

Object-oriented programming of filters with ImageJ

Workshop support material

Dimitar Prodanov^{a,b}

^aNeural Engineering Rehabilitation Laboratory, Catholic University of Louvain, Brussels, Belgium

^bDepartment of Pharmacology, Sofia Medical University, Sofia, Bulgaria

1. BASIC CLASSES

Programming of custom filtering tasks with *ImageJ* often requires development of certain customized applications. Within the framework of *ImageJ* these applications can be of 3 main types - subclasses of the classes `PlugIn`, `PlugInFilter` and `PlugInFrame`. Hereafter these custom commands will be referred to as *plugins*. The basic code that defines them is presented in Figs. 1 and 2. A more complete reference covering the basics of *ImageJ* and a tutorial can be found in Bailer (2003).¹ A good reference on programming in the Java language can be found in Eckel (2003)² and it is available also on-line at <http://www.mindview.net/Books/TIJ/>.

Figure 1. The basics of the custom commands

```
package ij.plugin;

public interface PlugIn {

    /** This method is called when the plugin is loaded.
     * 'arg', which may be blank, is the argument specified
     * for this plugin in IJ_Props.txt. */
    public void run(String arg);

}
```

1.1. The PlugIn interface

The `PlugIn` interface provides a framework that allows programming of extendable features (i.e. custom commands) of *ImageJ*. The `PlugIn` class is very simple, as can be seen from the listing in Fig. 1. By implementing this interface various other classes can be written. In general, these are fit to execution of simpler tasks where the processing does not require a particular type of image. The *arg* String variable passes the options that are required for the execution of the plugin. In the interface, the body of the `run` method is left empty. Therefore it should be rewritten in every custom plugin. Programming of a custom plugin usually requires handling of the `ImageProcessor` class. This is a general class that gives off several classes for handling of the different types of images. These are the `ByteProcessor`, `ColorProcessor`, `FloatProcessor`, and the `ShortProcessor` classes. Operations using these classes are usually constructed in the body of the `run` method.

Somehow more extended in functionality is the next class for custom commands.

1.2. The PlugInFilter interface

The `PlugInFilter` provides more extended capabilities for control and execution of commands. *ImageJ* plugins that filter or produce an image usually implement this interface. Upon initialization, the method `setup` is called once when the filter is loaded (Fig. 2). In the interface, the body of the method is left empty. The methods should be always overwritten by the user class that implements the `PlugInFilter` interface. The *arg* String variable passes

Mailing address: GREN, Catholic University of Louvain, Av. Hippocrates 54, PO Box UCL-5446, B-1200 Brussels, Belgium; e-mail: dimitrpp@gmail.com

Figure 2. PlugInFilter interfaces: commands that process images

```
package ij.plugin.filter;
import ij.*;
import ij.process.*;

/* ImageJ plugins that process an image should implement this interface. */
public interface PlugInFilter {

    /* arg, which may be blank, is the argument specified for this plugin in IJ_Props.txt.
    imp is the currently active image. */
    public int setup(String arg, ImagePlus imp);

    public void run(ImageProcessor ip);
    ...
    ...
}

```

the options that are required for the execution of the plugin. The *imp* ImagePlus variable passes the active image to be processed, i.e. the image on the screen that was in focus upon commencement of the plugin execution.

This method returns a flag that specifies the capabilities of the filter and the conditions of further plugin execution. There are different flags for 8-bit (grayscale and indexed color), 16-bit, 32-bit, and 24-bit (RGB) images. These flags are DOES_8G, DOES_8C, DOES_8G, DOES_16, DOES_32, and DOES_RGB; there is also the non-discriminative flag DOES_ALL that enables any type of image to be processed.

There are also flags signifying different capabilities, such as the ability to process stacks, to operate specifically in a (rectangular) region of interest (ROI) and to do masking in the bounding rectangle of the ROI. The *setup* method also defines the output state of the processed image, such as NO_CHANGES, NO_UNDO, NO_IMAGE_REQUIRED, ROI_REQUIRED, STACK_REQUIRED. There is also the special termination flag DONE to bypass the call to the *run* method.

After the initialization the method *run* is called (Fig. 2). This method actually processes the image. If the SUPPORTS_STACKS flag is set, *run* is called for each slice in a stack. *ImageJ* locks the image or the image stack before calling this method and unlocks it when the *run* is finished.

