão em variados tipos de materiais, tais como papel, plástico, borracha, madeira, vidro, tecido, superfícies cilíndricas, esféricas, irregulares, de diversas espessuras ou tamanhos, com diversos tipos de tintas ou cores.

Existem no mercado ferramentas específicas para esta área, porém com altíssimo custo. Uma destas ferramentas é o Desing and Repeat Pro, cujo fabricante é uma empresa européia, é Ned Graphics (Ned Graphics, 2004). Outra ferramenta conhecida para este fim é o bestIMAGE, fabricado pela Stork Prints (Stork Prints, 2007). Estas ferramentas, por serem especificamente desenvolvidas para esta área, garantem maior produtividade, reduzindo tempo de desenvolvimento e automatizando algumas etapas do processo.

Diante desta situação, surge a motivação para desenvolver a ferramenta, com algumas das principais funcionalidades necessárias em um software específico aqui apresentado. A

ferramenta foi desenvolvida na linguagem Java. Esta ferramenta tornou-se portável para qualquer sistema operacional que possua a máquina virtual Java instalada.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver uma ferramenta para desenvolvimento de padrões e separação de cores de imagens.

Os objetivos específicos do trabalho são:

a) manipular imagens;
b) auxiliar no desenvolvimento de padrões de imagens, visualizando a imagem em diversas formas de repetições;
c) automatizar o processo de sobreposição de cores;
d) automatizar o processo de separação de cores.

1.2 ESTRUTURA DO TRABALHO

O capítulo 2 apresenta a fundamentação teórica na qual este trabalho é baseado. Na seção 2.1 são apresentados conceitos sobre processamento de imagens. Na seção 2.2 são apresentados alguns tipos de repetições utilizados pela ferramenta Desing and Repeat Pro (Ned Graphics, 2004), que auxiliam no processo de distribuição dos elementos da imagem (motivos). A seção 2.3 comenta o trabalho de Fernandes (2002, p. 34) no que diz respeito ao entendimento dos sistemas de cores existentes. Na seção 2.4 são apresentados alguns dos modos de codificação de imagens existentes. A seção 2.5 apresenta alguns dos formatos de arquivos mais utilizados e algumas de suas características. Na seção 2.6 são apresentados diversos recursos da linguagem Java, utilizados no desenvolvimento do protótipo desta ferramenta. A seção 2.7 apresenta duas ferramentas correlacionadas ao protótipo da ferramenta desenvolvida, o Desing and Repeat Pro (Ned Graphics, 2004) e o BestIMAGE (Stork Prints, 2007).

O capítulo 3 aborda os processos envolvidos no desenvolvimento do protótipo da ferramenta, tais como requisitos da ferramenta, especificação de diagramas de casos de uso,

diagramas de classes e dois diagramas de seqüência ilustrando duas das possíveis seqüências de operações no protótipo. Tem-se também uma explanação sobre o desenvolvimento das funcionalidades implementadas com figuras e comentários sobre os códigos escritos.

O capítulo 4 apresenta as conclusões sobre as funcionalidades implementas no protótipo, bem como as que não foram implementadas, mas que estão envolvidas no projeto. O capítulo 4 apresenta também sugestões para possíveis extensões deste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Nas seções seguintes são detalhados conceitos sobre processamento de imagens, tipos de repetições, sistema de cores, modos de codificação de imagens, formatos de arquivos e alguns recursos gráficos da linguagem Java. Na última seção são descritos trabalhos correlatos.

2.1 PROCESSAMENTO DE IMAGENS

Segundo Gonzalez e Woods (2000, p. 4), uma imagem digital 2D (duas dimensões) é uma imagem f(x,y) discretizada tanto em coordenadas espaciais como em brilho. Uma imagem digital pode ser considerada como sendo uma matriz cujos índices de linha e colunas identificam um ponto na imagem e o correspondente valor do elemento da matriz identifica o atributo da cor naquele ponto. Os elementos da matriz são chamados de pixel (*picture elements)*. Conforme Facon (1993, p. 28), imagens digitais são úteis quando estão armazenadas em uma forma que possa ser utilizada por outras aplicações. Por este motivo, é necessário armazenar estas imagens de forma padronizada para que elas possam ser manipuladas.

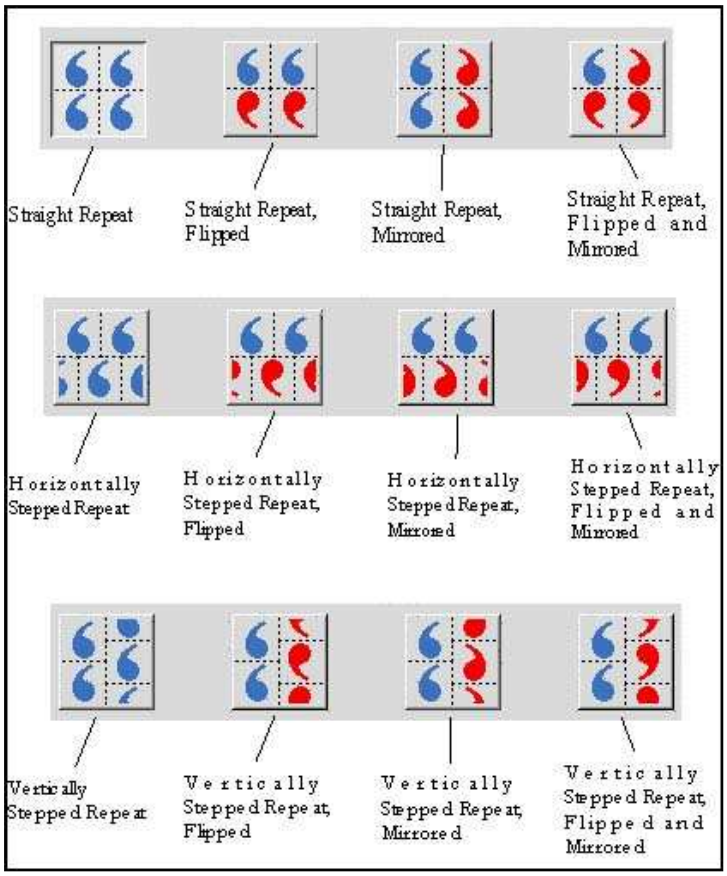
2.2 TIPOS DE REPETIÇÕES

Segundo Ned Graphics (2004), para ter uma boa distribuição dos motivos[1] em um padrão pode-se usar diversos tipos de repetições, como mostra a figura 1. Estes diversos tipos de repetições auxiliam na distribuição dos motivos, facilitando com que sejam distribuídos de forma homogênea.

A figura 2 mostra a transformação de uma imagem com repetição do tipo *horizontally stepped repeat* para *straight repeaet.* É possível notar que a imagem duplicou seu tamanho na

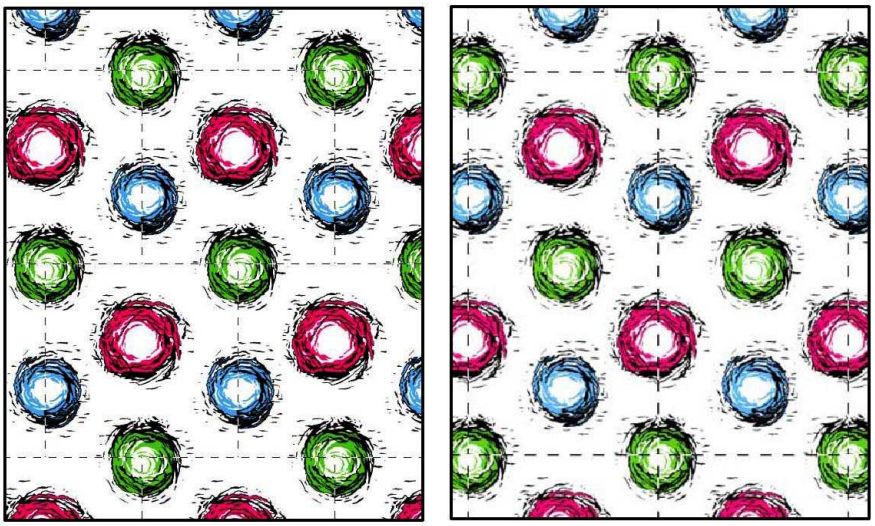
[1] Motivos são partes da imagem que compõem o padrão.

altura. Esta transformação é necessária pois não é possível gravar as matrizes de impressão (cilindros usados para impressão na técnica de serigrafia) com o padrão repetindo com deslocamentos ou saltos, ou seja, é necessário que o padrão esteja repetindo lado a lado, ou com o tipo de repetição *straight repeat*, como mostra a figura 1.



Fonte: Ned Graphics (2004).

Figura 1 – Tipos de repetições



Fonte: Ned Graphics (2004).

Figura 2 – Transformação de uma repetição do tipo *horizontally stepped repeat* para *straight repeat*

2.3 SISTEMAS DE CORES

Segundo Fernandes (2002, p. 34) um sistema de cores é um método que explica propriedades ou o comportamento das cores num contexto. Não existe um único sistema que seja capaz de explicar todas as características relacionadas à cor. Por isso existem diferentes modelos para descrever os aspectos percebidos pelo olho do ser humano. Existem sistemas baseados em adição de luzes de cores, como o sistema *Red, Green, Blue* (RGB) usado nos monitores e televisores, e sistemas baseados em subtração de luzes de cores, sendo um desses é o *Cyan, Magenta, Yellow, Black* (CMYK) que é a base do processo de impressão em quatro cores.

2.4 MODOS DE CODIFICAÇÃO DE IMAGENS

Segundo Marino (1998) existem várias formas de descreverem como os pixeis são organizados e representados na memória do computador. Na seqüência serão apresentados alguns os modos RGB, CMYK, preto e branco e indexado.

2.4.1 RGB

O RGB, que consiste em um sistema que usa três cores por pixel, permitindo reproduzir até 16,7 milhões de cores. Cada cor é representada por um *byte,* possibilitando assim 256 combinações possíveis para cada cor. O valor `0, 0, 0` de RGB equivale a cor preta e o valor `255, 255, 255` equivale a cor branca. Estas três cores são conhecidas como cores primárias e o sistema é baseado na combinação da luz emitida por estas três fontes de luz. Este processo de combinação das fontes de luz é chamado de aditiva (Marino, 1998).

2.4.2 CMYK

O CMYK usa quatro cores por pixel em seu sistema, onde cada cor é representada por um *byte*. O sistema CMYK baseia-se na qualidade de luz absorvida por uma tinta impressa sobre algum tipo de superfície. Desta forma quanto maior a quantidade de tinta depositada, mais saturada ficará a impressão. Os valores de CMYK são representados pela porcentagem da cor que será impressa. Valores de 0% de *cyan*, 0% de *magenta*, 0% de *yellow* e 0% de *black* correspondem a cor branca e no caso de 100% nas quatro cores correspondem a cor preta. Como o modo CMYK usa quatro *bytes* por pixel, ele ocupa 33% mais espaço em memória do que o modo RGB (Marino, 1998).

2.4.3 Modo preto e branco

Este modo também é conhecido como *bitmap* pelo motivo de representar a cor de um pixel com apenas um *bit*. Neste sistema cada pixel pode assumir o valor zero, que representa a cor preta, ou um, que representa a cor branca. Dentre todos os modos de armazenamento de imagens este é o que resulta em um arquivo de menor tamanho para armazenamento em disco (Marino, 1998).

2.4.4 Modo indexado

Neste modo cada pixel assume um valor presente numa paleta de 256 cores. Cada pixel desta tabela (paleta) é um índice de uma outra tabela que representa o valor RGB da cor da imagem. Este modo usa um *byte* para representar cada pixel, ou seja, o índice da tabela RGB. Segundo Corrigan (1994, p. 73), alguns usuários não necessitam milhares de cores para representarem suas imagens, o que acaba tornando-se um problema. Para estes usuários 256 cores ou até mesmo 16 cores é uma quantidade suficiente, tornando o modo indexado muito útil para estes casos (Marino, 1998).

2.5 FORMATOS DE ARQUIVOS

Segundo Corrigan (1994, p. 151), existe um número grande de formatos de arquivos para armazenamento de imagens. Para cada tipo de aplicação, um formato pode se adaptar-se melhor que outro. Para tanto, informações sobre os formatos GIF, TIFF e BMP são dadas as seguir.

