,

WrapImaJ User Guide

Rémy Malgouyres

`remy@malgouyres.org`

Contents

1 The Core *Application Programming Interface* 2
- 1.1 Class Diagram for the `core` package . . . 3
- 1.2 Loading and displaying an image . . . 3
- 1.3 Converting to 8 bits gray levels, saving an image . . . 4
- 1.4 Blurring an image and Thresholding (e.g. Otsu) . . . 4
- 1.5 Histogram manipulation . . . 5
- 1.6 Brightness/Contrast adjustment . . . 6
- 1.7 Low Level Access to The Voxel's Values . . . 7

2 The Wrapper for *ImageJ* 9

3 The Package for Connected Components 11
- 3.1 Class Diagram for the `connectivity` package . . . 11
- 3.2 Labeling Connected Components and basic Filtering . . . 12
- 3.3 Advanced Connected Components Filtering . . . 14

Introduction

What is *WrapImaJ* about ?

WrapImaJ purposes to be a multi-platform wrapper for different Image Processing systems using the *Java* programming language. The purpose of *WrapImaJ* is not to combine an exhaustive collection of all functionalities of different imaging system, but to offer a simple, concise Application Programming Interface (*API*) allowing to develop imaging software, the source code of which is independent from the underlying imaging system on which it relies.

In it's current form, it only wraps basic functionalities of *ImageJ*. The developers of *WrapImaJ* intend the library to support compatibilty with the main imaging systems available in the *Java* language and broadly used in the field of life sciences.

This guide is an introductory guide intended to hep developpers use *WrapImaJ*, and also possibly contribute to the development of *WrapImaJ* or *OpenSource* software built on *WrapImaJ*.

Outline of this guide

The content of this guide is organized as follows:

- In Section 1, the core of the *Application Programming Interface* is presented. It consist of the public interface which common to all the wrappers implemented in *WrapImaJ*.

- In Section 2, we present the *ImageJ* wrapper, which implements, using *ImageJ*, classes implementing all the interfaces which appear in the core *API*.

  In addition to these public interfaces, the only methods in the wrapper which are public are the constructors (or possibly *factory*-like methods) which allow to create instances. Once instances are created, only the methods of the public interface in the core *API* should be used. This mainly makes future program implementations independent from the particular wrapper used.

- In Section 3, we present the first module developed using the core *API*, as a proof of concept. This module deals with connected components in binary images, their labelling and filtering.

Current limitations

At the time this guide is written, only an *ImageJ* wrapper is written. The bulk of the development targets static 3D images, which are quickly to be converted to 8 bits per pixels images to work on histograms.

Even though the *ImageJ* wrapper has been successfully tested, the project is still experimental and the core *API* is liable to change. Though the changes will focus on extension, for the purpose of conciseness, we cannot yet be sure that it will be completely backward compatible. The current functionalities, though, should still be provided with a similar syntax in the future.