1.3. The PlugInFrame class: to handle frames

The PlugInFrame provides the functionality to handle closing and activating of windows and the associated user driven events. It is a subclass of the AWT.Frame class and implements the PlugIn interface. It can be used when the controls of the plugin are complex. These can be text areas, scrollbars, diagrams, plots etc. There are a lot of such examples on the *ImageJ* web-site. The PlugInFrame will not be presented in detail here. Further information regarding its use can be found in Bailer (2003).¹ In this class, the *run* method it is left empty as well and should be always overwritten by the user class that extends the PlugInFrame class.

2. A PRACTICAL EXAMPLE

In this section, the GranFilter plugin is used as an example for the custom plugin development. The source code of the plugin can be found at the *ImageJ* web-site (<http://rsb.info.nih.gov/ij/>) and in the conference CD-ROM. The plugin is used for the image processing and measurement tasks reported in previous works.^{3,4} Applications of the functionality provided by the plugin are described in my Workshop paper.⁵

2.1. Initialization

During the initialization, the values of the input variables should be specified and the correct type of image should be verified. An excerpt from the code is shown in Fig. 3. It is recommended also to check for the version of *ImageJ* and to require a certain minimal version. As can be seen from the source code, if the minimal operational conditions of the plugin are not satisfied its further execution is bypassed using the DONE flag.

Figure 3. The setup method: initialization

```
public int setup(String arg, ImagePlus imp){
    this.imp=imp;
    IJ.register(GranFilter_.class);
    if (arg.equals("about")){
        showAbout();
        return DONE;
    }
    if(IJ.versionLessThan("1.23") || !showDialog(imp)) {
        return DONE;
    }
    else {
        return DOES_8G+NO_CHANGES+NO_UNDO;
    }
} /* setup */
```

2.2. Input parameters

Figure 4. The showDialog: input of user parameters

```
boolean showDialog(ImagePlus imp)  {
    if (imp==null) return true;
    GenericDialog gd=new GenericDialog("Parameters");

    gd.addMessage("This plugin performs granulometric filtering\n");
    gd.addChoice("Type of structure element", strelitems, strelitems[eltype]);
    gd.addNumericField("Radius of structure element (pixels):", radius, 1);
    ...
    gd.addCheckbox("Euclidean distance:", thresh);
    gd.showDialog();
    ...
    if (gd.wasCanceled())
        return false;
    ...
    return true;
} /* showDialog */
```

A convenient way to get the values of user variables is by writing a method that prompts for their values using the `GenericDialog` capabilities provided by *ImageJ*. In this case this is the `ShowDialog` method (Fig. 4). The `ShowDialog` uses extensively the `GenericDialog` class, which is a customizable modal dialog box. Using this class different controls for user interaction can be added. These can be check boxes (`GenericDialog.addCheckbox`), text fields (`GenericDialog.addNumericField`), choice fields (`GenericDialog.addChoice`) etc. An excerpt of the source is provided in Fig. 4. The user input can be handled using the appropriate get-methods implemented in the `GenericDialog` class and the values of global variables can be updated. These methods can be `GenericDialog.getNextBoolean()`, `GenericDialog.getNextNumber()`, `GenericDialog.getNextText()`, `GenericDialog.getNextString()` etc.

2.3. Plugin execution

Once the plugin is initialized its execution is passed to the `run` method. It requires a variable of the `ImageProcessor` type. In the example, this variable is initialized using the active instance of `ImagePlus` variable in the `setup` method (Fig. 3).

Figure 5. The run method: how to make it tick

```
public void run(ImageProcessor ip) {
    ...
    ImageProcessor ip1 = ip.duplicate(); // makes a copy of image 1
    GrayOpen(ip1, eltype,minrad); // opens image 1

    ImageProcessor ip2 = ip.duplicate(); // makes a copy of image 2
    GrayOpen(ip2,eltype, maxrad); // opens image 2

    ip1.copyBits(ip2, 0, 0, Blitter.SUBTRACT); // subtracts 2 from 1
    GrayOpen(ip1, eltype,minrad); // clears artifacts
    result_image=new ImagePlus("Filtered_"+title+minrad+"_"+maxrad, ip1);
    ...
}
```

In Fig. 5 is shown the core of the image processing task implemented by the *GranFilter* plugin. In brief: the original image is duplicated 2 times in new *ImageProcessor* variables. Then morphological opening is performed for each one using call to the method *GrayOpen*. After this processing a new image is assembled from the subtraction of the second *ImageProcessor* from the first. Finally, another opening is performed to eliminate artifacts.

2.4. Basic help capabilities

It is useful to provide some elementary help for the operation of the plugin. A convenient way to do so is by using the construction shown in Fig. 6. The *showAbout* dialog mounts as an item in the Help/About Plugins sub menu in *ImageJ*.