2.5.1 Formato GIF

O *Graphics Interchange Format* (GIF), sempre comprime e codifica as imagens pela especificação *Lempel-Ziv-Welch* (LZW). A sua característica mais importante é suportar apenas 8 *bits* por pixel no máximo. Apesar desta limitação, o GIF ainda é o formato mais popular para armazenar imagens indexadas. Outro aspecto importante é que o formato GIF usa o método de compressão LZW sem perda de informação, ou seja, uma imagem GIF pode ser lida e gravada infinitas vezes e sempre será idêntica à original. Esta é uma vantagem do LZW sobre o *Joint Photographic Experts Group* (JPEG), que sempre acarreta em perda de informação (Corrigan, 1994).

Segundo Free Software Foundation (2007), a patente sobre o LZW da empresa Unisys caducou em junho de 2003, mas havia ainda uma patente aplicável da *International Business Machines* (IBM), que expirou em primeiro de outubro de dois mil e seis (01/10/2006), tornando assim o formato GIF livre.

2.5.2 Formato TIFF

O *Tagged Image File Format* (TIFF) é o formato mais aceito por programas de tratamento e manipulação de imagens. O TIFF é capaz de armazenar imagens *true color* de 24 ou 32 bits e é um formato bastante utilizado para transporte de imagens do *desktop* para saídas de *scanners* e separação de cores. O TIFF permite que imagens sejam comprimidas usando o método LZW e permite salvar campos informativos (*caption)* dentro do arquivo (Corrigan,

1994).

2.5.3 Formato BMP

O *Windows BitMaP* (BMP) é o formato gráfico nativo do Windows da Microsoft. É capaz de armazenar cores em até 24 bits. Devido à popularidade do Windows muitos programas, inclusive em plataforma Macintosh, suportam o formato BMP (Corrigan, 1994).

2.6 RECURSOS GRÁFICOS DA LINGUAGEM JAVA

Segundo a Sun Microsystems (2008), a API 2D do Java fornece recursos para manipulação de gráficos, textos e imagens para programas Java através da *Abstract Windowing Toolkit* (AWT). A seguir são apresentados alguns recursos da API Java 2D, que serão utilizados no desenvolvimento da ferramenta.

2.6.1 Sistema de coordenadas da API 2D Java

Conforme a Sun Microsystems (2008), a API 2D do Java mantém dois sistemas de coordenadas de espaço, sendo o espaço do usuário, onde os gráficos primitivos são especificados e o espaço de dispositivos que é o sistema de coordenada de um dispositivo de saída, tais como, vídeo, janela ou impressora.

O espaço do usuário é um sistema de coordenadas lógico, independente de dispositivo. É o espaço de coordenada que o programa usa. Toda geometria passada para as rotinas de *render* do Java 2D são especificadas no espaço de coordenadas do usuário.

Conforme Sun Microsystems (2003), quando é feita uma transformação padrão do espaço do usuário para o espaço de um dispositivo, a origem do espaço do usuário é o canto superior esquerdo da área de desenho do componente. A coordenada x aumenta para a direita e a y para baixo. O canto superior esquerdo é a coordenada (0,0). Todas as coordenadas são especificadas usando inteiros, que normalmente são suficientes. Entretanto em alguns casos

pode ser necessário usar `float` ou `double`, que também são suportados.

O espaço de coordenadas dos dispositivos é dispositivo-dependente, variando de acordo com o *render* do dispositivo. Embora o sistema de coordenadas de uma janela ou monitor seja muito diferente do sistema de coordenadas de uma impressora, essas diferenças são invisíveis em um programa Java. As conversões necessárias entre o espaço do usuário e o espaço do dispositivo são executadas automaticamente durante o processo de *rendering*.

2.6.2 Formas geométricas primitivas da API 2D Java

Segundo a Sun Microsystems (2008), a API do Java 2D possui um pacote de classes que fornecem algumas formas geométricas primitivas. O pacote `java.awt.geom` fornece classes para desenhar formas básicas tais como pontos, linhas, retângulos, arcos, elipses e curvas.

A classe `Point2D` do pacote `java.awt.geom` define a representação de um ponto nas coordenadas de espaço (`x,y`). Um `Point2D` não é o mesmo que pixel. Um `Point2D` não possui área e não possui cor. `Point2D` serve para criar outras formas e possui métodos para calcular a distância entre dois pontos. O código no quadro 1 mostra duas maneiras de se criar um `Point2D`.

```
Point2D.Double point = new Point2D.Double(x, y);
Point2D.float point = new Point2D.Float(x, y);
```

Fonte: Sun Microsystems (2008).

Quadro 1 – Construtores de `Point2D`

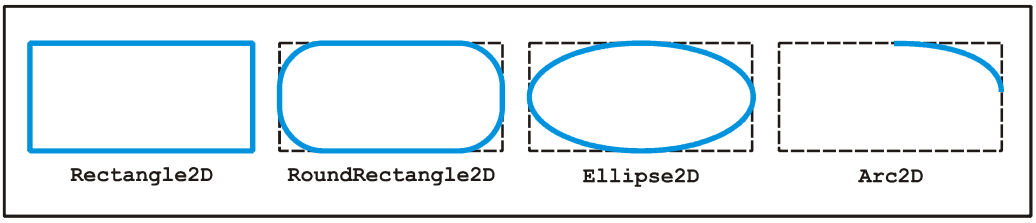
A classe `Line2D` do pacote `java.awt.geom` é uma classe abstrata que representa uma linha. Uma linha na classe `Line2D` é definida por dois pontos, o ponto inicial e o ponto final. O código no quadro 2 mostras três formas de serem criadas linhas usando a classe `Line2D`.

```
Line2D.Double(x1, y1, x2, y2);
Line2D.Float(float X1, float Y1, float X2, float Y2);
Line2D.Float(Point2D p1, Point2D p2);
```

Fonte: Sun Microsystems (2008).

Quadro 2 – Construtores de `Line2D`

As classes `Rectangle2D`, `RoundRectangle2D`, `Arc2D` e `Ellipse2D` são derivadas da classe `RectangularShape`. Esta classe define métodos para criar formas retangulares fechadas ou apenas contornos. A figura 3 mostra algumas das possíveis formas geométricas de serem criadas com a classe `RectangularShape` e suas descendentes.



Fonte: Sun Microsystems (2008).

Figura 3 – Formas geométricas primitivas da classe `RectangularShape`

2.6.3 Processo de *rendering* da API 2D Java

Conforme a Sun Microsystems (2008), quando um componente precisa ser mostrado o método `paint` ou `update` automaticamente são invocados com o contexto gráfico apropriado.

A API Java 2D possui a classe `java.awt.Graphics2D` que é estendida da classe `Graphics`, a qual fornece acesso a gráficos sofisticados e características de *rendering* do Java 2D. Estas características podem ser descritas como:

a) *rendering* do contorno de algum tipo geométrico primitivo, usando atributos de pintura ou espessura de contornos (método `draw`);

b) *rendering* do preenchimento de algum tipo geométrico primitivo, podendo ser preenchido com uma textura ou uma cor especificada nos atributos de pintura (método `fill`);

c) *rendering* de textos (método `drawString);`

d) *rendering* de imagens (método `drawImage`).

Segundo a Sun Microsystems (2008), os métodos de *rendering* são divididos em dois grupos, métodos para desenhar formas e métodos para modificar o *rendering*. Nos métodos que modificam o *rendering* pode-se modificar espessura de contornos, escalar e rotacionar objetos durante o processo de *rendering*.

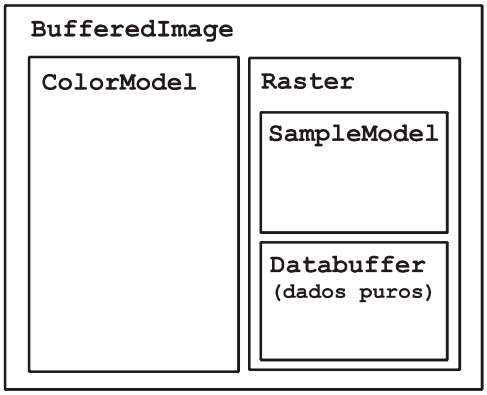
Conforme a Sun Microsystems (2008), para utilizar os recursos de *rendering* da API 2D do Java é necessário converter um objeto `Graphics` passado como parâmetro para o método de *rendering* do componente, para um objeto `Graphics2D`, conforme mostra o código no quadro 3.

```
public void paint (Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    ...
}
```

Fonte: Sun Microsystems (2008).
Quadro 3 – Conversão de tipo do objeto `Graphics` para `Graphics2D`

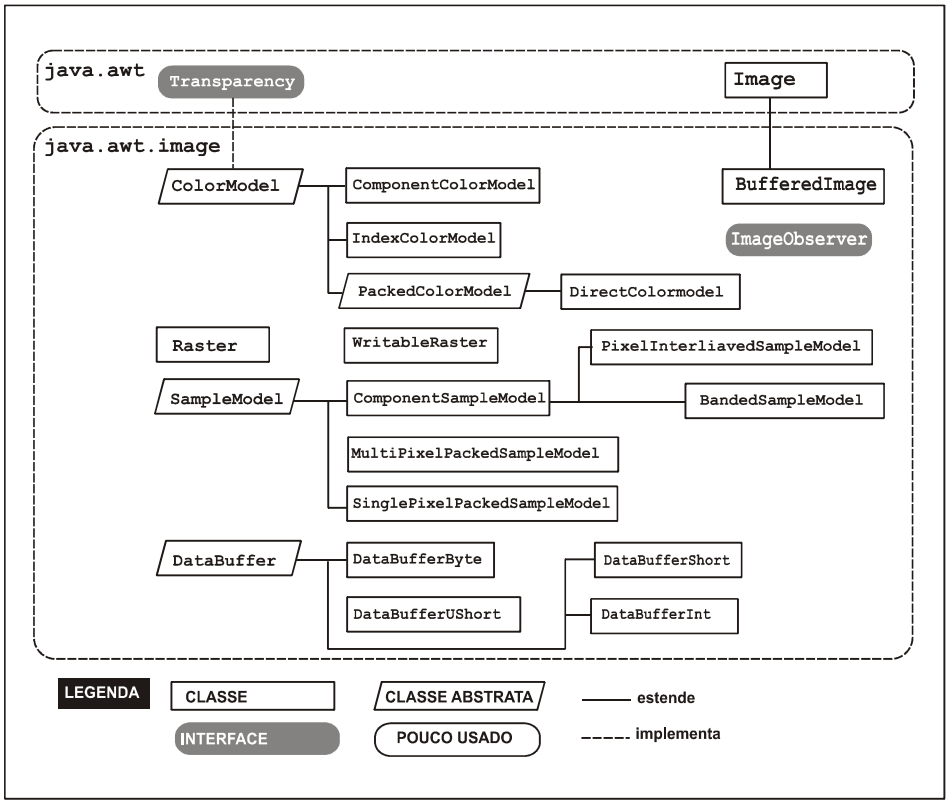
2.6.4 Estrutura para armazenamento de imagens

Conforme Niemeyer e Knudsen (2000, p. 543), para criar uma imagem em Java é preciso entender como um `BufferdImage` é formado, que é uma subclasse da classe `image`. A classe `BuffedImage` foi projetada para dar suporte a imagens com praticamente qualquer forma de armazenamento. A figura 4 mostra os elementos de um `BufferdImage`.



Fonte: Niemeyer e Knudsen (2000, p. 543).
Figura 4 – Elementos de um `BufferedImage`

Como mostra a figura 4, um `BufferdImage` consiste em duas partes, um `Raster` e um `ColorModel`. O `Raster` contém os dados reais da imagem e `ColorModel` interpreta os dados da imagem como cores, traduzindo os valores de dados do `Raster` em objetos `Color`. O sistema gráfico captura os dados para cada pixel da imagem a partir do `Raster`, depois o `ColorModel` informa que cor cada pixel deve ter. Conforme a figura 4, o `Raster` também é composto por duas partes, um `DataBuffer` e um `SampleModel`. O `DataBuffer` é um adaptador para *arrays* de dados, que são *arrays* de `byte`, `short` ou `int`. O `SampleModel` extrai os valores de dados de um determinado pixel a partir do `DataBuffer`. Como mostra a figura 5, a API 2D do Java possui vários tipos de `ColorModels`, `SampleModels` e `DataBuffers`, que servem como bloco de montagem aos formatos mais comuns de armazenamento de imagens.