1 The Core *Application Programming Interface*

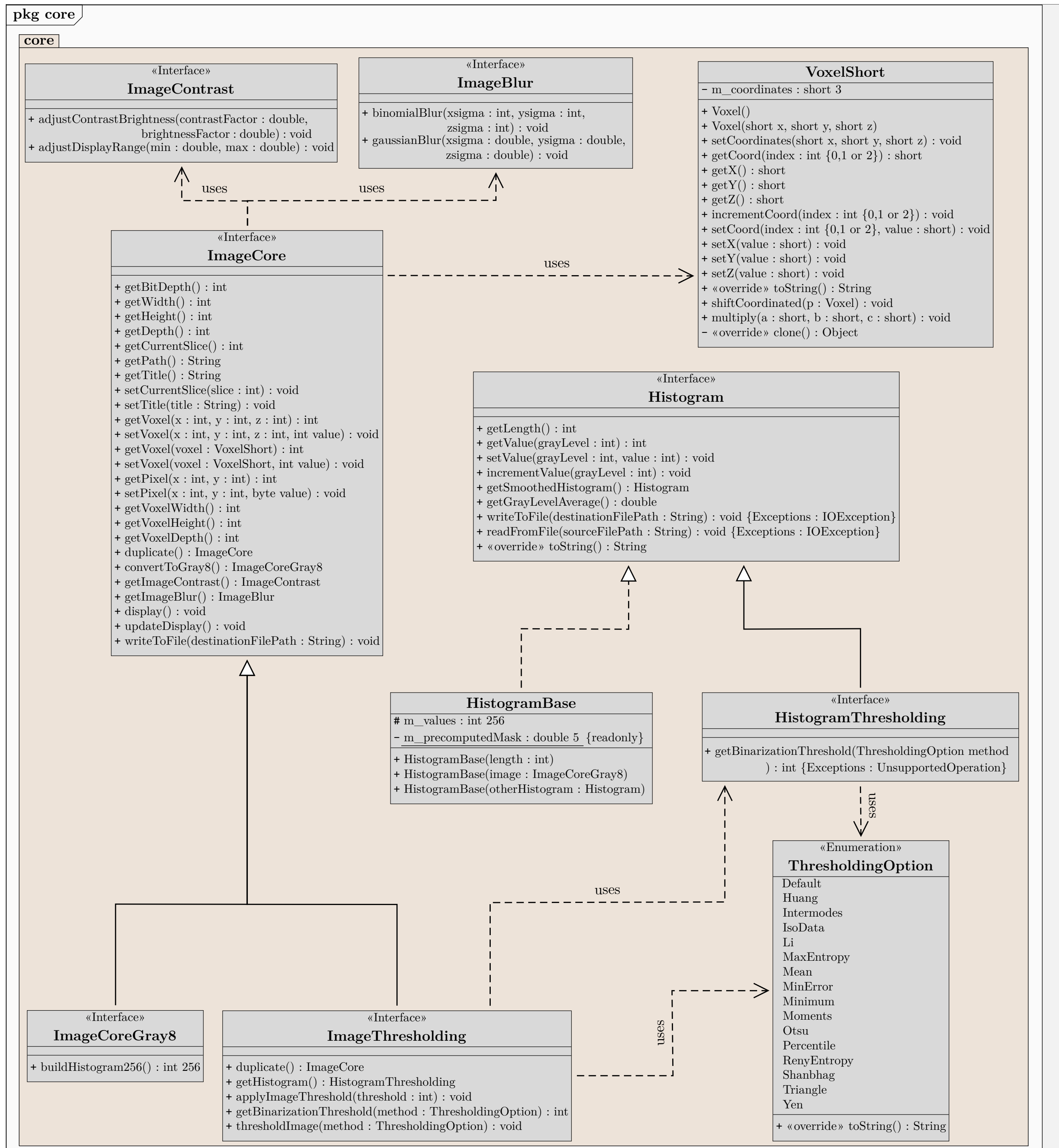
The core *API* is a set of interfaces and basic classes, the purpose of which is to expose the functionalities coming from the wrappers, each wrapper being implemented using a different imaging system. Diagram 1.1 represents the *UML* class diagram of the `core` package.

We present some examples of use and some documentation elements for the methods.

The complete documentation can be found in the *Javadoc*.

All of the examples, and possibly others, can be tested in the classes of the `test` package.

1.1 Class Diagram for the core package



Diag. 1. Class Diagram of the Package `core`

1.2 Loading and displaying an image

In the following example, we use the constructor for *ImageCoreIJ*, which is the basic wrapper for the `ij.ImagePlus` image class. The documentation and prototype for this constructors reads as follows:

```
/**
 * Constructs an instance by loading the image from a source file.
 * @param path path to the image source file on disk.
 * @param format_8bits true if the image should be converted () if necessary
     to 8 bits per pixels and memory should be optimized for this.
 * @throws IOException  IOException in case of failure to load the image from
     file
 */
public ImageCoreIJ(String path, boolean format_8bits) throws IOException;
```

