s c c h
software competence center
hagenberg

FUZZY.LOGIC.LAB.LINZ

Advances in Integrated Software Sciences

Proceedings of the
Master and PhD Seminar
Winter term 2009/10, part 1

Softwarepark Hagenberg
SCCH, Room 0/2
2 February 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

9:00–10:00 Session 1 ***(Chair: Roland Richter)***

9:00 Georg Buchgeher:
Tool Support for Continuous Software Architecture Analysis

9:30 Thomas Redenbach:
BvLib – The Image Processing Framework at the Fraunhofer ITWM

10:00 Coffee break

10:15–11:15 Session 2 ***(Chair: Thomas Natschläger)***

10:15 Thomas Klambauer:
Generic Image Processing – Plug-ins in C++

10:45 Roland Richter:
Design rationale of the flllip image processing library

Tool Support for Continuous Software Architecture Analysis

Georg Buchgeher

georg.buchgeher@scch.at

Extended Abstract

I Introduction

Software architecture analysis is the process of assessing the potential of a proposed architecture to deliver a system capable of fulfilling the required functional and non-functional requirements and to identify any potential risks [1]. Architecture analysis is an important means for risk reduction [2] and quality assurance [3] and helps finding architectural problems (risks) early in the development (before the implementation), when resolving them is still cheap [4]. In a software development project not only the architecture design needs to be analyzed but also architecture descriptions and the system implementation must be analyzed. Architecture descriptions can be analyzed for consistency and completeness, the system implementation must be analyzed for conformance with the software architecture.

Software architecture takes a key role between the requirements and the system implementation [5]. Typically requirements, architecture and implementation are developed incrementally [6]. During architecture design and system construction new and changed requirements emerge, which must be addressed in the architecture and implementation accordingly. Also in agile software development processes like Scrum and XP the architecture is developed incrementally. Given the incremental nature of architecture design we argue that also architecture analysis must be performed continuously throughout all activities of the software development process. However existing analysis techniques have been developed to be performed only at a single point in the development process [7]. They suffer from several deficiencies, which makes it hard to apply the continuously.

The contributions of this paper are twofold. First we describe the deficiencies of existing analysis techniques with regard to applying them continuously and derive requirements for continuous architecture analysis. Second we present LISA an approach that supports continuous software architecture analysis.

II Problems with the State of the Art

Today various software architecture analysis techniques exist differing in costs and complexity [6]: Scenario-based evaluation methods, formal and structured reviews, software architecture management tools, architecture description languages, prototypes and proof of concept systems and skeleton systems. A detailed analysis of these analysis techniques reveals the following obstacles for applying the continuously:

- Analysis approaches that are performed manually are time intensive and thus expensive: Scenario-based evaluation methods and structured reviews take between 30 and 60 staff days.

- All analysis approaches have been developed to be performed at a single point in the development process. Further they do not support incremental analysis.
- Automatic analysis approaches can only be applied for the analysis of single quality attributes but not for the architecture as a whole.
- Manual analysis approaches lack tools that support the analysis process.

In order to perform architecture analysis continuously an architecture analysis technique must fulfill the following requirements:

- Architecture analysis techniques must be inexpensive.
- Architecture analysis techniques must support incremental analysis.
- Architecture analysis techniques must support the analysis of all system goals.
- Architecture analysis must consider multiple stakeholders.

III LISA

LISA (**L**anguage for **I**ntegrated **S**oftware **A**rchitectures) is an approach to support architecture-centric activities like architecture design, architecture analysis and architecture implementation throughout the software development process. LISA consists of two main parts: An architecture description model and a toolkit working on this model. We have already described several aspects of LISA in other publications [8-12]. In this paper we focus on the support for continuous architecture analysis. Table 1 gives an overview how we address the requirements for continuous architecture analysis in LISA.

Requirement	Addressed through
Inexpensive Analysis	• High degree of automation
Analysis of all system goals	• Combination of multiple analysis approaches (automatic + manual) • Extensibility of model and toolkit
Support for incremental analysis	• Support for incomplete architectures • Customizable analysis
Support for multiple stakeholders	• View-based architecture description • Customizable analysis

Table 1: Requirements for continuous architecture analysis and their realization in LISA

High degree of automation. We reach a high degree of automation in multiple points: The use of a semi-formal architecture model allows the automatic analysis of the formally described parts of the architecture description. Also the architectural views are generated from the semi-formal architecture description. Further we support the definition of the architecture description by automatically extracting architecturally significant information from the system implementation – if a system implementation is already available. This information is then extended with architectural information is not contained in the implementation. We also support the partial generation of the system implementation from the architecture description.