Fonte: Niemeyer e Kanudsen (2000, p. 534).

Figura 5 – Pacote `java.awt.image`

Como mostra a figura 5, existem alguns modelos de cores já definidos no pacote `java.awt.image`. Por exemplo, o construtor de `IndexColorModel` possui três parâmetros, os quais são: número de *bits* por pixel, número de cores da palheta e três *`arrays`* de *`bytes`*, que servem para representar as cores RGB da paleta da imagem.

2.6.5 Recursos para construção de uma imagem em Java

Conforme descrito na seção anterior, uma imagem pode ser armazenada numa estrutura do tipo `BufferedImage`, que é composta por um `ColorModel` e um `Raster`. Porém nem sempre é necessária uma instância explícita de cada uma destas classes para criação de um `BufferedImage`.

Segundo a Sun Microsystems (2006), existem três construtores para um `BufferedImage`, conforme mostra o quadro 4.

```
public BufferedImage(int width, int height, int imageType);

public BufferedImage(int width, int height, IndexedColorModel cm);

public BufferedImage(ColorModel cm, WritableRaster raster, boolean
isRasterPremultiplied, Hashtable<?,?> properties);
```

Fonte: Sun Microsystems (2006).

Quadro 4 – Construtores de `BufferedImage`

O primeiro construtor mostrado no quadro 4 constrói um `BufferedImage` com um tipo de imagem predefinido. Os parâmetros `width`, `height` e `imageType` indicam respectivamente a largura, altura e o tipo da imagem.

Conforme Sun Microsystem (2006), existem treze tipos de imagens possíveis para se criar um `BufferedImage`. Os tipos são apresentados no quadro 5.

```
TYPE_INT_RGB
TYPE_INT_ARGB
TYPE_INT_ARGB_PRE
TYPE_INT_BGR
TYPE_3BYTE_BGR
TYPE_4BYTE_ABGR
TYPE_4BYTE_ABGR_PRE
TYPE_BYTE_GRAY
TYPE_USHORT_GRAY
TYPE_BYTE_BINARY
TYPE_BYTE_INDEXED
TYPE_USHORT_565_RGB
TYPE_USHORT_555_RGB
```

Fonte: Sun Microsystems (2006).

Quadro 5 – Tipos de imagem

O segundo construtor apresentado no quadro 4 constrói uma imagem com o tipo `TYPE_BYTE_BINARY` ou `TYPE_BYTE_INDEXED` que são tipos de imagens indexados.

Caso o argumento `imageType` for a constante `TYPE_BYTE_BINARY`, o número de entradas no `ColorModel` é usado para determinar se a imagem deve conter um, dois ou quatro bits por pixel. Se o `ColorModel` tem uma ou duas entradas, a imagem terá um bit por pixel. Se tiver três ou quatro entradas, a imagem terá dois bits por pixel. Se tiver entre cinco e dezesseis entradas, a imagem terá quatro bits por pixel. Caso contrário é lançada uma exceção `IllegalArgumentException`. Os parâmetros `width`, `height` e `cm` indicam respectivamente a largura, altura e o `ColorModel` que será usado.

Caso for usada a constante `TYPE_BYTE_INDEXED` no argumento `imageType`, é criada uma imagem com 256 cores. Quando uma informação de cor é armazenada numa imagem deste tipo, a cor mais próxima no mapa de cores é determinada pelo `IndexColorModel` e um índice resultante é armazenado.

O terceiro construtor apresentado no quadro 4 constrói um `BufferedImage` com o `ColorModel` e o `Raster` especificado.

2.6.6 Recursos para acesso e manipulação de pixeis

Conforme a *Aplication Program Interface* (API) do Java (SUN MICROSYSTEM, 2006), uma coleção de pixeis é representada por um `Raster`, que consiste em `DataBuffer` e um `ColorModel`. O `SampleModel` permite acessar amostras do `DataBuffer` e provê informação de baixo nível que pode ser usada pelo programador para acessar e manipular pixeis diretamente. Alguns dos métodos fornecidos pelo `SampleModel` para acesso e manipulação de pixeis são mostrados no quadro 6.

```
public Object getDataElements(int x, int y, int w, int h, Object obj,
DataBuffer data);

public int[] getPixel(int x, int y, int[] iArray, DataBuffer data);

public void setDataElements(int x, int y, Object obj, DataBuffer data);

public void setPixel(int x, int y, int[] iArray, DataBufer data);
```

Fonte: Sun Microsystems (2006).

Quadro 6 – Métodos para acesso e manipulação de pixeis

O método `getDataElements` retorna amostras dos pixeis da área retangular especificada nos parâmetros `x, y, w, h, obj e data`, que significam respectivamente a coordenada `x` inicial, coordenada `y` inicial, quantidade de pixeis na largura, quantidade de pixeis na altura, vetor para guardar as amostras dos pixeis e o `DataBuffer` da imagem no qual serão extraídas as amostras dos pixeis.

O método `getPixel` retorna uma amostra de um pixel isoladamente. Os parâmetros `x, y, iArray e data` determinam as coordenadas do pixel e significam respectivamente a coordenada x, coordenada y, vetor para guardar amostra e o `DataBuffer` da imagem no qual será extraída a amostra do pixel.

O método `setDataElements` configura uma determinada área retangular da imagem com as amostras recebidas no vetor `obj` como parâmetro. Os parâmetros `x, y, obj e data` significam respectivamente a coordenada inicial `x`, coordenada inicial `y`, as amostras dos pixeis e o `DataBuffer` da imagem que receberá a configuração das amostras.

O método `setPixel` configura um determinado pixel isoladamente. Os parâmetros `x, y, iArray e data` determinam as coordenadas do pixel e significam respectivamente a

coordenada `x`, coordenada `y`, vetor que contem a amostra e o `DataBuffer` da imagem no qual receberá configuração da amostra do pixel.

Conforme a Sun Microsystems (2006), caso seja especificada uma coordenada não existente na imagem, em qualquer um dos métodos do quadro 6, será lançada uma exceção do tipo `ArrayIndexOutOfBoundsException`. Caso algum dos métodos apresentados no quadro 6 contenham um `DataBuffer` com o valor `null`, será lançada uma exceção do tipo `NullPointerException`.

2.6.7 Recursos para leitura e gravação de imagens

Conforme Sun Microsystems (2001), a linguagem Java possui uma API específica para leitura e gravação de imagens, a Java I/O API. Esta API prove uma arquitetura para trabalhar com imagens armazenadas em arquivos e ou acessar imagens através de uma rede. O código no quadro 7 mostra como ler uma imagem no formato GIF.

```
File f = new File(“c:\imagens\minhaImagem.gif”);
BufferedImage bi = ImageIO.read(f);
```

Fonte: Sun Microsystems (2001).

Quadro 7 – Leitura de arquivos

Segundo a Sun Microsystems (2001), o formato da imagem é auto detectado pela API, baseando-se no conteúdo do arquivo. Muitos arquivos de imagens contém um número de identificação no cabeçalho que identifica o seu formato. Em formatos que não contém este número de identificação de formato são necessários códigos mais sofisticados para fazer a detecção. Chamando o método `ImageIO.getReaderFormatNames` é possível obter-se uma lista com o nome dos formatos suportados pela API do Java. O código no quadro 8 mostra como gravar uma imagem no formato GIF.

```
BufferedImage bi;
File f = new File(“c:\imagens\minhaImagem.gif”);
ImageIO.write(bi,”gif”,f);
```

Fonte: Sun Microsystems (2001).

Quadro 8 – Gravação de arquivos

A lista de formatos suportados pode ser obtida chamando o método `ImageIO.getWriterFormatNames` (Sun Microsystems, 2001).

2.7 TRABALHOS CORRELATOS

Nesta seção serão apresentados dois softwares existentes no mercado. Esses possuem funcionalidades que aumentam produtividade e automatizam algumas etapas do processo de serigrafia.

2.7.1 Design and Repeat Pro

O Design and Repeat Pro (Ned Grafics, 2004) possui funcionalidades específicas para o desenvolvimento de padrões têxteis.

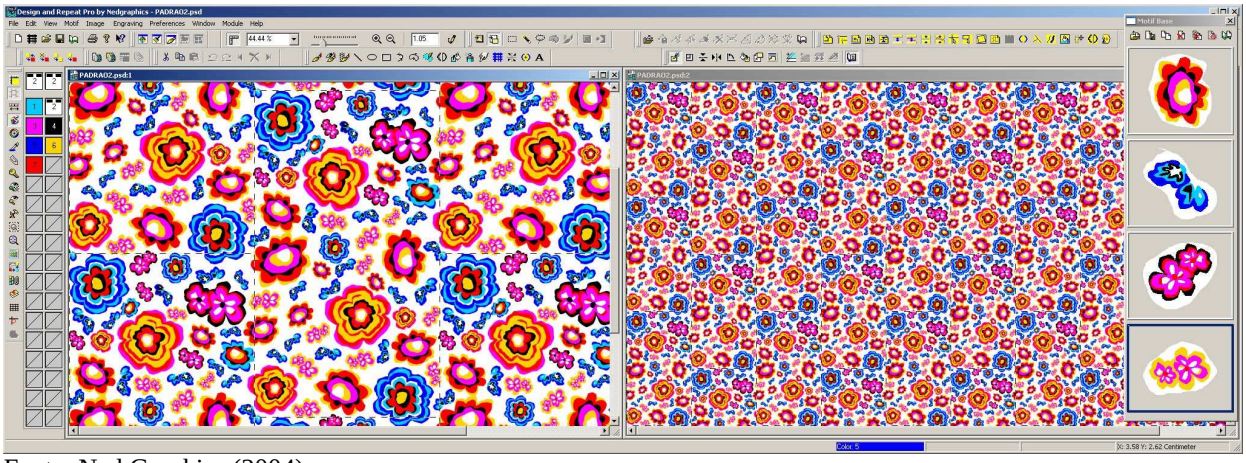
Uma funcionalidade que proporciona agilidade no desenvolvimento de padrões é o processo de distribuição dos motivos. São diversos tipos de repetições que o software oferece, onde os motivos podem ser inseridos aleatoriamente, sendo visualizados no padrão com mais repetições na medida em que são inseridos (figura 6).

O software permite que sejam recortadas partes do padrão e guardados numa base de motivos para uso posterior. Na figura 6 é possível visualizar a base de motivos no canto direito da tela do software.

Existe também uma barra de ferramentas de edição, composta por diversas ferramentas, tais como lápis, aerógrafo, elipse e retângulo. Da mesma forma existe uma barra de ferramentas de seleção, com as seguintes ferramentas: laço, retângulo, elipse e *path*. Estas ferramentas de seleções servem para recortar partes da imagem e adicioná-las na base de motivos.

Outra funcionalidade é a automação do processo de separação de cores, onde o software gera automaticamente uma imagem preta e branca para cada cor existente na imagem. A separação de cores pode ser com sobreposição e é configurável pelo usuário.

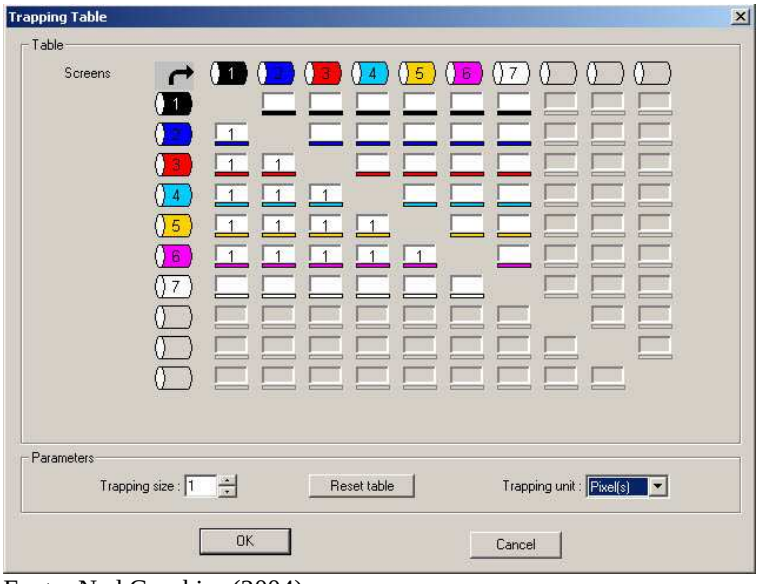
O processo de sobreposição de cores é configurado em uma tabela, onde são apresentadas todas as cores da paleta da imagem nas colunas e linhas, sendo assim possível especificar um valor para cada intersecção de cores. Este valor especifica quantos pixeis serão sobrepostos quando a cor da linha encontrar a cor da coluna. A figura 7 mostra a tabela de sobreposição de cores do software.