The correct use, catching a possible exception in case the file could not be accessed (e.g. file not found), is as follows:

```
/**
 * Validates image loading and display
 * @param filename source image file
 */
static void testLoadImageAnDisplay(String filename){
  try{
    ImageCore image = new ImageCoreIJ(filename, true);
    image.display();
  } catch(IOException e){
    e.printStackTrace();
  }
}
```

1.3 Converting to 8 bits gray levels, saving an image

The following example loads an image, converts it to 8 bits per voxel, and saves the modified image to a file. Not all output formats are supported (see the *Javadoc* for `ImageCoreIJ.writeToFile`).

```
/**
 * Validates image 8 bits conversion
 * @param inputFilename
 * @param outputFilename
 */
static void testConvertTo8bits(String inputFilename, String outputFilename){
  try{
    ImageCore image = new ImageCoreIJ(inputFilename, true);
    image.convertToGray8();
    image.writeToFile(outputFilename);
  } catch(IOException e){
    e.printStackTrace();
  }
}
```

1.4 Blurring an image and Thresholding (e.g. Otsu)

Two kinds of image blur are currently supported :

- Gaussian Blur (*ImageJ* based implementation);
- Binomial Blur (home made implementation).

The following examples performs an Otsu thresholding method after blurring the image. The code is for the binomial blur. The code for the Gaussian blur is similar.

```java
/**
  * Validates image thresholding
  * @param filename
  * @param thresholdingMethod
  */
 static void testThresholding(String filename, ThresholdingOption thresholdingMethod){
   try{
     ImageThresholding image = new ImageThresholdingIJ(filename, true);
     image.getImageBlur().binomialBlur(8, 8, 2);
     image.thresholdImage(ThresholdingOption.Otsu);
     image.display();
   } catch(IOException e){
     e.printStackTrace();
   }
 }
```

1.5 Histogram manipulation

Here is a basic example to construct the histogram of an image. See Diagram 1.1 and the *Javadoc* for other operations available on histograms.

```java
 /**
  * Validates image histogram construction and saving to a text file
  * @param inputImageFile
  * @param outputFile output text file for the histogram.
  */
 static void testHistogram(String inputImageFile, String outputFile){
   try{
     ImageThresholding image = new ImageThresholdingIJ(inputImageFile, true);
     image.getHistogram().writeToFile(outputFile);
   } catch(IOException e){
     e.printStackTrace();
   }
 }
```

In this example, the histogram is saved to a text file.

The histogram's graphical representation can be obtained by *gnuplot* as *SVG* using the following *gnuplot* input file.

In this example, we display three histograms on the same graph respectively saved in files .

```
set terminal svg size 600,400 dynamic enhanced fname 'arial'  fsize 10
set output 'histogram.svg'
set key inside right top vertical Right noreverse noenhanced autotitle nobox
```

```
set style data points
set auto y
set xrange [0:255]

set title "Histograms"
plot 'histogram_1.dat' lc rgb "red", 'histogram_2.dat' lc rgb "green", 'histogram_3.dat' lc
```

See *gnuplot* documentation for more information.

1.6 Brightness/Contrast adjustment

The following code allows to adjust first the contrast, and then the brightness of the image, thus modifying the voxel's colors.

```
1  /**
2     * Validates image contrast and brightness changes
3     * @param inputImageFile
4     * @param outputFile output image file where to save the modified image after
          contrast operation.
5     * @param constrastFactor parameter (between 0 and 1) to enhance contrast
6     * @param brightnessFactor parameter (between 0 and 1) to enhance brightness
7     */
8     static void testContrastBrightnessWriteToFile(
9       String inputImageFile,
10      String outputFile,
11      double constrastFactor,
12      double brightnessFactor){
13    try{
14      ImageCore image = new ImageCoreIJ(inputImageFile, true);
15      image.getImageContrast().adjustContrastBrightness(constrastFactor,
          brightnessFactor);
16      image.writeToFile(outputFile);
17    } catch(IOException e){
18      e.printStackTrace();
19    }
20  }
```