Combination of multiple analysis approaches. In order to support the analysis of the architecture as a whole we combine multiple analysis approaches in one single environment. We use automated analysis where possible. Quality attributes that are hard to analyze automatically can be analyzed by using manual analysis techniques like scenarios.

Extensibility of model and toolkit. LISA is an extensible architecture analysis platform. The LISA model defines a set of core models that can be extended with additional submodels that describe additional architectural aspects. Such a model extension must be accompanied with a corresponding extension of the toolkit that performs the automatic analysis of the submodel and a corresponding user interface needed for modifying the submodel.

Support for incomplete architectures. As described above, architecture definition is an incremental process. Typically architectures are defined using a top-down approach, where a high-level architecture description is incrementally refined towards a more detailed description. LISA supports the description of an architecture at multiple abstraction levels and supports the stepwise refinement of an initially incomplete architecture into a complete architecture description. We have described the support for incomplete architecture descriptions in [12].

Customizable analysis. Different stakeholders have different architectural concerns. This concern-orientation must also be reflected when analyzing the architecture. LISA offers a customizable architecture analysis functionality based on an extensible set of constraints. Based on these constraints the user defines specific analysis he is interested in. Single analysis can be activated and deactivated that toolkit only analysis and visualizes problems of a certain analysis.

View-based architecture description. In order to support multiple stakeholders LISA visualizes software architectures in multiple views. Each view focuses on a single aspect of the architecture. We have developed views for visualizing static structures like modules and package dependencies, views for describing runtime structures that describe the system as a set of interacting components and views that visualize the architecturally significant requirements. Further we currently are working on additional views for security aspects and product line architectures.

IV References

[1] M. Ali Babar, L. Bass, and I. Gorton, "Factors influencing industrial practices of software architecture evaluation: An empirical investigation," in *Software Architectures, Components, and Applications*, ser. Lecture Notes in Computer Science, S. Overhage, C. A. Szyperski, R. Reussner, and J. A. Stafford, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 4880, ch. 6, pp. 90-107.

[2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003.

[3] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, January 2009.

[4] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, Mas, and M. Pizka, "Tool support for continuous quality control," *Software, IEEE*, vol. 25, no. 5, pp. 60-67, 2008.

[5] D. Garlan, "Software architecture: a roadmap," in *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA: ACM Press, 2000, pp. 91-101.

[6] N. Rozanski and E. Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, April 2005.

[7] M. A. Babar, L. Zhu, and R. Jeffery, "A framework for classifying and comparing software architecture evaluation methods," in *ASWEC '04: Proceedings of the 2004 Australian Software Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 309+

[8] G. Buchgeher and R. Weinreich, "Integrated software architecture management and validation," *Software Engineering Advances, International Conference on*, vol. 0, pp. 427-436, 2008.

[9] G. Weiß, G. Pomberger, W. Beer, G. Buchgeher, B. Dorninger, J. Pichler, H. Prähofer, R. Ramler, F. Stallinger, and R. Weinreich, "Software engineering – processes and tools," in *Hagenberg Research*, B. Buchberger, M. Affenzeller, A. Ferscha, M. Haller, T. Jebelean, E. P. Klement, P. Paule, G. Pomberger, W. Schreiner, R. Stubenrauch, R. Wagner, and G. Weiß, Eds. Berlin: Springer, 2009, pp. 157-235.

[10] G. Buchgeher and R. Weinreich, "Connecting architecture and implementation," in *On the Move to Meaningful Internet Systems*: OTM 2009 Workshops, ser. Lecture Notes in Computer Science, R. Meersman, P. Herrero, and T. Dillon, Eds., vol. 5872. Springer, 2009, pp. 316-326.

[11] G. Buchgeher and R. Weinreich, "Tool support for component-based software architectures," *Asia-Pacific Software Engineering Conference*, vol. 0, pp. 127-134, 2009.

