s c c h
software competence center
hagenberg

FUZZY.LOGIC.LAB.LINZ

Advances in Integrated Software Sciences

Proceedings of the Master and PhD Seminar Summer term 2010, part 1

Softwarepark Hagenberg
SCCH, Room 0/2
7 April 2010

Software Competence Center Hagenberg
Softwarepark 21
A-4232 Hagenberg
Tel. +43 7236 3343 800
Fax +43 7236 3343 888
www.scch.at

Fuzzy Logic Laboratorium Linz
Softwarepark 21
A-4232 Hagenberg
Tel. +43 7236 3343 431
Fax +43 7236 3343 434
www.flll.jku.at

Program

Session 1: Wednesday, 7 April, Chair: Roland Richter

13:30 Ulrich Brandstätter, Oliver Buchtala:
Plugin development with the flllip image processing library

14:00 Thomas Trenker, Tomas Kazmar:
KNIME Extensibility

Technical Report
FLLL–TR–1002

FUZZY.LOGIC.LAB.LINZ

Plugin development with the flllip image processing library

Ulrich Brandstätter, Oliver Buchtala, and Roland Richter
Fuzzy Logic Laboratorium Linz
email: info@flll.jku.at

Abstract — flllip is a framework for fast image processing algorithm development. It is not thought as a replacement for well-established image processing frameworks, like OpenCV or Intel® IPP, but as a superstructure for existing libraries, with the goal to (re-)use exisiting image processing functionality in an uniform way. Therefore, it may reduce development and testing times.
flllip is specialized in image processing using C++, although it can be used in any environment which provides C bindings, such as MATLAB®. In order to place existing functionality at the disposal for different environments, the flllip framework uses dynamic link libraries. This also facilitates a separation between image processing and application logic, and in consequence alleviates co-operation amongst multiple parties. Using dynamic link libraries, flllip also enables an image processing algorithm written in C++ to execute any other algorithm encapsulated in a flllip plugin.

Key words — *image processing, C++, plugin, framework*

Institut für Wissensbasierte Mathematische Systeme
Johannes Kepler Universität Linz
Altenberger Str. 69
A-4040 Linz
Austria

Fuzzy Logic Laboratorium Linz
softwarepark hagenberg
Softwarepark 21
A-4232 Hagenberg
Austria

1 Introduction

flllip is a library to enable fast plugin development mainly for image processing in C/C++. The framework itself, as well as several of its critical design decisions, were first presented in [BBKR10]. In this report, we will provide a detailed description of the inner anatomy of a flllip plugin.

2 Plugin developer tutorial

The plugin developer tutorial explains the flllip framework with focus on C++ plugin developers. It aims at the presentation of framework specifics by exercising typical, yet rather simple examples. At the end of the developer tutorial, a C++ programmer should be able to use the features of flllip regarding plugin creation to the full.

As reference for subsequent code examples, the flllip plugin interface can be seen in Listing 1.

2.1 Basics

This section introduces the flllip plugin template, which will be used as starting point for following tutorial sections. In short, it implements the plugin interface (Listing 1), which therefore can be explained in more detail, yet from an algorithmic point of view, it does not do anything. The whole source code can be reviewed in Listing 2 and Listing 3.

2.1.1 Template header file

The header file implementation of the template image plugin is straightforward:

- Inclusion of ImagePlugin.hpp, the base class header file.

```
/* Template.hpp */
 4 #include <ImagePlugin.hpp>
```

- Usage of namespace flllip (because its comfortable - flllip is the namespace of the presented framework) and sub-namespace flllipin (our choice for our plugins).

```
/* Template.hpp */
 6 namespace flllip {
 7 namespace flllipin {
```

- Class declaration of Template:

```
/* Template.hpp */
 9 class Template: public ImagePlugin {
10   typedef ImagePlugin Base;
```

 Public inheritance from ImagePlugin (Listing 1)

10 Declaration of a Base typedef for the inherited ImagePlugin class. This is an optional, but convenient technique, to increase readability.

- Virtual constructor (Section 2.8.1) - mandatory (yet somewhat inconvenient) method which allows plugin instantiation with only a base-class pointer (otherwise not possible in C++). A virtual constructor must be implemented in a similar way for every flllip plugin.

```
/* Template.hpp */
17 virtual const Template* Create() const { return new Template(GetHost()); };
```

- Method to query the plugin version as integer value - the part < 100 indicates the minor revision, whereas the rest stands for the major revision. A value of 104 means major revision 1, minor revision 4. *Note: one should not prepend zeros, otherwise the number is interpreted as octal number.*

```
/* Template.hpp */
19 virtual const IP_UNSIGNED GetPluginVersion() const { return /*00*/1; }
```

- Methods for various meta information, like author name or description. The implementation of method GetDescription is mandatory, whereas GetAuthor or GetCopyRight are optional.

```
/* Template.hpp */
20 virtual const std::string GetAuthor() const { return "JKU Linz"; }
21 virtual const std::string GetCopyright() const { return std::string(); }
22
23 virtual const std::string GetDescription() const {
24  return "flllip template that does nothing.";
25 }
```

- The GetName method is somewhat important - it expects a human-readable and **unique** URL-style identifier for the plugin.

```
/* Template.hpp */
27 virtual const std::string GetName() const { return "Template/Template"; }
```

- The GetParameterSignature method can semantically be ignored for now, parameters are discussed in detail in later tutorial chapters (Section 2.3). Regarding its implementation, it returns a copy of a private parameter signature member.

```
/* Template.hpp */
29 virtual Arguments GetParameterSignature() const {
30  return GetHost()->CopyArguments(m_parameters);
31 }
```

- The Initialize method also can be ignored for now, it will become relevant in later tutorial chapters (Section 2.3, Section 2.5.2).

```
/* Template.hpp */
33 virtual void Initialize(const Arguments& parameters);
```

- GetInputSignature and GetOutputSignature return, as GetParameterSignature, a copy of private signature members. These members are supposed to contain information about what the plugin expects as input and what kind of output it will produce - in case of this template code, neither input nor output data is expected.

```
/* Template.hpp */
35 virtual Arguments GetInputSignature() const {
36  return GetHost()->CopyArguments(m_input_args);
37 }
38
39 virtual Arguments GetOutputSignature() const {
40  return GetHost()->CopyArguments(m_output_args);
41 }
```