Fonte: Ned Graphics (2004).

Figura 6 – Interface do Desing and Repeat Pro da Ned Graphics

A configuração de sobreposição de cores apresentada na tabela de configuração da figura 7 especifica que todas as cores aumentarão um pixel quando encontrarem a cor preta, ou seja, a cor preta vai sobrepor todas as cores em um pixel. A cor vermelha aumentará um pixel quando encontrar a cor preta e a azul, e assim sucessivamente, conforme a configuração apresentada na tabela.



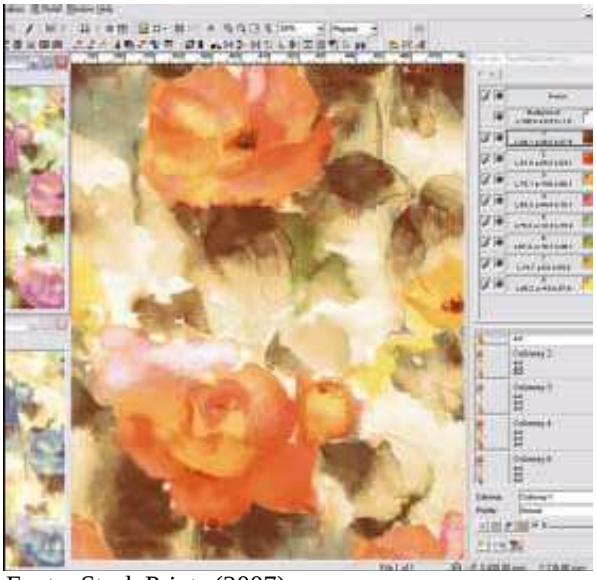
Fonte: Ned Graphics (2004).

Figura 7 – Tabela de configuração de sobreposição de cores

2.7.2 BestIMAGE

O besIMAGE (STORK PRINTS, 2007) também é um software para desenvolvimento de padrões têxteis. Esse apresenta funcionalidades similares ao Desing and Repeat Pro (NED GRAPHICS, 2004), tais como, diversos tipos de repetições e separação de cores automática

com ou sem sobreposição. Porém, este software possui alguns recursos diferenciados do Desing Repeat Pro (NED GRAPHICS, 2004), aceitando arquivos no padrão de cores RGB e trabalha com camadas. O fato de trabalhar em camadas é um diferencial importante, pois ao editar uma camada pode-se deixar as outras intactas, o que facilita o trabalho do usuário. A figura 8 mostra a interface do software BestIMAGE.



Fonte: Stork Prints (2007).

Figura 8 – BestIMAGE da Strok Prints

3 DESENVOLVIMENTO

Este capítulo descreve a especificação do projeto e detalha funcionalidades da ferramenta desenvolvida.

3.1 REQUISITOS

A ferramenta implementada neste trabalho deve possibilitar:

a) carregar e exibir uma imagem indexada (Requisito Funcional - RF);
b) possibilitar a visualização da imagem carregada em diversas formas de repetições (RF);
c) possibilitar a transformação das duas formas de repetição em uma repetição do tipo *straight repeat* (conforme mostrado na figura 2) (RF);
d) possibilitar copiar partes da imagem, formando uma janela com repositório[2] de motivos, para colar posteriormente em outros locais da imagem (RF);
e) permitir trocar ponto de origem[3] da imagem (RF);
f) permitir configurar sobreposição de cores (RF);
g) automatizar o processo de separação de cores, com ou sem sobreposição de cores conforme configurado na ferramenta (RF);
h) ser implementada usando a linguagem de programação Java (Requisito Não Funcional – RNF);
i) utilizar a API 2D da linguagem Java para auxiliar na manipulação das imagens (RNF).

[2] Uma janela com n divisões mostrando os motivos disponíveis para serem aplicados na imagem.

[3] Ponto da imagem de coordenada x e y igual à zero.

3.2 ESPECIFICAÇÃO

Para especificação deste trabalho foi utilizada a *Unified Modeling Language* (UML), com auxílio da ferramenta Enterprise Architect (SPARX SYSTEMS, 2007), para gerar os diagramas de casos de uso, de classe e de seqüência.

3.2.1 Diagrama de casos de uso

O diagrama de caso de uso ilustrado na figura 9 retrata todos os casos de uso existentes entre o usuário e o sistema.

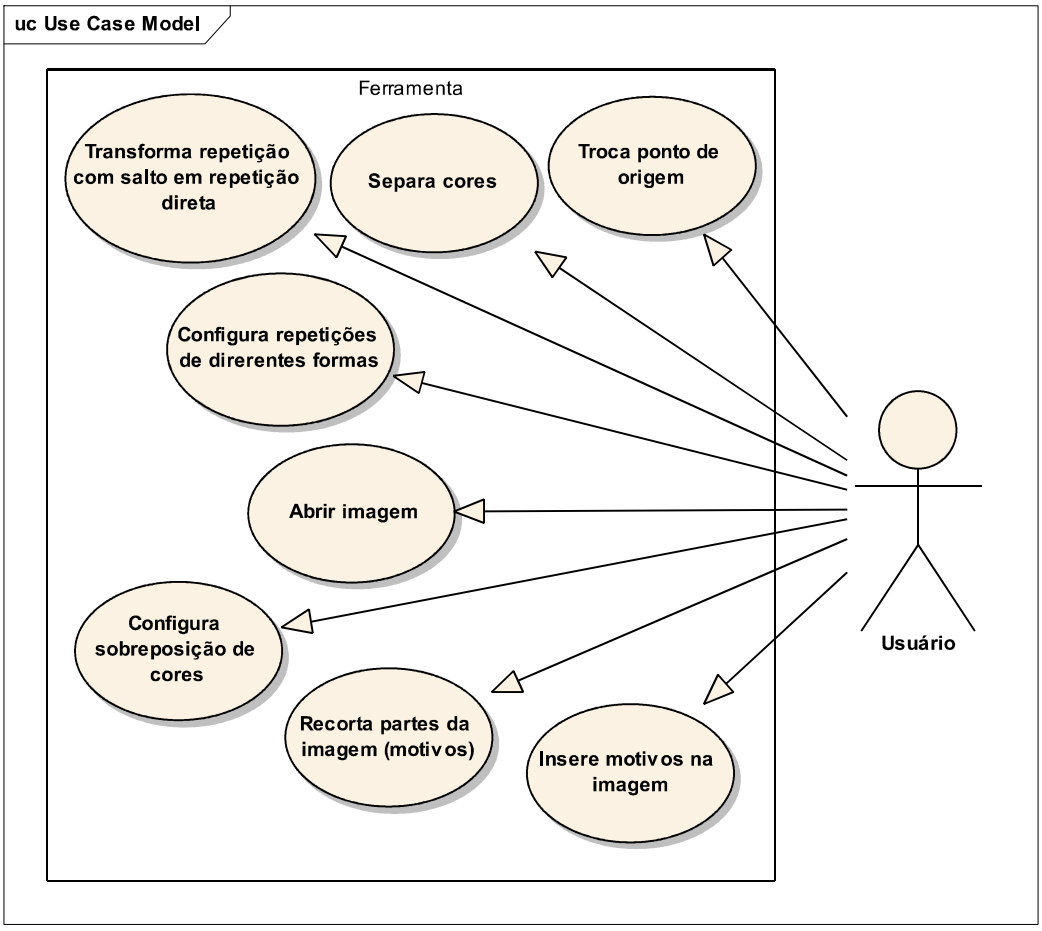


Figura 9 – Diagrama de casos de uso

O caso de uso `Troca ponto de origem` representa a situação do usuário quando necessita trocar o ponto de origem de uma imagem, o que faz com que a coordenada `0,0` da

imagem seja trocada por outra especificada por ele. O caso de uso `Separa cores` representa a possibilidade do usuário gerar a separação de cores da imagem com o intuito de gravar as matrizes de impressão para a técnica de serigrafia que podem ser quadros no caso de impressão localizada ou cilindros no caso de impressão de padrões. O caso de uso `Transforma repetição com salto em repetição direta`, representa a possibilidade do usuário interagindo com a ferramenta transformar uma imagem que está com uma visualização com salto vertical em uma imagem com repetição direta, deixando a imagem com a mesma aparência que estava com a visualização com salto vertical. O caso de uso `Configura repetições de diferentes formas` mostra a possibilidade de o usuário visualizar a imagem com repetições de diferentes formas. O caso de uso `Abrir imagem` mostra possibilidade do usuário abrir uma imagem na ferramenta. O caso de uso `Configura sobreposição de cores` ilustra a possibilidade do usuário configurar a sobreposição de cores para a separação de cores da imagem. O caso de uso `Recorta partes da imagem` mostra a possibilidade do usuário recortar partes da imagem para armazenar em uma base de motivos. Já o caso de uso `Insere motivos na imagem` ilustra a possibilidade do usuário inserir algum dos motivos da base de motivos na imagem.

3.2.2 Diagrama de Classes

O diagrama de classes ilustrado na figura 10 mostra todas as classes envolvidas no projeto da ferramenta.

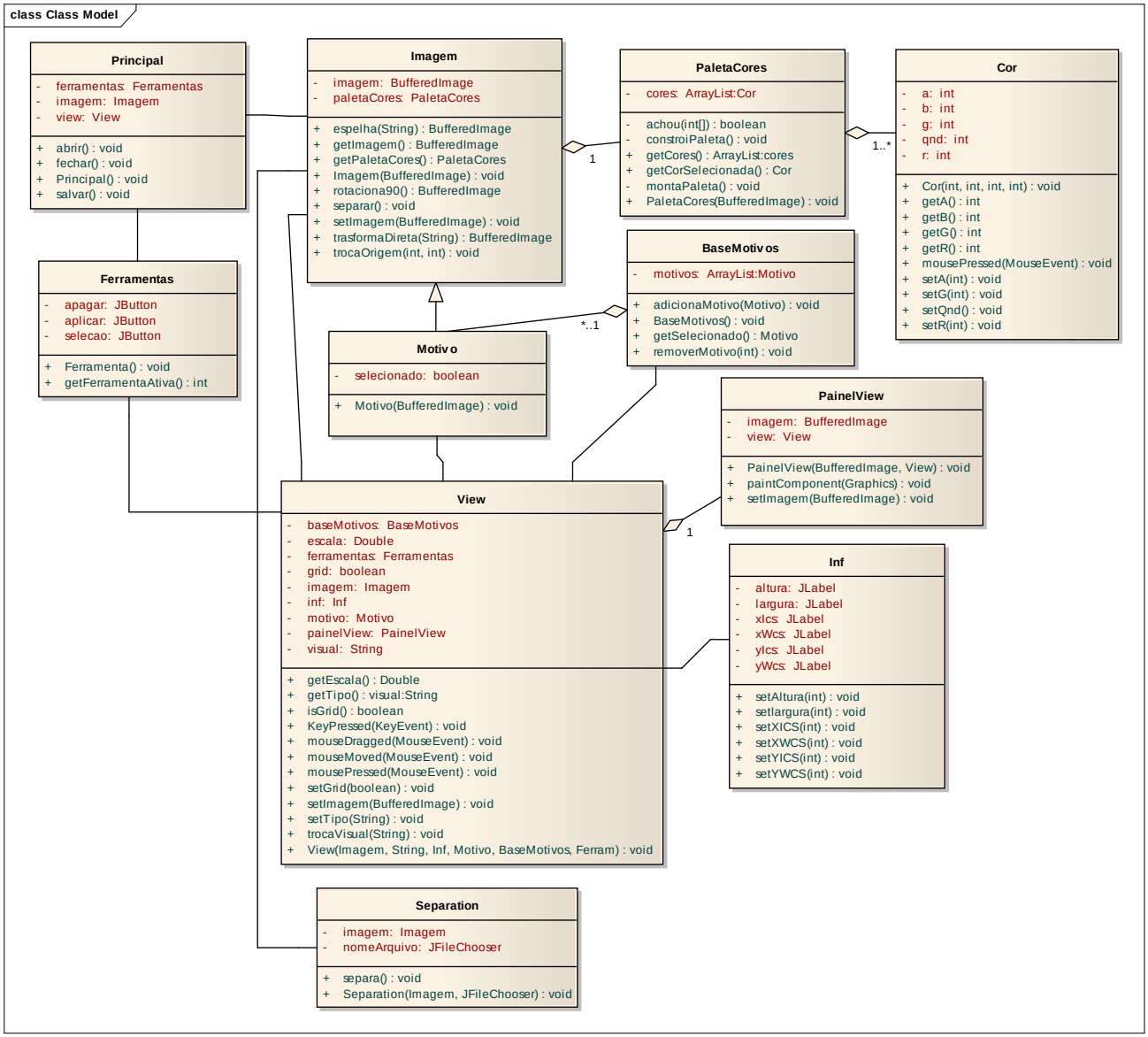


Figura 10 – Diagrama de classes

Conforme mostra o diagrama de classes da figura 10, a classe `Imagem` possui um `BufferedImage` que guarda os dados da imagem propriamente dita, e agrega um objeto `PaletaCores` que é instanciado a partir dos dados da imagem contida no `BufferedImage`.