[12] R. Weinreich and G. Buchgeher, "Paving the Road for Formally Defined Architecture Description in Software Development", *25th ACM Symposium on applied computing* (to appear), 2010.

BvLib - Image Processing Framework at ITWM

Thomas Redenbach and Andreas Jablonski
Fraunhofer ITWM Kaiserslautern

2 February 2010

Abstract

The department *Image Processing* at the Fraunhofer Institute for Applied Mathematics (ITWM) engages in applied research projects while also offering services to industrial customers. In order to make research results available in an industrial context quickly, a unified software framework was requested by our scientists. The presentation introduces BvLib, a software library which meets our special requirements for the development of 2- and 3-dimensional image processing algorithms. In addition, "ToolIp", a graphical tool for creating complex algorithm graphs will be presented.

In der Abteilung *Bildverarbeitung* am Fraunhofer ITWM werden sowohl Forschungsarbeiten als auch Projekte für Industriekunden durchgeführt. Um Forschungsergebnisse schnell im industriellen Umfeld nutzen zu können, wurde von unseren Wissenschaftlern eine vereinheitlichte Softwarearchitektur gewünscht. Der Vortrag stellt Bvlib, eine Software-Bibliothek vor, welche unseren speziellen Anforderungen zur Entwicklung von 2- und 3-dimensionalen Bildverarbeitungsalgorithmen erfüllt. Zusätzlich wird das grafische Werkzeug "ToolIp" zum Erstellen komplexer Algorithmengraphen gezeigt.

Generic Image Processing

Thomas Klambauer
Johannes Kepler University

Thomas@Klambauer.info

Abstract

Selected parts of the sections "Design Considerations" and "Evaluations" of [1] are presented. "Design Considerations" focuses on the problems of plug-in systems developed in and for the C++ programming language. The memory layout of objects, runtime library instances, error handling, type information and related topics of interest at library boundaries, as well as alleviations therefore are discussed. "Evaluations" will briefly present the Qt and FxEngine plug-in facilities.

References

[1] Thomas Klambauer. Generic Image Processing - Plug-ins in C++. Bachelor Thesis, Johannes Kepler University, `http://klambauer.info`, December 2009.

Technical Report
FLLL–TR–1001

FUZZY.LOGIC.LAB.LINZ

Design rationale of the flllip image processing library

Ulrich Brandstätter, Oliver Buchtala, Thomas Klambauer, 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 existing 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++. flllip is short for *FLLL's image processing plugins*. FLLL, in turn, is short for Fuzzy Logic Laboratorium Linz, the department of Johannes Kepler University Linz where the framework was developed.

1.1 Motivation

Several recent projects at our department included a considerable amount of work on (industrial) image processing algorithms. These projects typically

- offered lots of images of different types from different fields of application,
- involved a number of algorithms for these fields, developed with serval C/C++ third party libraries, such as Boost.GIL[1], OpenCV[2], FreeImage[3], Intel® IPP[4], and MATLAB®[5],
- and – unfortunately – made it necessary to create a number of project-specific test tools.

Often, these algorithms were first developed and tested with a *project-specific* tool, then integrated into a *vendor-specific* runtime environment. This transition, however, usually was far from being smooth. Being sick of reinventing the wheel over and over again, we tried to bring our image processing experiences down to a common denominator.

The bachelor thesis of one of the authors [Kla09] served as starting point for the work on the overall architecture of the framework. For a thorough discussion of some design possibilities, as well as a comparison of several other plugin frameworks, we refer to this thesis.

In the sequel, we will describe several alternative designs which were considered, as well as which design made it into the final architecture of the flllip framework.

2 Design rationale

As pointed out above, the requirement for an image processing superstructure lead to the development of the flllip framework. flllip is not intended to be a replacement for other C++ image processing frameworks. It is designed as a framework providing a plugin mechanism for algorithms from various sources, including methods from sophisticated frameworks, home-grown code, but also image processing procedures from non-C++ environments, such as MATLAB®.

2.1 Fundamental architecture decisions

When we started to think about the architecture of our framework, we faced two fundamental design considerations.