- The Execute method is the working method - here it actually does nothing, since this this first part of the tutorial is about the template code.

```
/* Template.hpp */
42 virtual void Execute (Arguments& input_args, Arguments& output_args);
```

- For the sake of completeness, the serialization and de-serialization function pair is declared - it can be ignored for now, as it will be explained in later sections (Section 2.8.2) in detail.

```
/* Template.hpp */
44 virtual const IP_BOOL Serialize (std::string& serialization_data) const;
45 virtual const IP_BOOL Deserialize (const std::string& serialization_data);
```

- Finally, we declare member variables for plugin arguments and parameters. Using member variables and the signature-methods listed above, a plugin-developer just has to fill the members according to the needs of the plugin at the proper spots. This is otherwise an implementation detail of our template code, the same could also be archieved in other ways. Subsequent tutorial sections stick to this scheme.

```
/* Template.hpp */
49 Arguments m_parameters, m_input_args, m_output_args;
```

2.1.2 Template implementation file

In the following, the implementation of the template plugin, which does not do anything, but otherwise will compile and execute as intended, is discussed:

- Inclusion of ArgumentChecks.hpp (shown in Listing 6), which allows a plugin developer to validate a set of arguments by only a single line of code.

```
/* Template.cpp */
 1 #include <ArgumentChecks.hpp>
```

- Inclusion of the class header file previously discussed (Listing 2).

```
/* Template.cpp */
3 #include "Template.hpp"
```

- Usage of our namespaces.

```
/* Template.cpp */
5 namespace flllip {
6 namespace flllipin {
```

- Implementation of class constructor:

```
/* Template.cpp */
8 Template::Template(const ImagePluginHost* host): Base(host) {}
```

We pass the given host argument to base class constructor using the initialization list - this is mandatory and has several implications; for now, we content ourselves with the knowledge that the host will allocate memory for arguments.

- The class destructor does not need to do anything in this example.

```
/* Template.cpp */
10 /*virtual*/ Template::~Template() {}
```

- Also, the Initialize method can be kept empty - it will be used in later examples.

```
/* Template.cpp */
12 /*virtual*/ void Template::Initialize (
13  const Arguments& parameters) {
14
15   CheckParameters (parameters, m_parameters);
16 } // /*virtual*/ void Template::Initialize (...)
```

- Argument validation of the Execute method: both CheckInputArguments and CheckOutputArguments are used to check if the given match the expected. In case of CheckInputArguments, the arguments are validated for correct type, for fitting meta information and for data availability - the last one is not true for CheckOutputArguments. If the expectations are not fulfilled, the methods throw proper exceptions, so subsequent code can rely on valid data. These methods are discussed in Section 2.6 in more detail. Here, of course, argument validation is not too reasonable, since the template does neither expect nor process any argument.

```
/* Template.cpp */
21 CheckInputArguments (input_args, GetInputSignature());
22 CheckOutputArguments (output_args, GetOutputSignature());
```

- The implementations of the Serialize and Deserialize methods, which will be covered at the very end of the tutorial (Section 2.5), in this case re-implement the default behaviour: they do nothing but return false to indicate that the plugin is not stateful.

```
/* Template.cpp */
27 /*virtual*/ const IP_BOOL Template::Serialize (
28  std::string& serialization_data) const {
```

```
29
30   return IP_FALSE;
31 } // /* virtual */ const IP_BOOL Template::Serialize (...) const
32
33 /* virtual */ const IP_BOOL Template::Deserialize (
34  const std::string& serialization_data) {
35
36   return IP_FALSE;
37 } // /* virtual */ const IP_BOOL Template::Deserialize (...)
```

2.2 Transformation of image to image

This section covers the implementation of the simplest image plugin type, an image processing filter which converts a given input image to an output image. Step by step, the integration of a simple but common filter, which transforms a RGB image to a grayscale image, into the fIIIip framework will be shown. This plugin, as well as all subsequent code examples in this tutorial, extend the template code presented in Section 2.1.
Because this is the first actual plugin tutorial, the changes to the header file of the template will be explained thoroughly:

Class declaration : the plugin of this tutorial section is called ConvertToGrayscale - the class has to be named accordingly:

```
/* ConvertToGrayscale.hpp */
 9 class ConvertToGrayscale: public ImagePlugin {
```

Plugin description: this tutorial plugin converts a given RGB image to a grayscale image - the plugin must yield this information for a human user:

```
/* ConvertToGrayscale.hpp */
23 virtual const std::string GetDescription() const {
24  return "Converts arbitrary image to grayscale image using FreeImage function.";
25 }
```

Plugin name: every fIIIip plugin must have a unique and human readable URL-style name. This tutorial plugin is identified as ConvertToGrayscale under section Color:

```
/* ConvertToGrayscale.hpp */
27 virtual const std::string GetName() const { return "Color/ConvertToGrayscale"; }
```

Now to the far more interesting implementation details:

- Implementation of the class constructor:

```
/* ConvertToGrayscale.cpp */
23 ConvertToGrayscale::ConvertToGrayscale(
24  const ImagePluginHost* host): Base(host) {
25
26   Argument inimage(
```

```
27     GetHost()->CreateImageArgument(
28       IP_Input,
29       IP_FALSE,
30       "Input image",
31       "Arbitrary single-page input image with 8, 24 or 32 bit depth.",
32       1));
33     m_input_args.push_back (inimage);
34     Argument outimage(
35       GetHost()->CreateImageArgument(
36       IP_Output,
37       IP_FALSE,
38       "Output image",
39       "Grayscale output image.",
40       1));
41     m_output_args.push_back (outimage);
42   }
```

23 - 24 In the constructor, we pass the given host argument to the base class constructor using the initialization list - this is mandatory and has several implications; for now, we content ourselves with the knowledge that the host will allocate memory for our images.

26 - 33 In this section, the input argument, which will receive the RGB image to convert, is specified.