A classe `PaletaCores` é um agregado de objetos `Cor`, o que deixa clara a estrutura básica de uma imagem no projeto da ferramenta, ou seja, uma imagem é composta por uma paleta de cores, que por sua vez é composta por cores. Esta estrutura foi definida, pois as operações da ferramenta trabalham diretamente manipulando os pixeis da imagem em função de suas cores. Uma amostra disso é o processo de separação de cores, que consistem em gerar uma nova imagem *bitmap* com cores preta e branca para cor da paleta. Outra amostra é o processo de seleção de motivos, que consiste em criar uma sub imagem a partir de uma área seleciona na imagem, para qual é necessário saber a cor da paleta selecionada para configurar

esta cor como transparente, o qual até o presente momento ainda não foi implementado.

A classe `Motivo` é uma estenção da classe `Imagem`, pois um `Motivo` é um tipo de `Imagem`, ou seja, um `Motivo` possui uma `PaletaCores` composta por objetos da classe `Cor`. Esta classe serve para especializar um tipo de `Imagem`. No seu construtor recebe um `BufferedImage` da área selecionada em uma `Imagem`.

A classe `BaseMotivos` é um agregado de `Motivos`. Esta classe serve para armazenar os `Motivos` e possui métodos para manipular estes. O método `adicionaMotivo`() é responsável por adicionar um `Motivo` na base, o método `removeMotivo`() por excluir um `Motivo` da base e o método `getSelecionado`() retorna o `Motivo` que está atualmente selecionado.

A classe `View` que gerencia a visualização das imagens, agrega um objeto `PainelView` que é responsável pelo processo de *rendering* das imagens.

A classe `Principal` é a interface do sistema. É através dela que o usuário interage com ferramenta solicitando as operações que deseja fazer com as imagens.

A classe `Inf` é responsável por exibir informações sobre coordenadas de tela e da `Imagem`. Estas informações são largura e altura da `Imagem`, coordenadas locais e coordenadas de trabalho.

A classe `Ferramentas` é uma barra de ferramentas que possui três (3) botões. Cada botão representa uma ferramenta. Esta classe possui um método `getFerramentaAtiva`(), o qual informa qual é a ferramenta que está ativa no momento.

A classe `Separation` é responsável pela separação de cores da `Imagem`. Quando o usuário solicita a separação de cores, é instanciado um objeto desta classe, o qual possui um método `separa`() que é executado no seu construtor, gerando assim a separação de cores em um diretório especificado pelo usuário.

3.2.3 Diagrama de seqüência

O diagrama de seqüência apresentado na figura 11 mostra a troca de mensagens entre alguns objetos da ferramenta. O digrama demonstra uma possível seqüência de operações de um usuário interagindo com a ferramenta. As operações são as seguintes:

a) abrir uma imagem indexada;

b) trocar o tipo de visualização da imagem para salto vertical;

c) transformar a imagem para repetição direta;
d) gerar a separação de cores da imagem;
e) fechar a ferramenta.

No diagrama da figura 11 foram omitidas as instâncias de alguns objetos que não sofreram interações nesta situação. Estas instâncias foram omitidas para dar maior legibilidade ao diagrama. As instâncias que não foram apresentadas, mas que numa situação real existem são as seguintes:

a) uma da classe `Ferramentas`;
b) uma da classe `Inf`;
c) uma da classe `BaseMotivos`;
d) uma da classe `Motivos`.

Estas instâncias ocorrem logo após a carga da ferramenta, e são passadas como parâmetro para o construtor da classe do objeto `View`.

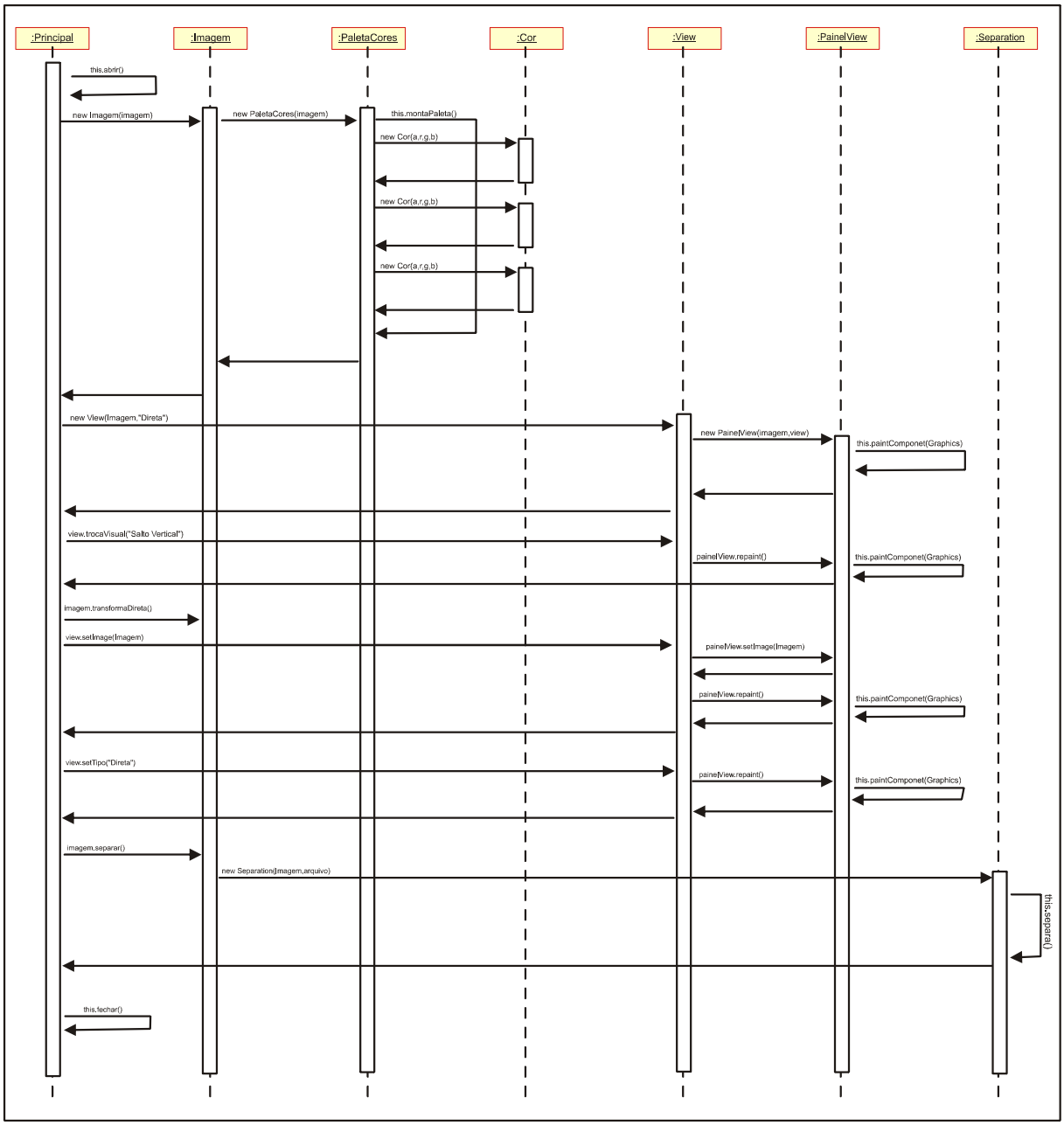


Figura 11 – Diagrama de seqüência 1

Outra possível seqüência de operações apresentada no diagrama de seqüência da figura 12 pode ser a seguinte:

a) abrir uma imagem indexada;

b) trocar o ponto de origem da imagem;

c) salvar a imagem;

d) fechar a ferramenta.

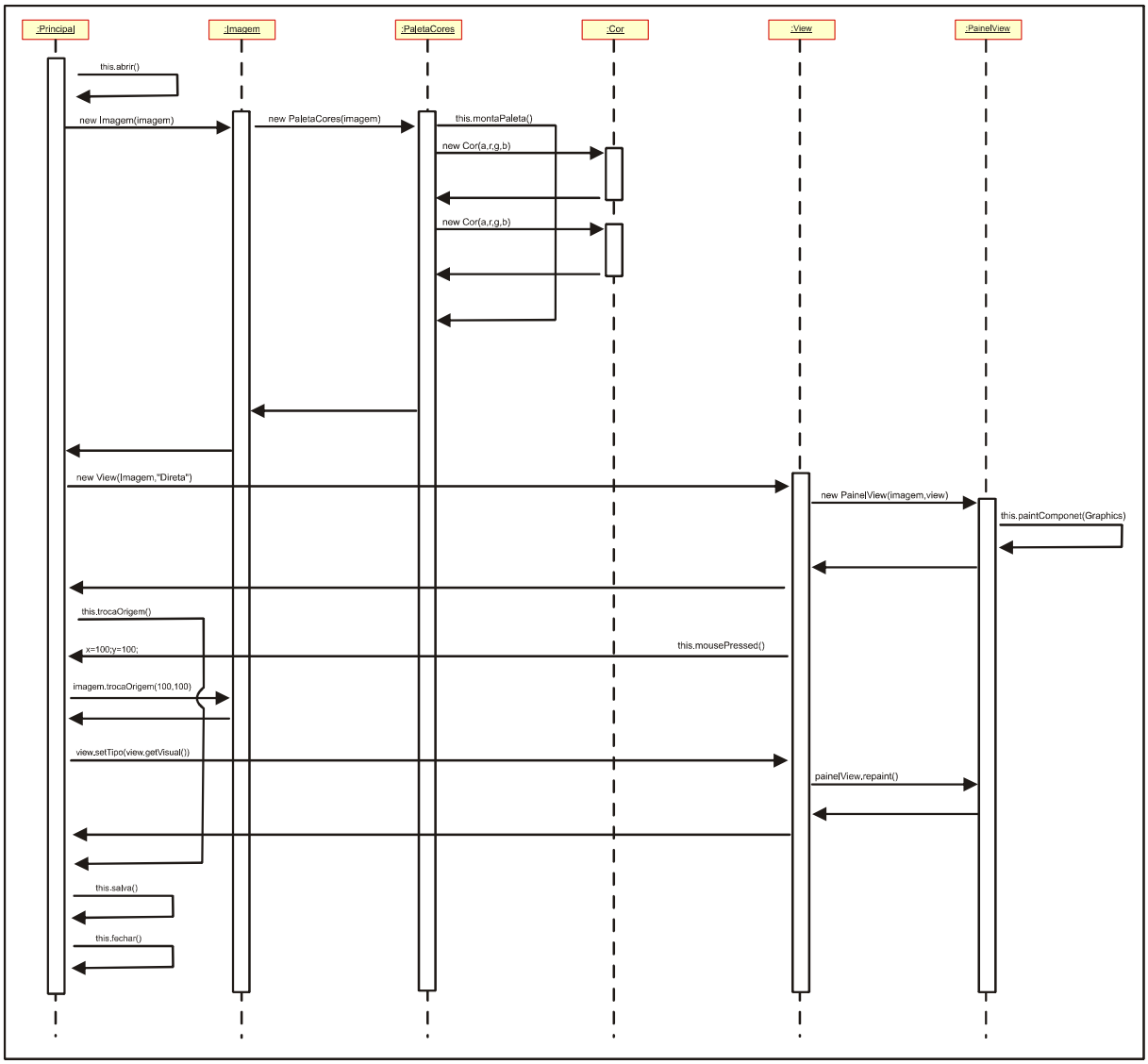


Figura 12 – Diagrama de seqüência 2

3.3 IMPLEMENTAÇÃO

Esta seção apresenta considerações sobre a implementação do protótipo da ferramenta, tais como métodos desenvolvidos, operacionalidade da implemantação e ferramentas utilizadas.