[1] `http://opensource.adobe.com/wiki/display/gil/Generic+Image+Library`
[2] `http://sourceforge.net/projects/opencvlibrary/`
[3] `http://freeimage.sourceforge.net/`
[4] `http://software.intel.com/en-us/intel-ipp/`
[5] `http://www.mathworks.de/`

1. The fundamental motivation for any plugin framework is the urge for code reuse. Code reuse can be accomplished in multiple ways:

   (a) using functions

   (b) using templates

   (c) using executable files

   (d) using dynamic libraries

   We agreed on using dynamic libraries, as they separate application and image processing logic better.

2. We don't want to roll out another discussion on the benefits and disadvantages of different programming languages. As we develop image processing tools in the context of industrial applications we need a language which is efficient, relatively close to hardware, and comfortable to use. We agreed on C++ as main development language, and we wanted to use available methods by a uniform interface.

For various (technical) reasons, the combination of using dynamic libraries in conjunction with C++ is not straightforward, and requires some careful design decisions, as shown subsequently in Section 2.2.1.

2.2 Framework design rationale – a view from the outside of a plugin

2.2.1 Crossing dynamic link library boundaries when using C++

C++ offers multiple ways to adopt dynamic link libraries, each with its individual shortcomings:

- Exporting / importing **C++ classes and methods**:
  While this possibility is probably the easiest to implement, it quickly turned out to be a solution not acceptable for its manifold problems: name mangling, dependency on specific compiler, compiler version, compiler build target (debug / release) and even compiler flags.

- Using **abstract base classes** for dynamic link library export and import:
  This possibility is more promising, and is used by the prominent COM architecture. Although we adopted this possibility for other projects, it too has its pitfalls: typically it is not possible to use virtual destructors, to provide any default implementation, it requires compilers to adopt the same vtable layout, yet worst, it does not allow to use complex types in the interface (such as vectors - their binary layout, which is independent from the interface, may differ).

- Using **C-only** dynamic link libraries:
  Although being limited to C, regarding dynamic link library import and export, is a severe restriction, we agreed on this way: C provides a stable ABI, so none of the shortcomings of the other possibilities are suffered. In addition, we managed to circumvent the C restriction by the creation of an adapter framework.

Adapter framework

Our adapter framework, which is responsible for the translation of C++ image processing plugins to C plugins for dynamic link library export and vice versa consists of two components:

- **C++ to C adapter**
  This adapter converts any C++ image processing plugin, which implements our C++ class interface, to a C image processing plugin. In short, it wraps all C++ specifics from the class interface to proper C only representations. This includes the translation of any caught exception (C++) to simple error codes. It also takes care for the synchronization of C++ class and its C counterpart regarding object lifetime: If the C plugin is deleted, this is also true for the C++ class.
- **C to C++ adapter**
  Because C++ is our main development language, we also want to use available image processing algorithms in C++. Therefore an adapter, which converts a C image processing plugin to its C++ counterpart, is also provided. Like before, this adapter also synchronizes object lifetimes.

2.3 Plugin design rationale – a view from the inside of a plugin

2.3.1 Working function signature

We want to enable simple filter-type plugins (one input image transferred to one output image), as well as plugins of arbitrary I/O signature complexity, regarding number and type. This has to be supported by ImagePlugin, more precisely, by its Execute(...) method. Considering the "..." part of "Execute(...)", i.e., the function signature, there are several possibilities:

1. Result Execute(InputArgument1, InputArgument2, ..., InputArgumentN)
2. OutputArgumentList Execute(InputArgumentList)
3. Execute(InputAndOutputArgumentList)
4. Execute(InputAndOutputArgumentList&)
5. Execute(InputArgumentList, OutputArgumentList)
6. Execute(InputArgumentList&, OutputArgumentList&)

Several considerations shortened the list above:

- There might be more than one result (discards 1).
- We don't want to have manifold method signatures (especially, since we do NOT want to have one single base class "Argument", see below) (discards 1, again).
- We want to avoid copying where possible (discards 1, 2, 3, 5).

- Some arguments might be input as well as output, which we want to support mainly for performance reasons; that is, we want to support some kind of "C++ reference style" (discards 1, 2, 3, 5).

With only option 4 and 6 left, we finally settled on 6: first, it is slightly more convenient to use, for not every argument has to be tested if it is input or output. Second, it is easier to understand.