27 For the allocation of the argument the host is used. *Note: in general, every argument which leaves the plugin boundaries must be created using the host, otherwise memory leaks can occur in the best case, unpredictable crashes in the worst case.*

28 The argument we want to create is an input argument.

29 The host must not allocate any actual argument data, because this argument is only used as signature. Setting this parameter to true, the host could f.i. allocate an image of specified size and depth using the same function.

30 "'Input image"' is the short description of the argument. To supply an argument description is optional, but nice.

31 The detailed description of the argument. Again, this is optional, but nice.

32 We only allow single-page images.

33 The newly created image argument is stored in the proper member variable and can, from now on, be retrieved using the GetInputSignature method.

34 - 41 In this section, the output argument, which will receive the grayscale image, is specified. In difference to the already created argument, this one is specified as output argument.

- Implementation of the Execute method:

```
/* ConvertToGrayscale.cpp */
50 /*virtual*/ void ConvertToGrayscale::Execute (
51   Arguments& input_args, Arguments& output_args) {
52
```

```
53   CheckInputArguments (input_args, GetInputSignature());
54   CheckOutputArguments (output_args, GetOutputSignature());
55   int channels = input_args[0].get()->meta.freeimage.channels;
56   int bpp = input_args[0].get()->meta.freeimage.bpp;
57   int width  = input_args[0].get()->meta.freeimage.width;
58   int height = input_args[0].get()->meta.freeimage.height;
59
60   FIBITMAP* indib(input_args[0].get()->data.freeimage.data.fi_bitmap);
61   FIBITMAP* outdib(0);
62   if ((outdib = FreeImage_ConvertToGreyscale (indib)) == 0) {
63    throw Exception::ExecutionException(
64     "Failed to apply grayscale conversion.");
65   }
66   GetHost()->GetCHost()->FreeArgumentData (
67    GetHost()->GetCHost(), output_args[0].get());
68   output_args[0].get()->data.freeimage.data.fi_bitmap = outdib;
69   output_args[0].get()->meta.freeimage.bpp = FreeImage_GetBPP (outdib);
70   output_args[0].get()->meta.freeimage.channels = 1;
71   output_args[0].get()->meta.freeimage.width = width;
72   output_args[0].get()->meta.freeimage.height = height;
73   output_args[0].get()->meta.freeimage.fif =
74    input_args[0].get()->meta.freeimage.fif;
75   output_args[0].get()->meta.freeimage.pages = 1;
76 }
```

53 - 54 Argument validation of the Execute method: both CheckInputArguments and CheckOutputArguments are used to check if the given match the expected. In case of CheckInputArguments, the arguments are validated for correct type, for fitting meta information and for data availability - the last one is not true for CheckOutputArguments. If the expectations are not fulfilled, the methods throw proper exceptions, so subsequent code can rely on valid data. These methods are discussed in Section 2.6 in more detail.

55 - 58 We access the given input image, which at this point must be valid, and read out meta information regarding the image. The syntax takes a little getting used to, it results from arguments being so called C POD structures, which are explained in subsequent tutorial chapters.

60 We access the actual input image, which is stores in a FreeImage container. Because of the previous input argument check, this is ensured to be a valid image.

62 The actual algorithm is, for us, a real no-brainer, since we use the implementation from the FreeImage library.

63 - 64 In case the FreeImage function failed for any reason, an exception is thrown to tell the outside world.

66 - 67 Because the output argument we received to store the grayscale image to may already contain a valid image, we free any used data by letting the host clean up, otherwise a memory leak may result.

68 - 75 Finally, the output argument meta data are set accordingly, as is the argument data pointer.

2.3 Parametrized plugins

Many image processing algorithms involve procedure parameters and therefore are more complex than simple filters regarding I/O behaviour. In this section an extension of the RGB to graylevel image plugin will be presented, which allows user-defined weightings for each colour channel. Plugin parameters make a fixed workflow necessary (Section 2.5) and introduce plugin states (Section 2.8.1).
Again, only section relevant code will be shown from now on instead of full listings. The most important change to the RGB to grayscale converter from the first tutorial section is the implementation of the Initialize method:

```
/* ConvertToGrayscaleWeighted.cpp */
58 /*virtual*/ void ConvertToGrayscaleWeighted::Initialize
     (const Arguments& parameters) {
59
60   CheckParameters (parameters, m_parameters);
61   m_red = parameters[0].get()->data.value.value.double_val;
62   m_green = parameters[1].get()->data.value.value.double_val;
63   m_blue = parameters[2].get()->data.value.value.double_val;
64   if (sum > 0.0) {
65     m_red /= sum;
66     m_green /= sum;
67     m_blue /= sum;
68   }
69 }
```

60 Like CheckInputArguments and CheckOutputArguments from Section 2.1, CheckParameters can be used to validate the parameter set given to the Initialize method. In case the given set deviates from the expectations (which are determined in the constructor by filling the m_parameters member), proper exceptions are thrown.

61 - 63 When the validity of the given parameters is ensured, the actual parameter values are extracted. The parameter values are stored into specific member variables m_red, m_green and m_blue of type **double**.

Under the hood: arguments and parameters

In the following, different properties of arguments and parameters are discussed:

- **Paramters are arguments:** arguments and parameters are of the same datatype, discrimination happens solely on a semantic level.
- **Available argument types:** Figure 1 shows the available argument types, which are focused on image processing tasks:

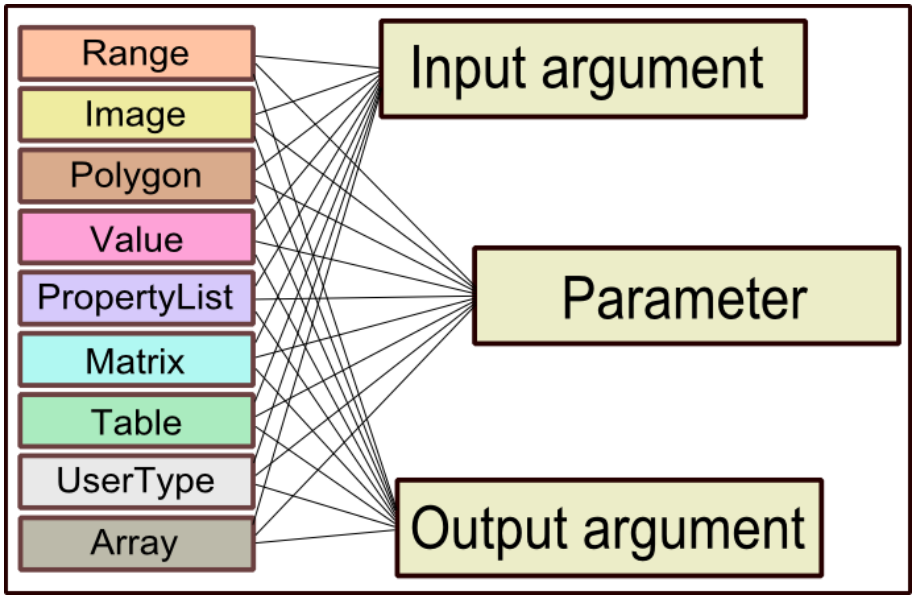

Figure 1. ARGUMENT TYPES

- **Image**, based on the FreeImage library, for both single- and multi-page images.
- **Value**, a simple type which holds either a boolean, an integral, a floating point or a string value.
- **Range**, a type which either can be used to limit the range of a numerical value or as interval type.
- **Polygon**, a field of x/y coordinate pairs.
- **Array**, a field of homogeneous values, useful for instance for histogram representation.
- **Matrix**, a two-dimensional array.
- **Table**, a not-yet available data type for a set of inhomogeneous data types, specialized for machine learning issues.
- **Property list**, a field of pairs of property names and inhomgeneous values.
- **User**, a string-based type with an id, which is available to facilitate user type costumization.

- **Arguments are C structs:**[1] they have a type id, meta information apart from actual argument data, and unions, which have to be accessed dependent on the type.
- **(De-)serialization:** all argument types can be serialized and deserialized innately by supplied function pointers to proper functions - see the declarations of **Serialize** and **Deserialize** in Listing 4.
- **Input and output arguments:** an argument can be flagged as input or output argument, or as both - parameters ignore these flags.

[1] C POD types for interoperability

```
Argument foo = GetArgumentFromSomewhere (...);
foo->flags = IP_Input | IP_Output;
```

- **Argument creation:** the host facilitates various argument creation methods - this way the complex argument structures can be allocated and initialized in a single function call (see Section 2.5.1 for more detail). Also argument deletion is supported by the host.
- **Automatic resource de-allocation:** from C++ view, all arguments created by the host are given wrapped in a reference counting shared pointer with custom deletion function so that no argument-related memory leaks can occur.

```
/// Argument uses a boost share_ptr for automated resource deallocation.
typedef boost::shared_ptr<ImagePluginArgument> Argument;
/// Arguments = Argument-vector
typedef std::vector<Argument> Arguments;
```

2.4 Multiple arguments

Image processing algorithm may utilize multiple input and produce several output arguments, once again increasing the I/O complexity of the previous tutorial sections: two input images may be mixed together, an RGB image may be split up into three graylevel images, etc.. This time the relevant sections of a crop image plugin, which copies a rectengular area of an input image to the output image, are discussed:

- **Constructor**: in difference to the previous examples, two input arguments instead of a single one are specified during plugin construction:

```
/* Crop.cpp */
13 Crop::Crop(const ImagePluginHost* host): Base(host) {
14   Argument inimage(
15     GetHost()->CreateImageArgument(
16     IP_Input,
17     IP_FALSE,
18     "Input image",
19     "Arbitrary single-page input image with 8, 24 or 32 bit depth.",
20     1));
21 m_input_args.push_back (inimage);
22
23 Argument rectsize(
24   GetHost()->CreatePolygonIntArgument(
25     IP_Input,
26     "Crop rectangle",
27     "Rectangle to specify crop size",
28     2));
29 m_input_args.push_back (rectsize);
```

23 - 28 The second input argument has a new type: polygon. filllip polygons are collections of two dimensional vectors (supported argument types are discussed in 2.3). As a rectangle can be represented with two vectors, the polygon size is fixed to two (last parameter).

29 We add also the second input argument signature to the proper member.

- **Execute** method: after the argument validation in the Execute method, the second input argument of type polygon can be accessed by using the proper argument index, as shown in the following:

```
/* Crop.cpp */
53 IP_INT left(input_args[1].get()->data.polygon.values[0].int_val.x);
```

2.5 Plugin construction, initialization and execution

This section covers the plugin lifecycle. First, the role of arguments regarding plugin initialization and execution is discussed, followed by a treatise of so called dynamic signatures.
The lifecycle of a plugin typically consists of three stages: construction, initialization and execution. During construction, a plugin must specify its parameter signature, whereas during initialization it has to establish its input and output signature. Of course, a plugin without parameters may build its I/O signatures already during construction. Additional information regarding plugin states can be found in Section 2.8.1.

2.5.1 Argument signatures vs. arguments with data

Signatures are arguments which typically do not contain actual data, only meta information. Plugins yield their expected signatures upon query, the relevant functions are listed subsequently:

```
virtual Arguments ImagePlugin::GetParameterSignature() const;
virtual Arguments ImagePlugin::GetInputSignature() const;
virtual Arguments ImagePlugin::GetOutputSignature() const;
```

Regarding the differentiation of argument signatures and arguments with data the following remarks may straighten things up:

- Meta information give the caller opportunity to prepare the arguments according to the plugin expectations. For instance, certain image plugins may only work with some kinds of images: morphological image plugins typically work upon binary images, whereas other algorithms may only work on RGB images with 8-bit per channel representation.
- An image filter on the other hand expects an actual image as input, not just its signature. Therefore, the caller must take care to actually *fill the signatures* with proper data. Of course the input data may actually be the output of another plugin[2].
- Many specialized functions exist to easy the creation of arguments and parameters. These functions also can create empty data sets according to the specified meta information.