3.3.1 Métodos da classe `Imagem`

Conforme mostra o diagrama de classes apresentado na figura 10, foram implementados na classe `Imagem` métodos para manipulação de imagens. Estes métodos atuam diretamente sobre as componentes da `Imagem`. Nas seções a seguir serão apresentados alguns destes métodos e como foram desenvolvidos.

3.3.1.1 Método troca origem da `Imagem`

A classe `Imagem` possui o método `trocaOrigem(x,y)`, o qual recebe como parâmetro as coordenadas do novo ponto de origem. Conforme mostra o diagrama de seqüência da figura 12, quando o usuário solicita trocar a origem da imagem, o programa principal fica aguardando o método `mousePressed()` da classe `View` retornar a nova coordenada, ou seja, sob qual coordenada da `Imagem` foi clicado.

A figura 13 mostra, da esquerda para direita, a imagem com o ponto de origem original, retângulo amarelo no canto superior esquerdo, a imagem com o novo ponto de origem definido, retângulo verde no centro com seta apontando para o amarelo e a imagem com o ponto de origem trocado, retângulo verde no canto superior esquerdo.

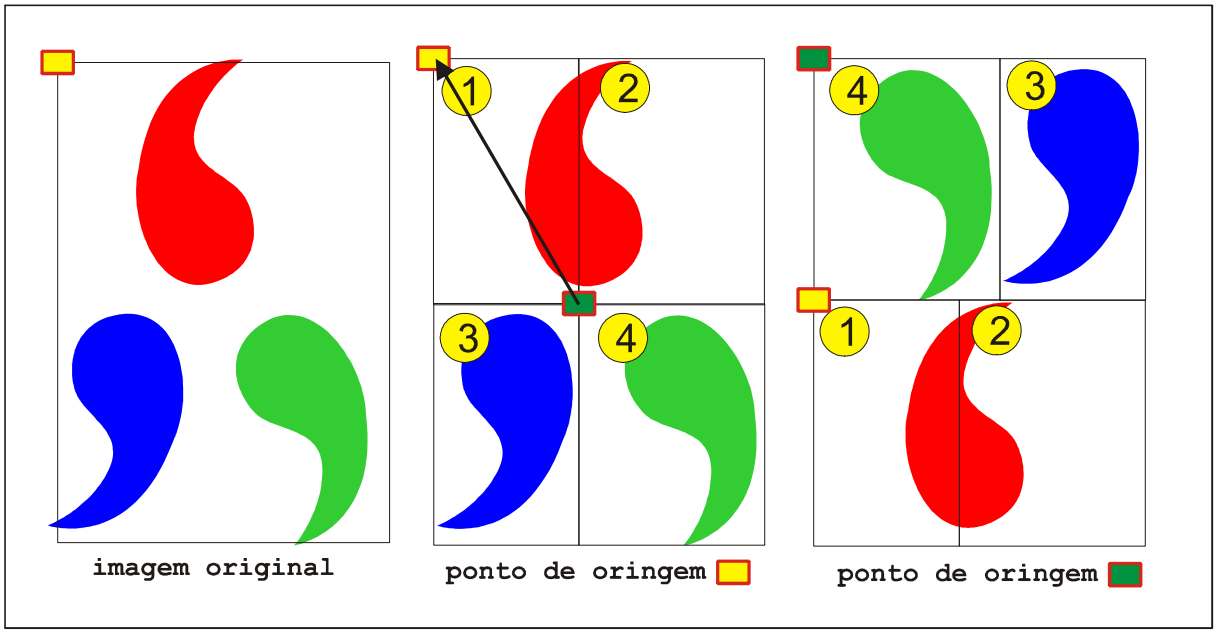


Figura 13 – Troca do ponto de origem da `Imagem`

Para implementação deste método são utilizados os métodos `getDataElements()` e

`setDataElements()` da classe `SampleModel`, que permitem copiar uma área retangular de um `DataBuffer` original para um outro novo `DataBuffer`. A operação de troca do ponto de origem da `Imagem` se resume em dividir a `Imagem` em quatro áreas retangulares e reposicionando estas áreas (figura 13). O quadro 9 mostra o código completo do método `trocaOrigem()` implementado no protótipo da ferramenta.

```
private void trocaOrigem(int x,int y){
 //pego as diemnsões da imagem antiga
 int w = imagem.getWidth();
 int h = imagem.getHeight();
 WritableRaster raster         = imagem.getRaster();
 ColorModel     cm             = imagem.getColorModel();
 SampleModel    smOriginal     = raster.getSampleModel();
 DataBuffer     dbOriginal     = raster.getDataBuffer();
 //pega o tipo da imagem original para criar a nova com o mesmo tipo
 int tipo = imagem.getType();
 //cria uma nova bufferd imagem com o dobro da largura
 BufferedImage nova = new BufferedImage(w,h,tipo, (IndexColorModel) cm);
 //pegar SampleModel e DataBuffer da nova imagem
 WritableRaster rasterNova         = nova.getRaster();
 SampleModel    smNova             = rasterNova.getSampleModel();
 DataBuffer     dbNova             = rasterNova.getDataBuffer();
 //divide a imagem em 4 partes e remonta a imagem com a nova origem
 smNova.setDataElements(0,0,(w-x),(h-y),smOriginal
       .getDataElements(x,y,(w-x),(h-y),null,dbOriginal),dbNova);
 smNova.setDataElements((w-x),(h-y),x,y,smOriginal
        .getDataElements(0,0,x,y,null,dbOriginal),dbNova);
 smNova.setDataElements(0,(h-y),(w-x),y,smOriginal
        .getDataElements(x,0,(w-x),y,null,dbOriginal),dbNova);
 smNova.setDataElements((w-x),0,x,(h-y),smOriginal
       .getDataElements(0,y,x,(h-y),null,dbOriginal),dbNova);
JOptionPane.showMessageDialog(null, "Operação realizada com sucesso!");
}
```

Quadro 9 – Código do método `trocaOrigem()` da classe `Imagem`

3.3.1.2 Método espelhamento horizontal e vertical

Foi implementado o método utilitário `espelha()` na classe `Imagem`, o qual recebe como parâmetro o sentido que o usuário quer espelhar a `Imagem`, que pode ser horizontal ou vertical.

Na implementação deste método utilizou-se dois métodos da classe `SampleModel`, que permitem acessar pixeis de uma coordenada em um `DataBuffer` original e configurá-los em um novo `DataBuffer`. O quadro 10 mostra o código do método `espelha()` da classe `Imagem`.

```
private BufferedImage espelha(BufferedImage imgOriginal,String sentido){
 //pego as diemnsões da imagem original
   int w = imgOriginal.getWidth();
   int h = imgOriginal.getHeight();
   WritableRaster raster         = imgOriginal.getRaster();
   ColorModel     cm             = imgOriginal.getColorModel();
   //pega o tipo da imagem original para criar a nova com o mesmo tipo
   SampleModel    smOriginal     = raster.getSampleModel();
   DataBuffer     dbOriginal     = raster.getDataBuffer();
   //pega o tipo da imagem original
   int tipo = imagem.getType();
   //cria uma nova imagem com as mesmas características da original
   BufferedImage imgNova = new BufferedImage(w,h,tipo,(IndexColorModel)cm);
   WritableRaster rasterNova     = imgNova.getRaster();
   SampleModel    smNova         = rasterNova.getSampleModel();
   DataBuffer     dbNova         = rasterNova.getDataBuffer();
   //trasfere os pixeis de forma que deixe a imagem espelhada
   int cont;
   //espelhamento horizontal
   if (sentido.equalsIgnoreCase("h")){
      for (int y=0;y<h;y++){
          cont =w-1;
          for(int x=0;x<w;x++){
              int [] vetor=new int[1];
              smOriginal.getPixel(x,y,vetor,dbOriginal);
              smNova.setPixel(cont,y,vetor,dbNova);
              cont--; } }
   }else{
   //espelhamento vertical
     for (int x=0;x<w;x++){
         cont =h-1;
         for(int y=0;y<h;y++){
            int [] vetor=new int[1];
            smOriginal.getPixel(x,y,vetor,dbOriginal);
            smNova.setPixel(x,cont,vetor,dbNova);
            cont--; } }
   }
}
```

Quadro 10 – Código do método `espelha()` da classe `Imagem`

3.3.1.3 Método rotação da `Imagem`

Na classe `Imagem` também foi desenvolvido o método `rotaciona90()`, o qual rotaciona a imagem em 90 graus.

Para a implementar este método foi utilizado a operação de transposição de uma matriz, cuja operação é fazer o elemento A[i,j] = B[j,i]. Durante a implementação verificou-se que a transposta gera uma matriz rotacionada em 90 graus e espelhada horizontalmente. Por este motivo após invocar o método `rotaciona90()` é invocado o método `espelha()`, passando como parâmetro o sentido horizontal, o que deixa a `Imagem` rotacionada sem espelhamento.

Para acessar e configurar os elementos da matriz da `Imagem`, no caso os pixeis, utilizou-se os métodos `getPixel()` e `setPixel` da classe `SampleModel`, conforme mostra o código do quadro 11.

```
private BufferedImage rotaciona90(BufferedImage imgOriginal){
 //pego as diemnsões da imagem original
 int w = imgOriginal.getWidth();
 int h = imgOriginal.getHeight();
 WritableRaster raster         = imgOriginal.getRaster();
 ColorModel     cm             = imgOriginal.getColorModel();
 //pega o tipo da imagem original para criar a nova com o mesmo tipo
 SampleModel    smOriginal     = raster.getSampleModel();
 DataBuffer     dbOriginal     = raster.getDataBuffer();
 //pega o tipo da imagem original
 int tipo = imagem.getType();
 //cria uma nova imagem com as mesmas características da original
 //porem com tamanho invertido, a lagura se torna a altura
 BufferedImage imgNova = new BufferedImage(h,w,tipo,IndexColorModel)cm);
 WritableRaster rasterNova     = imgNova.getRaster();
 SampleModel    smNova         = rasterNova.getSampleModel();
 DataBuffer     dbNova         = rasterNova.getDataBuffer();
 int [] vetor=new int[1];
 //faz a transposta da imagem
 for (int i=0;i<w;i++){
     for(int j=0;j<h;j++){
        smOriginal.getPixel(i,j,vetor,dbOriginal);
        smNova.setPixel(j,i,vetor,dbNova);} }
 //espelha a imagem na horizontal
 imagem = espelha(imgNova,"h");
 return imagem;
 }
```

Quadro 11 – Código do método `rotaciona90()` da classe `Imagem`

3.3.1.4 Método `transf()` da classe `Imagem`

Na classe `Imagem` foi implementado o método `transf()`, o qual é responsável por transformar uma imagem para repetição direta. Antes de invocar este método é preciso perguntar para `View` qual o tipo de repetição que a imagem está sendo exibida, pois esta transformação ocorre em função do tipo de visualização que está sendo exibido.

O diagrama de seqüência apresentado na figura 11 mostra uma situação em que o usuário solicita a transformação para repetição direta em uma imagem com visualização do tipo salto vertical. Neste diagrama da figura 11 é possível observar a troca de mensagens entre os objetos, notando que é enviada a mensagem para a `View` perguntando seu tipo de

visualização. Após isso é invocado o método `transf()`, passando como parâmetro o tipo de visualização retornado pela `View`.

A figura 14 mostra um *printscreen* do protótipo da ferramenta na transformação de uma imagem com visualização de salto vertical transformada para repetição direta.