2.3.2 Arguments for image processing plugins

Assume that a plugin creates a new image, and returns it; or assume that the result image of one plugin is passed as an argument to another plugin. This scenario has a number of consequences:

- The lifetime of plug-ins and arguments must be decoupled.
- Ressources must not be managed by plugins (but by the host[6]).
- For performance and memory reasons, call-by-value must be avoided with large data blocks.
- Ressource leaks must be avoided.
- Allocations and de-allocations must happen on the same side of the dll boundary.
- The result of a plugin can also be used as input argument, or even as parameter, by another plugin.

Regarding the implementation of arguments, there were several options:

- Using **void**$*$, which is both unsafe and ugly.
- Deriving everything from one single base class (such as *Object* in *Java*). This option has several issues again:
  - How to pass these "Objects" over the C ABI without performance loss?
  - This might induce a compiler dependency, due to vtable layout.
  - This would require RTTI, hence enlarging objects.
- Serializing and deserializing everything, which most likely causes significant overhead, especially when dealing with larger types. Note: this is the way we go when a plugin developer introduces a new argument type.
- Using C POD types for arguments turned out as (compromise) solution:
  - Slightly faster data access (in comparison to objects with base class).
  - A combination of a C union (with data types focused on image processing, and a type id gives type safety to a certain degree.
  - Unproblematic regarding dynamic link library boundaries, because of consistent object layout due to C ABI.

[6] The host in this case refers to its function as central ressource manager. See the tutorial for additional information.

2.3.3 Plugins call other plugins

We want to reuse existing functionality (home grown, as well as foreign), therefore we want to enable a plugin to call another plugin. Once again, there were several design considerations:

- A plugin should not be burdened by the plugin loading mechanism.
- There must be a centralized point where plugin loading is done.
- There must be a simple, yet unambiguous way to query the availability of foreign plugins - we use an URL-style plugin registration mechanism.

2.3.4 Parametrized plugins

We want plug-ins to be parametrized. During the design phase, it turned out that "Parameter" is similar to "Argument"; the distinction is just on a semantic level. The introduction of plugin parameters leads to a specific plugin lifecycle, as well as to plugin states.

2.3.5 Plugin expectations

A plugin must possess facilities to tell which kind of data it expects:

- On the one hand, a plugin expects a (fixed) set of Parameters.
- On the other hand, a plugin has to provide means to query its signature. This is not limited to returning the types which Initialize () or Execute() expects; it should also be possible to communicate that the method expects, for instance, an image of certain colour depth and/or size.

Therefore, we introduce the notion of *Signature*: a signature is an argument without actual data - it only yields meta-information about the kind of argument it expects. Every argument type has meta information.
Very important: We want plugins to be able to change their signatures, depending on given parameters. For an example, see ...

2.3.6 Plugin lifecyle

Several of the above points indicate that the plugin life cycle consists of separated steps:

- Construction: after plugin construction, a plugin must be able to satisfy a request for its parameter signature.
- Initialization: is semantically equivalent to plugin parametrization, since the Initialize () method expects parameters. After plugin initialization, a plugin must be able to satisfy request for its input and output signatures (since they may be dependent on plugin parametrization). A plugin may be initialized many times.
- Execution.
- Deletion.

2.3.7 Image argument

We are focusing on image processing; we want to support as many image types as possible, as well as images with meta-data, images with multiple pages, etc.. We considered Boost.GIL, OpenCV, FreeImage, libtiff, and a home-grown type, yet we agreed on FreeImage:

- Fast image container (yet with some drawbacks, such as alignment issues).
- Lots of supported image types.
- C container.

Note, however, that it would be possible to introduce another (or an alternative) image data type without much impact on the rest of the framework.

2.3.8 Unified error handling

Potentially, errors of various kinds can occur during plugin construction, initialization, execution and (de-)serialization. We agreed on the following:

- We use exception handling, adhering to the standard C++ rules.
- We supply a set of image processing exceptions.
- Because errors regarding arguments are very common (at least in our experience), we also provide a set of functions to ease the parameter and argument validation.

2.3.9 Argument / plugin persistation

We want to store any result, as well as any plugin, to the harddisk and reload it when necessary. We agreed on the following:

- Regarding persisted argument types, they should be human readable. Also they should be intuitive - a matrix should look like a matrix -, and simple to edit. Therefore, we implemented a home-grown serialization mechanism.
- Regarding plugins, we honour the freedom of a plugin developer: he may adopt any serialization mechanism (including XML, JSON, Yaml, etc.), the only requirement is the ability to desirialize properly.