Figure 6. The showAbout method: some credits

```
void showAbout() {
    IJ.showMessage("About GranFilter...",
        "This plug-in filter performs granulometric filtering of images\n"+
        "doi:10.1016/j.jneumeth.2005.07.011\n"+
        "more information at www.neuromorf.com"
    );
} /* showAbout */
```

3. IMPLEMENTING EXTENDED FUNCTIONALITY

Sometimes, the character of the image processing task requires more extended functionality to be implemented. For example, if one needs to develop several plugins that use similar classes of filters. In such case, it is convenient to encapsulate the core of the image processing into a custom package library that can be shared between different plugins.

I will give an example by the *MorphoProcessor* class that implements the core engine of Mathematical morphology (see Fig. 7). Mathematical apparatus was introduced by Matheron⁶ and Serra.⁷ The *MorphoProcessor* class is encapsulated in the *mmorpho* package. This class implements the functionality used in the *GrayMorphology*, *Granulometry*, *GranFilter* and *Covariogram* plugins available at the *ImageJ* web-site. For this implementation it is designed a library of constants that is shared between the classes of the package, e.g. *mmorpho.Constants*. *MorphoProcessor* uses 2 other classes from the same package - *StructureElement* and *LocalHistogram* (see Fig. 7). *StructureElement* can produce flat structure elements of various types, such as lines, squares, diamonds, circles etc. *LocalHistogram* is used for the sliding window algorithm implementation of the morphological operations.

Figure 7. The MorphoProcessor class

```
package mmorpho;
...
public class MorphoProcessor implements Constants {

    private StructureElement se, minus_se, plus_se, down_se, up_se;
    private LocalHistogram bh,p_h,m_h;
    private int[][]pg,pg_plus,pg_minus;
    int width, height;

    /** Creates a new instance of MorphoProcessor */
    public MorphoProcessor(StructureElement se) {
        ...
    }
    ...
}
```

Further, I will give an example by the implementation of the grayscale morphological *erosion* operation. The operation is usually denoted by \ominus or by $\epsilon(S, E)$ and is defined by:

$$S \ominus E = \min_{b \in E} \{S(x + b) - E(b)\}$$

where S denotes the image, E the Structure Element and $+$ the translations of the E from the point x by the vector b . The source code for its naïve implementation is given in Fig. 8. As can be seen from the listing, the code is quite simple and is actually directly derived from the definition of the operation.

Figure 8. The erode method

```
public void erode(ImageProcessor ip){
    ...
    byte[] pixels=(byte[])ip.getPixels();
    int[] wnd=new int[sz];
    byte[] newpix= new byte[pixels.length];

    for (int c=0;c<pixels.length;c++) {
        wnd=getMinMax(c, width, height, pixels, pg, ERODE);
        min=wnd[0]+255;
        newpix[c]=(byte)(min&0xFF);
    }
    System.arraycopy(newpix, 0, pixels, 0, pixels.length);
}
```

ACKNOWLEDGMENTS

I would like to acknowledge W.S. Rasband from the National Institutes of Health, Bethesda, Maryland, USA for his continuous development and support of *ImageJ*. The author is currently a J. G. Nicholls Fellow of the International Brain Research Organization.

REFERENCES

1. W. Bailer, *Writing ImageJ Plugins – A Tutorial*. Institute of Information Systems and Information Management at Joanneum Research GmbH., Graz, Austria, 1.6 ed., 2003.
2. B. Eckel, *Thinking in Java*, Prentice Hall, Upper Saddle River, NJ, 3rd ed., 2002.
3. D. Prodanov, *Morphometric analysis of the rat lower limb nerves. Anatomical data for neural prosthesis design*. PhD thesis, Twente University, Enschede, The Netherlands, 26 Jan 2006.

4. D. Prodanov, J. Heeroma, and E. Marani., “Automatic morphometry of synaptic boutons of cultured cells using granulometric analysis of digital images.,” *J Neurosci Methods* **151**, pp. 168–177, Mar 2006.
5. D. Prodanov, “Automatic measurements of cell structures: Programing and applications,” *ImageJ User and Developer Conference*, (Luxembourg), May 2006.
6. G. Matheron, *Random Sets and Integral Geometry*, Wiley series in probability and mathematical statistics, Wiley, New York, 2nd ed., 1975.
7. J. Serra, *Image Analysis and Mathematical Morphology*, Academic Press Ltd, London, 1st ed., 1982.