The *Javadoc* for the `ImageCore.adjustContrastBrightness` method reads as follows:

```
1     /**
2     * Adjusts the contrast, brightness to change the display range of an image.
3     * The method maps linearly pixel values in the display range
4     * to display values in the range 0--255. Pixels with a value less than the
         minimum
5     * are set to black and those with a value greater
6     * than the maximum are set to white.
7     *
8     * Contrast change is applied first, and the the brightness shift.
9     *
10    * @param contrastFactor ranges between 0.0 and 1.0, Increases or decreases
         image contrast by varying the width of the display range. The narrower
         the display range, the higher the contrast
```

```
11     * @param brightnessFactor  ranges between 0.0 and 1.0, Increases or decreases
           image brightness by moving the display range
12     */
13    public void adjustContrastBrightness(double contrastFactor, double
         brightnessFactor);
```

It is also possible to set directly the display range through its new minimal and maximal gray level values. The *Javadoc* for the `ImageCore.adjustContrastBrightness` method reads as follows:

```
1    /**
2     * Sets the maximum value of the display range
3     * Pixels with a value greater than the maximum are set to white.
4     * @param min new minimum value of the display range.
5     * @param max new maximum value of the display range.
6     */
7    public void adjustDisplayRange(double min, double max);
```

1.7 Low Level Access to The Voxel's Values

There are two modes to access the voxel's color for *read/write* operations :

- Orderly Access to the values (slice by slice);
- Random Access to the values (voxel are accessed in any possible order)

Orderly **slice by slice access is significantly faster**, at least when using the *ImageJ* wrapper.

a) Orderly Access to the values (slice by slice).

The following example goes over the whole image, tests if the gray level is above 125, and sets the color to 255 accordingly.

```
1    /**
2     * Validates image orderly access to voxel's gray levels by going over an
          image
3     * This slice by slice access to the 8 bits per voxel image is faster than
          random access.
4     * @param inputImageFile
5     */
6      static void testOrderlyAccessVoxels(String inputImageFile){
7
8        try{
9          ImageCore image = new ImageCoreIJ(inputImageFile, true);
10         // Go over the image slice by slice
11         for (short z=1 ; z<=image.getDepth() ; z++){
12           // set the slice
13           image.setCurrentSlice(z);
14           for (short y=0 ; y<image.getHeight() ; y++){
15             for (short x=0 ; x<image.getWidth() ; x++){
16               // if the gray level is greater than 125
17               if (image.getPixel(x, y)){
```