[2] for instance an image loader plugin which receives a path and automatically loads and decodes the image file

2.5.2 Dynamic signatures

In order to implement an image processing algorithm for lots of different use-cases, it is sometimes desirable to change its signature according to the actual usage. For instance, the application a filter to the given input image renders the creation and allocation of an output image unnecessary, and therefore decreases the memory usage and execution time. On the other hand, an in-place operation may strip the user of the opportunity to visually compare the input with the output image.
Dynamic argument signatures are dependent on the plugin parameters. For an image processing algorithm, they are given to the plugin using the Initialize method: if dynamic signatures are desired, they have to be determined during the initialization dependent on the given parameters.
At this point another example is presented: a median plugin for images, which either filters the given input image itself or applies the filter to a copy of the input image:

- **Initialize** method: the necessary adaption of the plugin I/O signature is done in the initialize method, as shown in the following.

```
/* OrderFilterBase.cpp */
49 /*virtual*/ void OrderFilterBase::Initialize (const Arguments& parameters) {
50
51  CheckParameters (parameters, m_parameters);
52  // Depending on parameter p_copyimage, the plug-in has a separate output image.
53  m_copy_image = parameters[0].get()->data.value.value.bool_val;
54  // extract other parameter values
55  m_mask_width = parameters[1].get()->data.value.value.int_val;
56  m_mask_height = parameters[2].get()->data.value.value.int_val;;
57
58  m_input_args.clear();
59  m_output_args.clear();
60
61  Argument inimage(
62   GetHost()->CreateImageArgument(
63    (m_copy_image  == IP_TRUE ? IP_Input : IP_Input | IP_Output),
64    IP_FALSE,
65    (m_copy_image == IP_TRUE ? "Input image" : "Input/output image"),
66    "Arbitrary input image with 8, 24 or 32 bit depth.",
67    1));
68  m_input_args.push_back (inimage);
69
70  if (m_copy_image == IP_TRUE) {
71   Argument outimage(
72    GetHost()->CreateImageArgument(
73     IP_Output,
74     IP_FALSE,
75     "Output image",
76     "Filtered output image.",
77     1));
78   m_output_args.push_back (outimage);
79  }
80 } // /*virtual*/ void OrderFilterBase::Initialize (...)
```

53 First, the parameter responsible for the inline or copy setting is queried and stored into a boolean member variable.

63 Depending on the responsible parameter, the expected input image is either flagged for input only or for input and output.

65 Also, the argument description is adapted.

70 - 79 If the user did not specify an inline operation, an output image signature is also created.

- **Execute** method: of course, in the Execute method, the plugin code must also react on the setting, as shown in the following code snippet:

```
/* OrderFilterBase.cpp */
146 if (m_copy_image == IP_TRUE) {
147   output_args[0] = crop_output[0];
148 } else {
149   input_args[0] = crop_output[0];
150 }
```

2.6 Exceptions

A plugin must also cope with situations where the caller gives parameters and arguments which differ from the signatures specified by the plugin. In our framework, no static type checks can be used because all arguments are dynamically created, therefore runtime type checks must be used. In most cases, a plugin will check if the given arguments or parameters are in accordance to its signature, which can be done quite easily by the usage of provided functions, which are shown in Listing 6.

Supported exceptions

These argument checking functions are implemented to throw exceptions when the given differs from the expected. The set of exceptions (shown in Listing 7) which can possibly be thrown, reflects all types of possible runtime errors. Not all specified exceptions are dedicated to argument checks however: if the execution of the plugin failed for other reasons (out of memory conditions, etc.), proper exceptions must be thrown to give the caller opportunity to react properly.
There are no restrictions to the use of exceptions in plugins - well-known C++ rules apply: exceptions may be thrown from any method and from the constructor. An exception must not leave the scope of the destructor because of *stack unwinding* issues[3]. In the following, the crop plugin from the previous tutorial will be presented with focus on the spots exceptions may be thrown:

- **Constructor**: the constructor of the crop plugin example takes care of the preparation of the input and output signatures:

```
/* Crop.cpp */
13 Crop::Crop(const ImagePluginHost* host): Base(host) {
14   Argument inimage(
15     GetHost()->CreateImageArgument(
```

[3] http://www.parashift.com/c++-faq-lite/exceptions.html#faq-17.3

```
16    IP_Input,
17    IP_FALSE,
18    "Input image",
19    "Arbitrary single-page input image with 8, 24 or 32 bit depth.",
20    1));
21  m_input_args.push_back (inimage);
```

14 - 20 In case the signature cannot be created due to insufficient memory, a proper exception is thrown. Also, if invalid flags are given (IP_Input, IP_Output or both), an exception holds. *Note: if the method is used to allocate actual data, in case of insufficient memory atm. no exception is thrown, because the object itself is still valid. Of course, it than differs from the users expectations.*

- **Execute** method: in the crop plugin Execute method, there are several spots where exceptions may be thrown:

```
/* Crop.cpp */
43  /*virtual*/ void Crop::Execute (Arguments& input_args, Arguments& output_args) {
44
45   CheckInputArguments (input_args, GetInputSignature());
46   CheckOutputArguments (output_args, GetOutputSignature());
47
48   int channels = input_args[0].get()->meta.freeimage.channels;
49   int bpp = input_args[0].get()->meta.freeimage.bpp;
50   int width  = input_args[0].get()->meta.freeimage.width;
51   int height = input_args[0].get()->meta.freeimage.height;
52
53   IP_INT left(input_args[1].get()->data.polygon.values[0].int_val.x);
54   IP_INT top(input_args[1].get()->data.polygon.values[0].int_val.y);
55   IP_INT right(input_args[1].get()->data.polygon.values[1].int_val.x);
56   IP_INT bottom(input_args[1].get()->data.polygon.values[1].int_val.y);
57
58   FIBITMAP* indib(input_args[0].get()->data.freeimage.data.fi_bitmap);
59   FIBITMAP* outdib(FreeImage_Copy (indib, left, bottom, right, top));
60
61   if (outdib == 0) {
62    throw Exception::ExecutionException("Failed to crop image.");
63   }
```

45 - 46 CheckInputArguments and CheckOutputArguments yield exceptions in case the given arguments do not match: the given arguments must match in count (given arguments must be at least of equal size as the expected signatures), in type and regarding meta information. In difference to CheckOutputArguments, CheckInputArguments also ensures the availability of actual data - if an image is expected as input argument, it is mandatory that image data really is available.

62 In case the crop operation failed for various reasons (again, the actual work is handled by FreeImage), an exception is thrown.