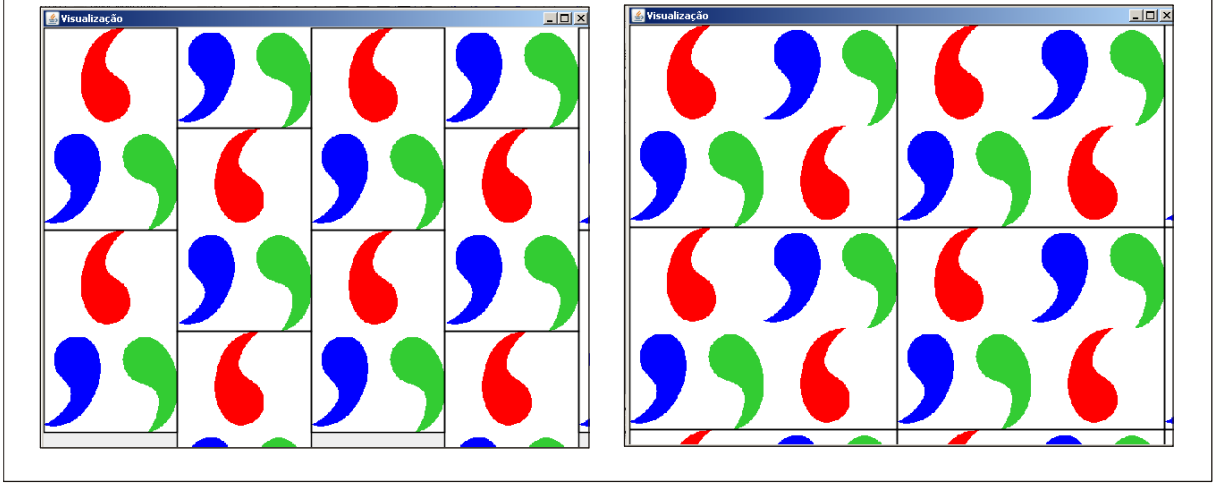


Figura 14 – Transformação de salto vertical para repetição direta

Na figura 14 é possível observar que a imagem transformada para repetição direta precisou ser duplicada no seu tamanho na largura. Isso é necessário neste tipo de transformação para manter a `Imagem` sem deslocamento vertical com a mesma aparência de como se tivesse o deslocamento vertical. Na figura 15 é possível observar a `Imagem` sendo aumentada em sua largura na transformação para repetição direta. É possível notar também a cópia de áreas retangulares. Esta transformação ocorre pela execução do código do quadro 12.

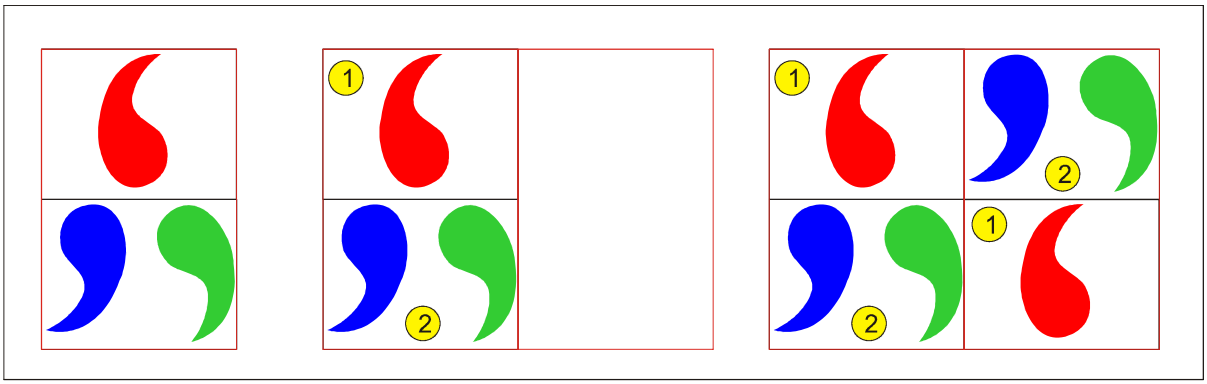


Figura 15 – Passo a passo da transformação com salto vertical para repetição direta

```
//pego as diemnsões da imagem original
 int w = imagem.getWidth();
 int h = imagem.getHeight();
 WritableRaster raster         = imagem.getRaster();
 ColorModel     cm             = imagem.getColorModel();
 SampleModel    smOriginal     = raster.getSampleModel();
 DataBuffer     dbOriginal     = raster.getDataBuffer();
 //pega o tipo da imagem original para criar a nova com o mesmo tipo
 int tipo = imagem.getType();
 //cria uma nova bufferd imagem com o dobro da largura
 BufferedImage nova = new BufferedImage(w*2,h,tipo,IndexColorModel)cm);
 //pegar SampleModel e DataBuffer da nova imagem
 WritableRaster rasterNova         = nova.getRaster();
 SampleModel    smNova             = rasterNova.getSampleModel();
 DataBuffer     dbNova             = rasterNova.getDataBuffer();
 //copia partes da imagem original para imagem nova
smNova.setDataElements(0,0,w,h,smOriginal
      .getDataElements(0,0,w,h,null,dbOriginal),dbNova);

smNova.setDataElements(w,h/2,w,h/2,smOriginal
      .getDataElements(0,0,w,h/2,null,dbOriginal),dbNova);

smNova.setDataElements(w,0,w,h/2,smOriginal
      .getDataElements(0,h/2,w,h/2,null,dbOriginal),dbNova);
```

Quadro 12 – Código para transformação da visualização com salto vertical para repetição direta

3.3.2 Paleta de cores

A paleta de cores da `Imagem` foi implementa na classe `PaletaCor` conforme mostra o diagrama de classes da figura 10. Uma objeto `PaletaCor` é um agregado de objetos da classe `Cor` armazenados em um `ArrayList`.

No método construtor da classe `PaletaCor` é invocado um método responsável pela montagem da paleta de cores, que é o método `constroiPaleta()`.

O método `constroipaleta()` utiliza o método `getRBG()` da classe `BufferedImage`, o qual retorna um vetor com os pixeis da `Imagem`, cujo elementos são números do tipo `int`. Cada elemento do tipo `int` deste vetor contém as quatro componentes *Alpha*, *Red*, *Green* e *Blue* (ARGB) dos pixeis, que correspondem respectivamente em transparência do pixel, que são as componentes *red*, componente *green* e a componente *blue*. Um `int` pode armazenar quatro *bytes*, ou seja, cada componente do ARGB é um *byte* deste `int`. Para acessar isoladamente cada uma destas componentes foi utilizado o código mostrado no quadro 13.

```
int [] corDaPaleta = new int[4];
corDaPaleta[0]=((coresRGB[x]&0x00FF0000)>>>24); /*  alfa */
corDaPaleta[1]=((coresRGB[x]&0x00FF0000)>>>16); /*  R    */
corDaPaleta[2]=((coresRGB[x]&0x0000FF00)>>>8);  /*  G    */
corDaPaleta[3]= (coresRGB[x]&0x000000FF) ;      /*  B    */
```

Quadro 13 – Acessando os `bytes` de um `int`

O método `constroiPaleta()` varre o vetor de pixeis retornado pelo método `getRGB()` e vai adicionando as cores dos pixeis em um `ArrayList` de objetos de `Cor` sem repetição de cor. Para evitar a repetição de cores na paleta, foi criado o método `achou()`, o qual verifica no `ArrayList` de cores se a `Cor` já foi adicionada. Caso a `Cor` não tenha sido adicionada é instanciado um novo objeto de `Cor` e adicionado no `ArrayList` de cores.

O quadro 14 apresento o código completo do método `constroiPaleta()`.

```
private void constroiPalheta() {

  w = this.imagem.getWidth();
  h = this.imagem.getHeight();
  //cria um vetor para armazenar todas as cores da imagem
  int [] coresRGB = new int[w*h];
  //pega todas as cores da imagem e armazena nesse vetor
  this.imagem.getRGB(0,0,w,h,coresRGB,0,w);
  //varre o vetor, e adicionas as cores sem repetição na paleta
  for (int x = 0; x<(w*h); x++){
     int [] corDaPaleta = new int[4];
     corDaPaleta[0]=((coresRGB[x]&0x00FF0000)>>>24); /*  alfa */
     corDaPaleta[1]=((coresRGB[x]&0x00FF0000)>>>16); /*  R    */
     corDaPaleta[2]=((coresRGB[x]&0x0000FF00)>>>8);  /*  G    */
     corDaPaleta[3]= (coresRGB[x]&0x000000FF) ;      /*  B    */
     //verifica se a cor já esta adicionada na paleta
     if (!achou(corDaPaleta)){
        Cor nova = new cor(corDaPaleta[0],
                           corDaPaleta[1],
                           corDaPaleta[2],
                           corDaPaleta[3]);
        cores.add(nova); }
        }
    }
```

Quadro 14 – Método `constroiPaleta()` da classe PaletaCor

Ainda na construção da paleta de cores é calculado o histograma da imagem que é mostrado na paleta de cores sob cada cor e que consistem em calcular quantos pixeis existem de cada cor na imagem. Este recurso pode ser utilizado para calcular a área de cada cor separada em função da resolução, o que pode ser útil para saber-se o gasto de tinta no processo de impressão.

3.3.3 Separação de cores

Para automatizar o processo de separação de cores foi criado uma classe `Separation`, que possui um método `separa()`, o qual é responsável pela separação de cores da imagem. O processo de separação de cores poderia ser apenas um método da classe `Imagem`, mas por questões de modularização e legibilidade do projeto, optou-se por esta opção de criar uma classe específica para este procedimento.

O construtor de `Separation` recebe como parâmetro uma `Imagem`, contendo os dados da imagem tais como paleta de cores e pixeis da imagem, e um `JFileChooser` contendo o caminho e nome do arquivo onde serão salvos os arquivos com as separações de cores.

Ao instanciar um objeto `Separation`, o seu construtor invoca o método `separa()`, o qual inicia o processo de separação de cores.

O método `separa()` varre todos os pixeis da imagem o número de vezes que contiverem cores na paleta de cores da imagem. Por exemplo, se a imagem tiver três cores, a imagem será percorrida três vezes. Esta varredura ocorre afim de que se compare a cor atual da paleta de cores, com o pixel da `Imagem`. Caso a cor atual da paleta seja igual a cor do pixel examinado na `Imagem`, este pixel é configurado na cor preta em uma nova `Imagem`, e caso sejam de cores diferentes o pixel é configurado na cor branca.

Este processo acontece com a intenção de criar uma imagem preta e branca para cada cor da paleta de cores da imagem. Estas imagens geradas são salvas em arquivos no formato GIF em disco, com o nome especificado pelo usuário acrescido de um número seqüencial crescente iniciado em um (1), que serve para diferenciar os nomes dos arquivos. O quadro 15 mostra o código do método `separa()` da classe `Separation`.

A figura 16 mostra da esquerda para direita uma `Imagem` com 4 cores carregada pela ferramenta, juntamente com sua separação de cores gerada.

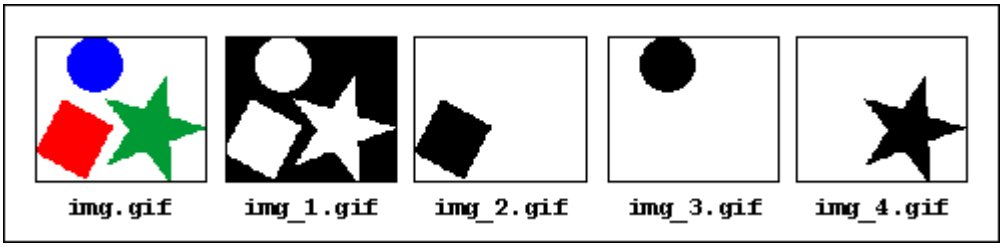


Figura 16 – Separação de cores de uma `Imagem`

```
public void separa(){
 int w = imagem.getWidth();
 int h = imagem.getHeight();
 Cor corDaVez;
 int r,g,b,rA,gA,bA;
 int contadorNome = 0;
 for(int numCores=0;numCores<cores.size();numCores++){
//cria uma imagem do tipo TYPE_BYTE_BINARY(12) com dim da imagem original
//para armazenar a cor separada
 BufferedImage  novaImagem = new BufferedImage(w,h,12);
//pega a cor da paleta da vez para fazer a separacao da cor
 corDaVez = (Cor) cores.get(numCores);
//usado para nomear as cores da separação
//pega os componentes RGB da cor da paleta
  r = corDaVez.getR();g = corDaVez.getG();b = corDaVez.getB();
  int pixel;
  for (int x = 0; x<w; x++)
      for (int y =0;y<h;y++){
      //pega as componentes RGB do pixel atual da imagem
      pixel = imagem.getRGB(x,y);
      rA=((pixel&0x00FF0000)>>>16); // Red
      gA=((pixel&0x0000FF00)>>>8);  // Green level
      bA= (pixel&0x000000FF);       // Blue level
//compara com o atual da paleta, se for igual copia para uma nova
//imagem preto e branco
      if((rA==r)&&(gA==g)&&(bA==b))
           novaImagem.setRGB(x,y,new Color(0,0,0).getRGB());
      else
           novaImagem.setRGB(x,y,new Color(255,255,255).getRGB());
  }//fim do laço
 String nome = nomeArquivo.getSelectedFile()
              .toString()+"_"+contadorNome+".gif";
 File arquivo = new File(nome);
 try {ImageIO.write(novaImagem, "GIF", arquivo); }
 catch (IOException ex) { ex.printStackTrace();}
}//fim do for da paleta
JOptionPane.showMessageDialog(null,"Separação de cores concluída com
sucesso!");
}
```

Quadro 15 – Método `separa()` da classe `Separation`

3.3.4 Operacionalidade da implementação

Para rodar o protótipo da ferramenta é necessário executar o arquivo `Ferramenta.jar`, o qual pode ser feito através de um duplo *click* em cima do ícone do arquivo `Ferramenta.jar` ou em linha de comando digitando `java –jar Ferramenta.jar`. A figura 17 mostra a interface do protótipo da ferramenta rodando em plataforma Windows, após carregar uma imagem. Observando a figura 17 é possível notar a barra de menus superior, a paleta de cores, o painel de visualização e a caixa de informações. A paleta de cores situada no canto esquerdo da figura 17 exibe as cores da imagem com suas respectivas quantidades de *pixeis*.