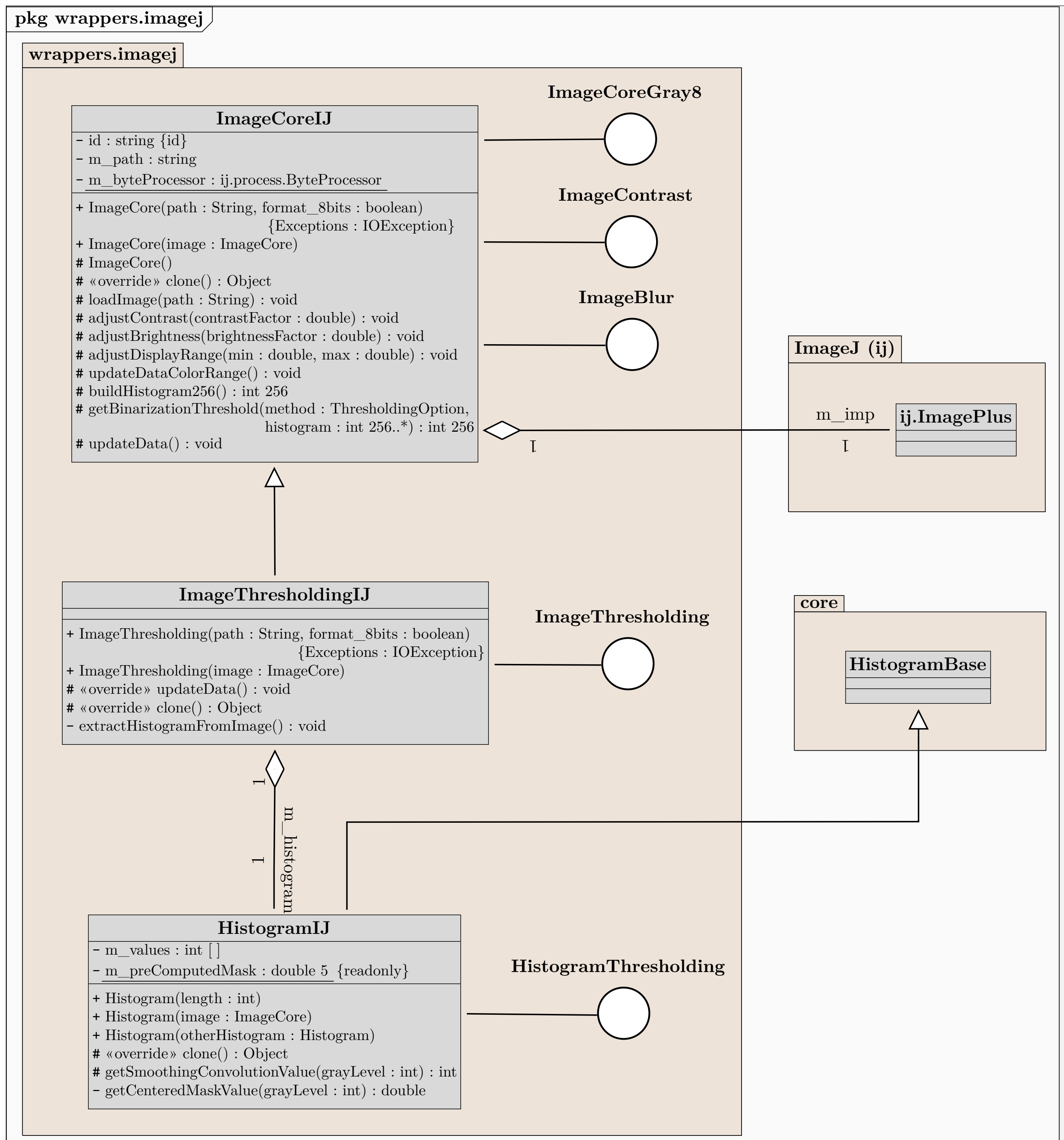
```
18                  // Set the pixel on the current slice
19                  image.setPixel(x, y, (byte)255);
20                }
21              }
22
23            }
24          }
25          // reinitialize the slice to update the last slice
26          image.setCurrentSlice(1);
27          image.display();
28        } catch(IOException e){
29          e.printStackTrace();
30        }
31      }
```

b) Random Access to the values.

The following example adds some random noise to many random voxel's gray levels.

```
1    /**
2     * Validates image random access to voxel's gray levels
3     * This random access is slower than the slice by slice access to the image
4     * @param inputImageFile
5     */
6     static void testRandomAccessVoxels(String inputImageFile){
7       Random random = new Random();
8       // pre-allocated voxel (to avoid saturating the garbage collector)
9       VoxelShort voxel = new VoxelShort();
10    try{
11      ImageThresholding image = new ImageThresholdingIJ(inputImageFile, true);
12
13      // we add random noise to voxels' gray-level for a number of random voxels
14      for (int i=0 ; i < 1.0e8 ; i++){
15        // Get a random voxel :
16        voxel.setCoordinates((short)random.nextInt(image.getWidth()),
17                    (short)random.nextInt(image.getHeight()),
18                    (short)random.nextInt(image.getDepth()));
19        // retrieve the original gray-level
20        int grayLevel = image.getVoxel(voxel);
21        // add noise
22        grayLevel += random.nextInt(50);
23        // clamp
24        short newGrayLevel = (short)((grayLevel < 0) ? 0 : (grayLevel > 255) ?
            255 : grayLevel);
25        // update the voxel's gray-level in the image
26        image.setVoxel(voxel, newGrayLevel);
27      }
28      image.display();
29    } catch(IOException e){
30      e.printStackTrace();
31    }
32  }
```