2.7 Plugin delegation

Often complex image processing operations involve simple operations. If simple or common image operations are available for re-use, development, maintenance and possibly execution times can be decreased.
The presented image processing library allows the instantiation of foreign image processing operations registered to the system. The actual source of the foreign plugin does not matter at all: it may be known to the system by source-code, as static library or even as dynamic link library. What finally counts for a plugin developer is the functionality. For any plugin, it is possible to query the system for its known plugins and to instantiate an available plugin by its unique name[4], independent of its actual source.
As tutorial example, the median plugin from the previous section is used: as order filter, it has to deploy a strategy for image boundaries. Our median filter creates an artificial border around the given image (the size is depending on the given mask size), it extends the boundary pixel values of the original image to the border. The median operation is then executed on the extended image, afterwards the border is removed by application of the crop plugin from previous tutorial sections. Therefore, the median filter can make use of other fllllip plugins to create and remove the artificial border. The steps required to instantiate and execute foreign plugins are shown subsequently:

- First, member variables to hold foreign plugin instances must be declared:

```
1 /* OrderFilterBase.hpp */
...
67 boost::shared_ptr<ImagePlugin> m_border_plugin;
68 boost::shared_ptr<ImagePlugin> m_rcrop_plugin;
```

  Note: although not necessary, the usage of shared pointers, as shown in the code above, which take care of automatic deallocation, is recommended.

- The foreign plugins are instantiated during construction of the median filter, mainly because our implementation does not provide an alternate solution for the image boundary problem, and therefore relies on the existence of the foreign border and crop plugin:

```
 1 /* OrderFilterBase.cpp */
...
16 m_border_plugin = boost::shared_ptr<ImagePlugin> (
     host->SpawnPlugin ("Size/Border"));
17 if (m_border_plugin.get() == 0)
18  throw Exception::PluginQueryException(
      "Failed to instantiate Size/Border plugin.");
19 m_rcrop_plugin = boost::shared_ptr<ImagePlugin> (
     host->SpawnPlugin ("Size/Crop"));
20 if (m_rcrop_plugin.get() == 0)
21  throw Exception::PluginQueryException(
      "Failed to instantiate Size/Crop plugin.");
```

  As consequence, if either crop or border plugin cannot be created, the construction of the median filter fails.

[4] a plugin name actually consists of a path and a name, for instance "Morphology/Open"

- Because the foreign plugin parameters are depending on median plugin parameters, foreign plugins must be initialized within the Initialize method of the median:

```
59 // setup border plugin
60 Arguments border_params(m_border_plugin->GetParameterSignature());
61 border_params[0].get()->data.value.value.int_val =
62   (m_mask_width / 2) & 1 ? (m_mask_width / 2) + 1 : (m_mask_width / 2);
63 border_params[1].get()->data.value.value.int_val =
64   (m_mask_height / 2) & 1 ? (m_mask_height / 2) + 1 : (m_mask_height / 2);
65 m_border_plugin->Initialize (border_params);
66 // setup crop plugin
67 m_rcrop_plugin->Initialize (m_rcrop_plugin->GetParameterSignature());
```

  As already stated, the border size for the border plugin is dependent on the mask size of the median filter. Because the implementation of the crop plugin does not contain any parameter, we call its initialize method using the (in this case empty) parameter signature of the crop plugin and therefore adhere to the plugin lifecycle (Section 2.5). If the initialization of either foreign plugin fails, the initialization of the median plugin also fails.

2.8 Stateful plugins and serialization

In the fIIIip framework, plugin states are strongly related to the serialization mechanism. In the following, their connection is explained, starting with a section which addresses the different kinds of states and concluding with serialization, which reflects these concepts and enables state conservation.

2.8.1 Plugin states

Often, image processing plugins are stateful: they may have been trained on a specific set of sample data, they may store noise-models of images encountered so far, etc..
On architectural level, there are three kinds of states:

- **Workflow state**: plugin is constructed or initialized.
- **Parameter state**: if the plugin was initialized, it is guaranteed to have specific parameter values, as well as determined input / output signatures.
- **Inner state**: a plugin may learn from applied samples during consecutive calls of the execution method.

There are three kinds of possible instantiation of a plugin:

- **Constructor**: the resulting instance is set to its initial state, but it is not initialized - it has a parameter signature, but its input and output signatures are not guaranteed to be valid, although stateless plugins may determine their I/O signatures already during construction. The dynamic type of the plugin must be known to the caller.

```
/* constructor */
ImagePlugin::ImagePlugin(const ImagePluginHost* host);
```

- **Create** method: this corresponds to the use of the constructor, but without the need to know the dynamic type[5].

```
/* virtual constructor */
virtual const ImagePlugin* ImagePlugin::Create() const;
```

- **Clone** method: this corresponds to the use of a virtual copy constructor, the resulting instance has the same internal state as the original instance. As plugin developer, one can either overwrite the Clone method to implement the proper functionality[6], or use the standard implementation of the Clone method, which relies on the serialization / deserialization mechanism and therefore uses the Serialize and Deserialize methods: First, the actual instance is serialized, a new instance is created using the Create function, finally the internal state is duplicated using the Deserialize method of the new instance. Using serialization mechanisms to archieve state conservation is regarded superior, as it can also be used to persist a plugin.

```
/* stateful virtual constructor */
virtual const ImagePlugin* ImagePlugin::Clone() const;
```

2.8.2 Serialization

Plugin serialization is used for the plugin cloning mechanism as default implementation and allows the conservation of a plugin for future uses. The presented image processing framework does not require or enforce the usage of a specific serialization technique[7], the only demand for a plugin is the ability to serialize and deserialize itself.
Note: a plugin which was not initialized must not be serialized - the call to the serialize method in this case has to fail - this is in the responsibility of the plugin developer who overwrites the serialize method.
In the following, the median filter from previous sections will be extended with a serialization mechanism. To do so, the plugin must provide implementations for the Serialize and Deserialize methods (signatures can be seen in Listing 1):

- The Serialize method has the purpose of converting the actual plugin state into a string of arbitrary layout - XML or JSON could be used as well as a user defined more compact string representation. As already stated, the only requirement for a plugin is the ability to serialize and deserialize itself.

```
/* OrderFilterBase.cpp */
158 /*virtual*/ const IP_BOOL OrderFilterBase::Serialize (
159   std::string& serialization_data) const {
160
161     if (m_mask_width < 0 || m_mask_height < 0) {
```

[5] This technique is also referred to as *virtual constructor idiom*: `http://www.parashift.com/c++-faq-lite/abcs.html#faq-22.5`

[6] Typically, a plugin developer will store the state of the plugin in some member variables. Implementing Clone, the member variables must be simply copied to the new instance.