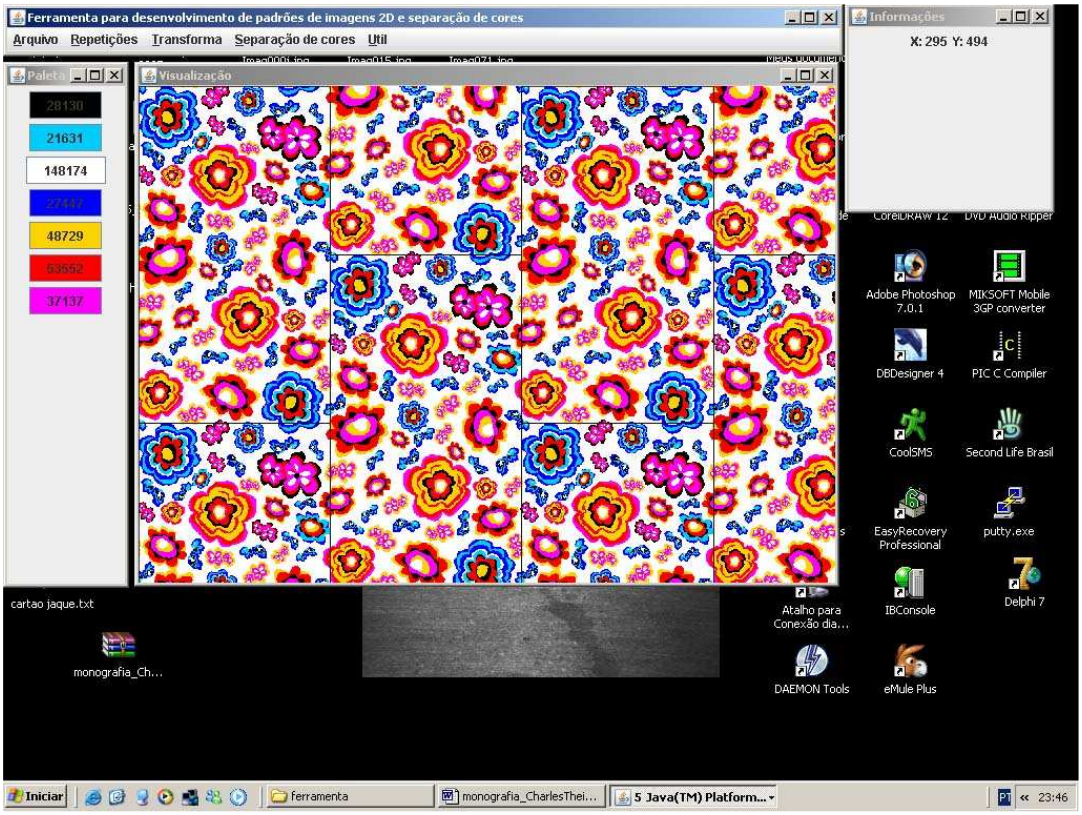


Figura 17 – Interface do protótipo da ferramenta

Através do menu `Arquivo` o usuário tem a opção de abrir e salvar uma imagem e fechar a ferramenta. No menu `Repetições` o usuário pode trocar o tipo de repetição do padrão a fim de encontrar um tipo que melhor lhe agrade e também tem a opção de habilitar e desabilitar o *grid*, que serve para mostrar os limites da imagem. No menu `Transforma` o usuário pode transformar a repetição com salto vertical em repetição direta, mantendo a mesma aparência na repetição do padrão. No menu `Separação de cores` o usuário pode gerar a separação de cores da imagem. No menu `Útil` o usuário tem as opções de espelhamento horizontal, espelhamento vertical, rotação 90 graus e troca do ponto de origem da imagem.

3.3.5 Ferramentas utilizadas

Para o desenvolvimento deste trabalho foi utilizada a linguagem Java e a sua API 2D. O ambiente utilizado para o desenvolvimento foi o Netbeans ide 5.5.1 (NETBEANS, 2007). O sistema operacional no qual o ambiente de desenvolvimento rodou durante o desenvolvimento foi o Windows XP versão 5.1. A versão da máquina virtual Java utilizada foi a *VM* 1.6.02-b05, cujo fornecedor é a Sun Microsystems Inc (Sun Microsystems, 2005).

4 CONCLUSÕES

A linguagem Java, juntamente com sua API 2D, forneceu recursos suficientes para o desenvolvimento do protótipo da ferramenta.

O protótipo da ferramenta mostrou-se funcional, mesmo sem estar totalmente concluído. O processo de separação de cores, mesmo sem sobreposição, é uma automação bastante produtiva para a técnica das áreas de serigrafia, pois separar cores é algo extremamente repetitivo e operações deste tipo podem ser automatizadas. Em ferramentas comercias e populares como Corel e Photoshop, este processo precisa ser feito manualmente pelo usuário, ou seja, o usuário da ferramenta precisa selecionar a cor que deseja separar, criar uma nova imagem, pintar a cor e salvar a imagem. Este processo pode ser repetido diversas vezes, o que acaba tornando-se improdutivo, tornando muito útil o uso de ferramentas deste tipo para automatizar este processo, principalmente em linhas de produção em larga escala.

A visualização com salto vertical e sua transformação para repetição direta mostra como simplesmente deslocando verticalmente as repetições da imagem é possível melhorar significativamente a distribuição dos motivos.

A troca do ponto de origem da imagem permite que a imagem seja salva em disco para possível abertura e edição em outro software convencional, a fim de melhorar as emendas do padrão de imagem, já que o foco deste protótipo não foi desenvolver ferramentas de edição de imagem.

Embora não tenha sido concluído ou implementado o processo de seleção e aplicação de motivos, foi feito um levantamento bibliográfico que traz embasamento para futura implementação. Na seção 2.6.2 são apresentados recursos da API 2D Java para desenho de formas geométricas que podem ser usados para selecionar áreas e acessar os pixeis desejados na seleção de sub imagens (`Motivos`).

4.1 EXTENSÕES

Uma sugestão seria implementar a parte de seleção e aplicação de motivos, já que o projeto inicial da ferramenta foi preparado para possuir estas funcionalidades, sendo que as classes de negócio já foram projetadas (figura 10), e esta funcionalidade deixaria o protótipo

da ferramenta ainda mais útil na questão de desenvolvimento de padrões de imagem.

Outra extensão seria implementar mais tipos de repetições e suas respectivas transformações para repetição direta. O processo de sobreposição de cores também pode ser implementado.

Uma outra possível extensão é implementar um módulo para quantização de cores (redução de cores) similar ao existente na ferramenta Desing and Repeat Pro (Ned Graphics, 2004), o que acabaria com a limitação deste protótipo no aspecto de que suporta apenas imagens indexadas. Seria interessante poder abrir imagens do modo de cores RGB, e convertê-las para o modo indexado, já que o resultado final precisa ser uma imagem indexada, para que as operações de separação de cores ocorram sobre as cores da paleta.

No Desing and Repeat Pro (Ned Graphics, 2004) é possível abrir uma imagem RGB que pode conter 16,7 milhões de cores e especificar a quantidade de cores que se deseja na imagem resultante. Nesta conversão também seria interessante poder selecionar algumas cores da imagem RGB com uma ferramenta tipo conta gotas, por exemplo, para fazer conversão, o que resultaria em uma imagem apenas com as cores selecionadas. As demais cores da imagem original devem ser unificadas por aproximação de tonalidade. Muito provável que será necessário um novo levantamento bibliográfico, pois estas técnicas não foram estudadas para o desenvolvimento desta ferramenta.

REFERÊNCIAS BIBLIOGRÁFICAS

CAZA, Michel. **Técnicas de serigrafia.** Barcelona: Blume, 1967.

CORRIGAN, John. **Computação gráfica:** segredos e soluções. Rio de Janeiro: Ciência Moderna, 1994.

FACON, Jacques. **Processamento e análise de imagens.** Embalse: EBAI, 1993.

FERNANDES, Antônio C. **Protótipo de visualizador para modelos de cor para medições de objetos em espectrofotômetros por reflectância.** 2002. 88 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Natturais, Universidade Regional de Blumenau, Blumenau.

FREE SOFTWARE FOUNDATION. **Why there are no GIF files on GNU web pages.** [S.l.], 2007. Disponível em: <http://www.gnu.org/philosophy/gif.html>. Acesso: em 20 set. 2007.

GONZALEZ, Rafael C.; WOODS, Richard E. **Processamento de imagens digitais.** São Paulo: Edgard Blücher, 2000.

MARINO, Mario. **Formatos de imagem e formatos de arquivos**. [S.l.], 1998. Disponível em: <http:// www.mariomarino.com.br/digital/formatos.htm >. Acesso: em 7 nov. 2007.

NED GRAPHICS. **Design and Repeat Pro**: help. Version 4.0. [S.l.], 2004. Documento eletrônico disponibilizado com a ferramenta.

NIEMEYER, Patrick; KNUDSEN, Jonathan. **Aprendendo Java.** Rio de Janeiro: Campus, 2000.

NETBEANS. **Using netbeans 5.5.1.** [S.l.]. 2007. Disponível em: <http://www.netbeans.org/kb/55/using-netbeans/index.html> Acesso em: 2 maio 2008.

SPARX SYSTEMS. **Enterprise Architect**: UML design tools and UML CASE tools for software development. [Creswick], 2007. Disponível em: <http://www.sparxsystems.com/products/ea.html>. Acesso em: 22 mar. 2008.

STORK PRINTS. **BestIMAGE.** [S.l.], 2007. Disponível em: <http://www.storkprints.com.br/prints-en/page.html-ch=DEF&id=85.htm>. Acesso em: 21 set. 2007.

SUN MICROSYSTEMS. **The Source for Java technology**. [S.l.], [2005?] Disponível em: <http://java.sun.com/>. Acesso em: 10 maio 2008.

_____. **2D Graphics**. [S.l.]. 2008. Disponível em: <http://java.sun.com/docs/books/tutorial/2d/index.html>. Acesso em: 05 abr. 2008.

_____. **Java™ 2 Platform, Standard Edition, 6 API specification**. [S.l.]. 2006. Disponível em: <http://java.sun.com/javase/6/docs/api/index.html>. Acesso em: 05 abr. 2008.

_____. **Java™ image I/O API guide**. [S.l.]. 2001. Disponível em: <http://java.sun.com/javase/6/docs/technotes/guides/imageio/spec/title.fm.html>. Acesso em: 05 abr. 2008.

_____. **Programmer's guide to the JavaTM 2D API**. [S.l.]. 2003. Disponível em: <http://java.sun.com/javase/6/docs/technotes/guides/2d/spec/j2d-bookTOC.html>. Acesso em: 05 abr. 2008.