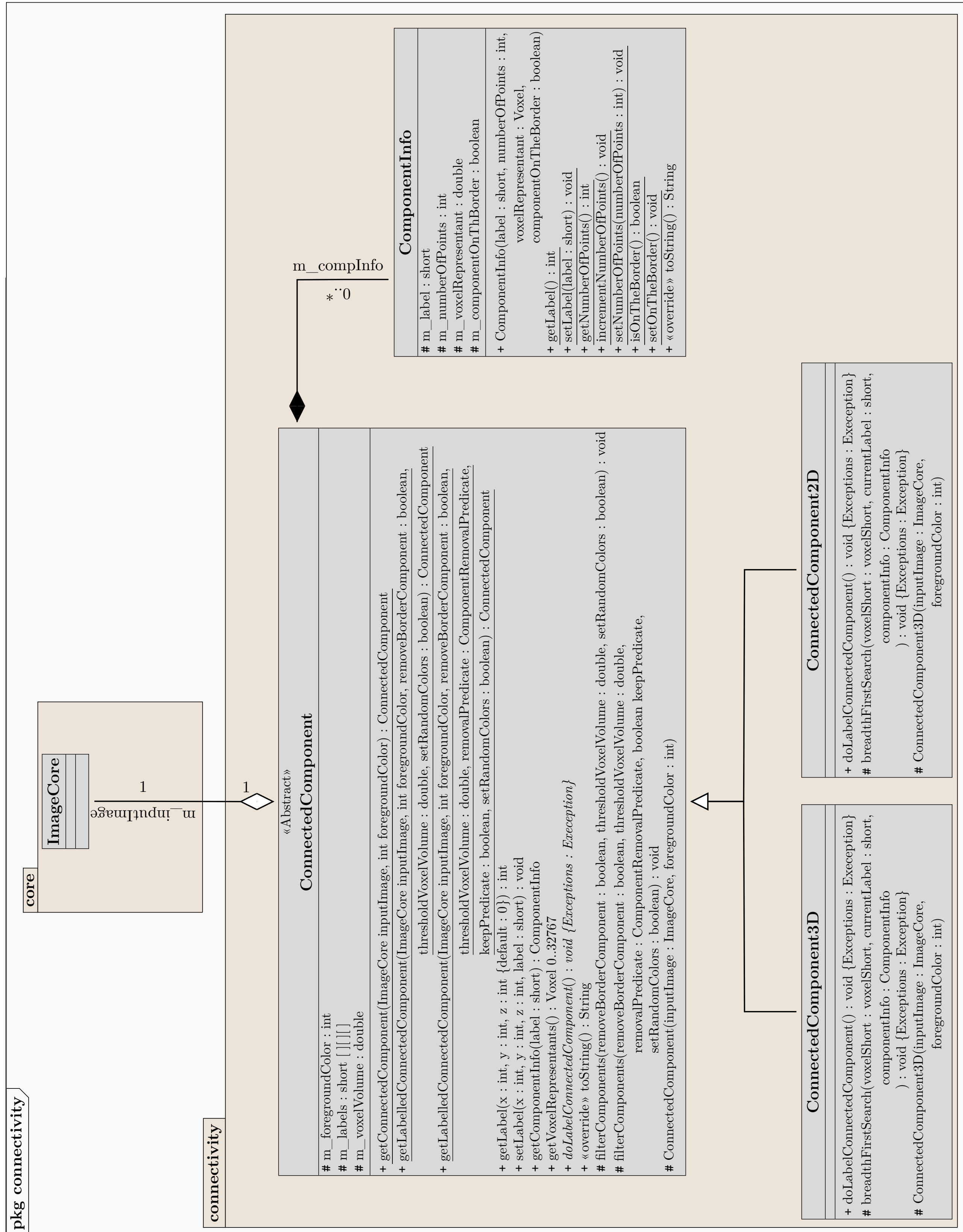
2 The Wrapper for *ImageJ*



Diag. 2. Class Diagram of the Package `wrappers.imagej`

3 The Package for Connected Components

3.1 Class Diagram for the `connectivity` package



Diag. 3. Class Diagram of the Package `connectivity`

3.2 Labeling Connected Components and basic Filtering

Here is a basic example of connected components labeling. The following code loads an image, binarizes it through a pass of blur followed by Otsu thresholding, and labels the connected components. Then it display the informations concerning the obtained components (data contained in the `connectivity.ComponentInfo` class)