[7] common examples are XML (`http://www.w3.org/XML/`), YAML (`http://www.yaml.org/`), JSON (`http://www.json.org/`)

```
162     // plugin was not (correctly) initialized
163     return IP_FALSE;
164   }
165   std::stringstream ss;
166   ss << m_copy_image << m_mask_width << m_mask_height;
167   serialization_data = ss.str();
168   return IP_TRUE;
169 } // /*virtual*/ const IP_BOOL OrderFilterBase::Serialize (...) const
```

161 - 168 The implementation shown above first ensures that the plugin was correctly initialized, and afterwards serializes its parameter values into the given string reference, using the *stringstream* standard library - the readability is not very high, but this form of serialization is good enough - the median filter knows how the string is built and therefore can use it to deserialize itself.

- The Deserialize method on the other hand must interpret a given serialization string and use it to restore the encoded state.

```
/* OrderFilterBase.cpp */
171 /*virtual*/ const IP_BOOL OrderFilterBase::Deserialize (
172   const std::string& serialization_data) {
173
174   std::stringstream ss(serialization_data);
175   IP_BOOL copy_image;
176   IP_INT mask_width, mask_height;
177
178   try {
179     ss >> copy_image >> mask_width >> mask_height;
180   } catch (...) {
181     throw Exception::DeserializationException(
        "Order filter deserialization failed.");
182   }
183   Arguments params(GetParameterSignature());
184   params[0].get()->data.value.value.bool_val = copy_image;
185   params[1].get()->data.value.value.int_val = mask_width;
186   params[2].get()->data.value.value.int_val = mask_height;
187   try {
188     Initialize (params);
189     return IP_TRUE;
190   } catch (...) {
191     return IP_FALSE;
192   }
193 } // /*virtual*/ const IP_BOOL OrderFilterBase::Deserialize (...)
```

179 First, the Deserialize method tries to unpack the string and store its values into local variables, again using the *stringstream* library.

181 In case the unpacking fails, a DeserializationException exception is thrown.

183 - 192 Finally, this implementation retrieves the default parameter signature, sets the parameter values according to the unpacked data and calls the Initialize method

using a proper parameter set - even if the instance was not initialized before, the Deserialize call initializes the instance and ensures its validity.

References

[BBKR10] Ulrich Brandstätter, Oliver Buchtala, Thomas Klambauer, and Roland Richter, *Design rationale of the flllip image processing library*, Tech. Report FLLL-TR-1001, Fuzzy Logic Laboratorium Linz, Johannes Kepler University Linz, February 2010.

```
1  namespace flllip {
2  class ImagePlugin {
3
4  public:
5
6          /** Constructor. */
7          ImagePlugin(const ImagePluginHost* host);
8          /// Destructor (virtual).
9          virtual ~ImagePlugin();
10
11         /// Method to retrieve the ImagePluginHost of the plugin.
12         const ImagePluginHost* GetHost() const;
13
14         /** Create function to supply virtual copy constructor functionality. */
15         virtual const ImagePlugin* Create() const = 0;
16         /** Clone function to retrieve a stateful copy of the instance. */
17         virtual const ImagePlugin* Clone() const;
18
19         /** Method to retrieve the API version. */
20         const IP_UNSIGNED GetAPIVersion() const;
21         /** Method to retrieve the version of the plugin. */
22         virtual const IP_UNSIGNED GetPluginVersion() const = 0;
23
24         /** Returns the author of the plugin. */
25         virtual const std::string GetAuthor() const;
26         /** Returns the copyright notice of the plugin. */
27         virtual const std::string GetCopyright() const;
28         /** Returns the description of the plugin. */
29         virtual const std::string GetDescription() const = 0;
30         /** Returns the unique name of the plugin. */
31         virtual const std::string GetName() const = 0;
32
33         /** Method to retrieve the parameter signature of the plugin. */
34         virtual Arguments GetParameterSignature() const = 0;
35
36         /** Method to initialize the plugin using given parameters. */
37         virtual void Initialize (const Arguments& parameters) = 0;
38
39         /** Method to query the input argument signature of the plugin. */
40         virtual Arguments GetInputSignature() const = 0;
41         /** Method to query the output argument signature of the plugin. */
42         virtual Arguments GetOutputSignature() const = 0;
43
44         /** Worker method of the plugin. */
45         virtual void Execute (Arguments& input_args, Arguments& output_args) = 0;
46
47         /** Method to serialize the plugin state. */
48         virtual const IP_BOOL Serialize (std::string& serialization_data) const;
49         /** Method to deserialize the plugin state. */
50         virtual const IP_BOOL Deserialize (const std::string& serialization_data);
51
52 }; // class ImagePlugin
53 } // namespace flllip
```

LISTING 1. The flllip image plugin interface (shortened version of *ImagePlugin.hpp*)

```
#ifndef IMAGE_PLUGIN_TEMPLATE_HPP
#define IMAGE_PLUGIN_TEMPLATE_HPP

#include <ImagePlugin.hpp>

namespace flllip {
namespace flllipin {

class Template: public ImagePlugin {
  typedef ImagePlugin Base;

public:

  Template(const ImagePluginHost* host);
  virtual ~Template();

  virtual const Template* Create() const { return new Template(GetHost()); };

  virtual const IP_UNSIGNED GetPluginVersion() const { return /*00*/1; }
  virtual const std::string GetAuthor() const { return "JKU Linz"; }
  virtual const std::string GetCopyright() const { return std::string(); }

  virtual const std::string GetDescription() const {
    return "flllip template that does nothing.";
  }

  virtual const std::string GetName() const { return "Template/Template"; }

  virtual Arguments GetParameterSignature() const {
    return GetHost()->CopyArguments(m_parameters);
  }

  virtual void Initialize (const Arguments& parameters);

  virtual Arguments GetInputSignature() const {
    return GetHost()->CopyArguments (m_input_args);
  }
  virtual Arguments GetOutputSignature() const {
    return GetHost()->CopyArguments(m_output_args);
  }

  virtual void Execute (Arguments& input_args, Arguments& output_args);

  virtual const IP_BOOL Serialize (std::string& serialization_data) const;
  virtual const IP_BOOL Deserialize (const std::string& serialization_data);

protected:

  Arguments m_parameters, m_input_args, m_output_args;

}; // class Template: public ImagePlugin
} // namespace flllipin
} // namespace flllip

#endif // IMAGE_PLUGIN_TEMPLATE_HPP
```