2.3.10 Decoupling of application and plugin logic

We want to decouple application and plugin logic:

- We adopt dynamic link libraries for the loose coupling.
- We split our framework into three components:
  1. flllip, the core framework.
  2. flllipin, a set of image processing plugins.
  3. flllipbook, a GUI frontend.

2.3.11 Additional programming languages

We want to use languages in addition to C++. Due to the decision to base on a C ABI it is possible to enable other languages (with C bindings), as well.

- C is natively supported.
- MATLAB®code can be loaded using a special plugin.
- Any language with C bindings can possibly be used (requires some work).

2.3.12 All for plugin developers

We want to favour plugin developers where possible:

- Rather simple C++ interface, few methods to implement.
- Many convenience methods (especially for argument creation and validation).
- Automatic ressource clean up (of Arguments, Parameters and foreign plugins).
- Few stumbling blocks (FreeImage has some...).
- Documentation: there are lots of code examples (home-grown, IPP, CImg, Tesseract, FreeImage, ...), Doxygen documentation, tutorial, ...

2.4 Design goals

In the following, a list containing accomplished design goals and relevant design decisions is presented:

- Above all, we favour plugin developers where possible: to ensure a simple and easy to understand interface, the internals of the fllllip framework are rather complex. In other words: whenever there was a design decision regarding the simplicity of the framework for any role, plugin developers were favoured.
- fllllip allows an easy reuse of existing functionality: any plugin can query the central registrar (which we call host) for the availability of *registered* other plugins and execute them, if desired, in a simple way.
- We use both C and C++ programming languages: C++ is the primary plugin development language (although any other language with C bindings can be used as well, provided that proper bindings for fllllip are available[7]), and C is used because of its binary compatibilty. With the notable exception of the provided data types, a C++ developer using fllllip is not constricted by C in any way. As consequence, C++ developers are not dependent on particular compiler vendors or settings.

[7] MATLAB® bindings are already available

- For flllip, the origin of an image processing plugin does not matter at all. Plugins can be used from source code, from static libraries or from DLLs, for the framework this is completely transparent.
- In the course of designing the framework, differentiating between three parts made design simpler:
  1. flllip: the core library, providing the basic framework mechanisms
  2. flllipin: a collection of algorithms using flllip
  3. flllipbook: a graphical user interface to work with flllipins
- Resource efficiency: digital image processing often involves large data blocks (images, matrices, etc.). Their copying is problematic not only because of the memory usage, but also because of the time the copying takes. Therefore, flllip involves a central resource allocator, the host, which conceptually avoids multiple copies.
- To enforce that the data delivered from one plugin to another is the same all way, all available data types are implemented as C types[8]. They only involve reasonable overhead, they do not use void pointers at all, therefore a certain degree of type safety is archieved. As C types, inheritance is not used for data (f.i. derive everything from "'Object"').
- flllip involves mechanisms for automatic resource de-allocation for C++ developers. If any (exported) data block is no longer used, its memory is deleted automatically (using the central resource allocator).
- Although the framework is kept as generic as possible, it still focuses on image processing. flllip uses FreeImage as sole image container - we considered several alternatives, such as Boost.GIL, OpenCV, etc., yet FreeImage turned out as the favourable library.
- flllip distinguishes between algorithm parameters and arguments on a pure semantic level: although both are, in fact, based on the same type, they play a different role and enable regulation and control tasks for image processing.
- Serialization plays a major role for flllip: it not only enables the persistation of stateful plugins, it also enables the persistation of any parameter and argument supported, therefore image processing results can be easily preserved. Also, the plugin serialization mechanism is used to create stateful clones of plugin instances per default.
- flllip delegates 3rd party dependencies to the plugins: if a plugin is dependent on a 3rd party library, this is not visible outside to the outside, a flllip-enabled application need not be aware of any plugin dependency. At the FLLL, we organized our plugin projects according to their third-party dependencies.

References

[Kla09] Thomas Klambauer, *Generic Image Processing - Plug-ins in C++*, Bachelor Thesis, Johannes Kepler University, December 2009.

[8] so callod POD - plain old data - types