The following operations are performed with the options used:

1. Connected components labeling (8−connectivity).
2. Removal of all connected components which are on the boundary of the image.
3. Removal of all connected components with a volume, as estimated by the calibration data (if any) and the number of voxels, below a specified volume threshold.
4. The code also attributes to each connected component some random gray level to be identified through display, as in Figure 3.2.

The *Javadoc* for the *Factory*-style method which performs the labeling, as well as several further operations, is as follows:

```
1  /**
2   * Constructs a ConnectedComponent derived class instance with relevant
       dimension (2D or 3D) and labels the components.
3   * Filters the image components according to two criteria:
4   * <ul>
5   *    <li>Possibly remove the components which are on the edge of the image</
       li>
6   *    <li>Possibly remove the components with size bellow some threshold</li>
```

```
7    * </ul>
8    * @param inputImage : input (probably binary) image, the components of which
         to compute.
9    * @param foregroundColor label of the 1's in the input image inputImage
10   * @param removeBorderComponent true if the components which are on the edge
         of the image should be removed by filtering
11   * @param thresholdVoxelVolume minimal volume for filtering (taking into
         account the calibration) for the components. 0 if no minimal volume is
         required
12   * @param setRandomColors true if the colors of the original image should be
         set according to the components labels.
13   * @return an instance of a concrete derived class for ConnectedComponent
14   * @throws Exception in case the number of connected components exceeds the
         Short.MAX_VALUE (32767)
15   */
16  public static ConnectedComponent getLabelledConnectedComponent(
17      ImageCore inputImage,
18      int foregroundColor,
19      boolean removeBorderComponent, double thresholdVoxelVolume,
20      boolean setRandomColors
21      ) throws Exception;
```

The use is as follows:

```
1   /**
2    * Test for labeling connected components of a binarized image.
3    * Only connected components with no voxel on the image's boundary
4    * are kept in the filtering process.
5    *
6    * Connected components with a volume below some threshold are
7    * also removed.
8    *
9    * a constant random gray level is set on each connected component.
10   *
11   * @param imageSourceFile the input image file on disk
12   */
13  public static void testComponentsLabeling(String imageSourceFile) {
14    try {
15      ImageThresholding image = new ImageThresholdingIJ(imageSourceFile, true);
16      System.out.println("Image " + image.getTitle() + " loaded.");
17
18      image.getImageBlur().binomialBlur(6, 6, 1);
19      //image.getImageBlur().gaussianBlur(8, 8, 1);
20      System.out.println("Blur done.");
21
22      // Image thresholding to get a binary image
23      image.thresholdImage(ThresholdingOption.Otsu);
24
25        ConnectedComponent cc;
26      try {
27        cc = ConnectedComponent.getLabelledConnectedComponent(
28            image, 255, true, 800.0d,
29            true);
30            // print connected components informations:
31            System.out.println(cc);
```

```
32          // Display the result:
33          image.display();
34      } catch (Exception e) {
35        System.err.println("Too many connected components");
36      }
37
38    } catch (IOException e) {
39      e.printStackTrace();
40    }
41  }
```

The output of `ComponentInfo` description is as follows (extract):

```
...
Component label : 20, Number of points : 13327
Component label : 21, Number of points : 13428
Component label : 22, Number of points : 1177
Component label : 23, Number of points : 62325
Component label : 24, Number of points : 14444
Component label : 25, Number of points : 1014
Component label : 26, Number of points : 13576
Component label : 27, Number of points : 13826
Component label : 28, Number of points : 12476
Component label : 29, Number of points : 94140
Component label : 30, Number of points : 94459
...
```

3.3 Advanced Connected Components Filtering

To be written.