LISTING 2. flllip template header file

```cpp
#include <ArgumentChecks.hpp>

#include "Template.hpp"

namespace flllip {
namespace flllipin {

Template::Template(const ImagePluginHost* host): Base(host) {}

/*virtual*/ Template::~Template() {}

/*virtual*/ void Template::Initialize (
 const Arguments& parameters) {

  CheckParameters (parameters, m_parameters);
} // /*virtual*/ void Template::Initialize (...)

/*virtual*/ void Template::Execute (
 Arguments& input_args, Arguments& output_args) {

  CheckInputArguments (input_args, GetInputSignature());
  CheckOutputArguments (output_args, GetOutputSignature());
  // does otherwise nothing...

} // /*virtual*/ void Template::Execute (...)

/*virtual*/ const IP_BOOL Template::Serialize (
 std::string& serialization_data) const {

  return IP_FALSE;
} // /*virtual*/ const IP_BOOL Template::Serialize (...) const

/*virtual*/ const IP_BOOL Template::Deserialize (
 const std::string& serialization_data) {

  return IP_FALSE;
} // /*virtual*/ const IP_BOOL Template::Deserialize (...)

} // namespace flllipin
} // namespace flllip
```

LISTING 3. flllip template implementation file

```
1  typedef struct _ImagePluginArgument {
2
3      /** Pointer to serialization function. */
4      IP_BOOL (*Serialize) (struct _ImagePluginArgument* argument);
5      /** Pointer to deserialization function. */
6      struct _ImagePluginArgument* (*Deserialize) (
7        struct _CImagePluginHost* host, IP_CSTR data);
8
9      /** Type of the argument. */
10     enum _ImagePluginArgumentType type;
11     /** Argument data. */
12     union _ImagePluginArgumentData data;
13     /** Argument data description. */
14     union _ImagePluginArgumentMeta meta;
15
16     /** Argument flags. Refer to @see _ImagePluginDataFlags */
17     IP_INT flags;
18     /** Human-readable name of the argument. */
19     IP_CSTR name;
20     /** Human-readable description of the argument. */
21     IP_CSTR description;
22
23     /** Pointer to the host which allocated the argument.
24         This allows one-side data allocation and deletion. Additionaly,
25         on C++ side, the host pointer can be used for automatic resource
26         deletion using a smart pointer. */
27     struct _CImagePluginHost* host;
28
29     /** Serialization data. */
30     IP_CSTR serialization_data;
31
32 } ImagePluginArgument;
```

LISTING 4. Argument struct (CImagePluginArgument.h)

```
1  Argument ImagePluginHost::CreateUserArgument (...) const;
2  Argument ImagePluginHost::CreateImageArgument (...) const;
3  Argument ImagePluginHost::CreateValueStringArgument (...) const;
4  Argument ImagePluginHost::CreateRangeArgument (...) const;
5  Argument ImagePluginHost::CreatePolygonDoubleArgument (...) const;
6  Argument ImagePluginHost::CreateArrayIntArgument (...) const;
7  Argument ImagePluginHost::CreateMatrixDoubleArgument (...) const;
8  Argument ImagePluginHost::CreatePropertyListArgument (...) const;
```

LISTING 5. Argument creation functions (extract)

```
1  namespace flllip {
2
3  void CheckParameter (
4   const Argument& given,
5   const Argument& prototype);
6
7  void CheckParameters (
8   const Arguments& given,
9   const Arguments& prototype);
10
11 void CheckInputArgument (
12  const Argument& given,
13  const Argument& prototype);
14
15 void CheckInputArguments (
16  const Arguments& given,
17  const Arguments& prototype);
18
19 void CheckOutputArgument (
20  const Argument& given,
21  const Argument& prototype);
22
23 void CheckOutputArguments (
24  const Arguments& given,
25  const Arguments& prototype);
26
27 } // namespace flllip
```

LISTING 6. Argument checking methods (extract from ArgumentChecks.hpp)

```
1  namespace flllip {
2  namespace Exception {
3
4  class Exception: public std::exception {...}
5
6  class NotInitializedException: public Exception {...}
7
8  class InitializationException: public Exception {...}
9
10 class ParameterException: public Exception {...}
11
12 class MissingParameterException: public ParameterException {...}
13
14 class IncompatibleParameterException: public ParameterException {...}
15
16 class IncompleteParameterException: public ParameterException {...}
17
18 class ArgumentException: public Exception {...}
19
20 class MissingArgumentException: public ArgumentException {...}
21
22 class IncompatibleArgumentException: public ArgumentException {...}
23
24 class IncompleteArgumentException: public ArgumentException {...}
25
26 class PluginQueryException: public Exception {...}
27
28 class MemoryException: public Exception {...}
29
30 class ExecutionException: public Exception {...}
31
32 class SerializationException: public Exception {...}
33
34 class DeserializationException: public Exception {...}
35
36 class MissingHostException: public Exception {...}
37
38 } // namespace Exception
39 } // namespace flllip
```

LISTING 7. flllip Exceptions (extract from ImagePluginException.hpp)

Tomas Kazmar, Thomas Trenker

VMI-CME using KNIME

intention, intermediate results, outlook

Within the SCCH project VMI-CME the team analyses different modelling platforms suitable for image processing. One of these platforms is the Konstanz Information Miner (KNIME [naim]). KNIME is a modular data exploration platform (based on Eclipse) that allows the visual creation of data flows, the execution of analysis steps, and the generation of interactive output views. Aspects of the projects are the usage of integrated image processing features as well as the extension of KNIME to allow the integration of any image processing library available.