

# Generic Image Processing

Thomas Klambauer\*

April 27, 2009

## Abstract

The generic composition of image processing components poses a non-trivial software architectural problem. Especially dealing with multiple inputs and outputs of different types per component and the handling of parameters as well as component interconnection introduce substantial complexity. A close analysis of the problem and the multitude of desirable properties will be presented. Possible solutions together with a short assessment of popular in-use architectures are explored.

## 1 Introduction

Modularization and composition of software components as well as the design of stable generic interfaces are some of, if not *the* most prevalent problems in software design. As precursor of the same-titled bachelor thesis<sup>1</sup>, this paper will address this problem - also by restricting itself to the specific requirements of the environment of the accompanying project.

The goal is to devise a framework architecture that allows for the dynamic integration of components - *plugins* - without changes to the hosting application or other components.

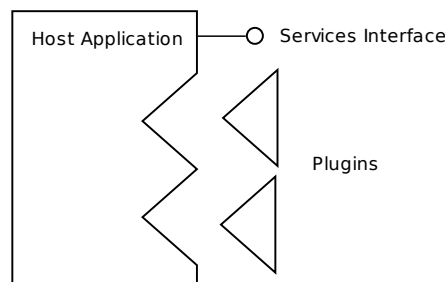


Figure 1: Plugin Architecture (Component Level)

It is to be realized on the *Windows* platform with implementations and interfaces specified in *C++* supported by Microsoft tools: compiler, linker, etc. No cross-machine - thus no cross-architecture - situation is part of the usage scenario,

---

\*Thomas@Klambauer.info

<sup>1</sup>To be available at <http://klambauer.info> by August 2009.

thus the operation of the framework is always restricted to a local machine. To stay in scope the discussion will not consider multiple processes for components and assume a single process to contain all code and data. Furthermore Microsofts *Component Object Model (COM)*[1] is not considered for complexity reasons.

## 2 Problem Area Analysis

Integral problem fields are the *Application Programming Interface (API)*, the *Application Binary Interface (ABI)* and *Module interconnection*.

### 2.1 API

The typical use case for the framework is the processing of image data by an image processing algorithm. Thus the passive, algorithmic view of a component is predominant and we expect a component to receive data as input from probably multiple sources and multiple types, to process it and yield probably multiple outputs of different types. This view also assumes that all the thread management lies with the host application and no relevant component-owned threads will exist.

Data types include *image data* (including different color types) and *related information* like descriptions of geometric objects in the image, text, positional information and calculation results. Future additions to this list are likely and upwards compatibility is desirable. Figure 2 illustrates these goals.

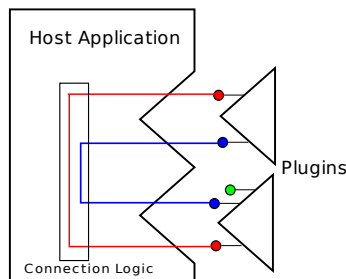


Figure 2: Plugin Interconnection

A central problem is the type-safe, generic interconnection of plugins and here, the lack of *reflection* in C++ prominently enters the stage. Concerning type safety some options are reasonable, but which also affect the API as a direct result:

**Centralized, static types** All cross-interface types and interfaces are managed centrally through *headers* and/or some *shared base library*. Depending on interface conventions this might at least apply to C and C++ standard library types as well as the Boost.GIL[2] which is being projected as image carrier. However for other more frequently changing composite types this approach greatly reduces modularity and independence of plugins and the host.

**Dynamic types** enabled by a runtime type system with basic reflection-like functionality. Though the most versatile approach, much naturalness in the interface design is lost - interface functions and their arguments to be registered for export; composite datatypes requiring a nontrivial setup.

We can clearly see the correlation of those concepts to statically and dynamically typed (scripting) languages in coherence to the tasks at hand. Between those two extremes there is some room for interpolation. For instance an *interface querying* mechanism like in use in COM seems feasible.

Caution is furthermore required concerning C++ language specifics and the crossing of module (OS library) boundaries. Template types for instance need to be *instantiated* to be exported as only a few (Comeau, Borland) Compilers support the module-level *export* of uninstantiated templates.

Another consideration stemming from performance concerns - as the data dealt with are likely to be megabyte ranged image objects - is the *ownership* of objects that will be passed by reference. More specific, it has to be ensured that memory occupied by interface-passed object will be freed. The usual C++ way to guarantee this, is the wrapping into a ownership-managing template class like `std::auto_ptr<T>` or `boost::shared_ptr<T>`. However this brings us to the topic of binary dependencies between modules.

## 2.2 ABI

*Memory management*, already quite complex outside of garbage collected runtimes, is faced with another dimension when references to dynamically allocated memory pass module boundaries. The C++ standard dictates that heap and free store<sup>2</sup> operations must not be mixed on the same memory chunk due to probably independent implementations and thus resulting corruption of memory management data structures. A key observation here is that the same set of problems can occur with simply different runtimes or implementation versions of the C or C++ runtime libraries.

Consider two shared libraries (DLLs) both linking statically to the C++ runtime library meaning that during runtime static data structures and algorithms will not be shared between them. When `delete`ing an object created by the other library and passed over an interface, this will result in heap corruption of the runtime [3, 4]. Two solutions to this problem are illustrated in figure 3<sup>3</sup>.

Here the first solution requires that each object is handed back to the source for deallocation near its creation point, often seen as a `create`, `destroy` function pair. The second approach builds on some enforcement or trust that the same instance of the runtime will be shared dynamically such that *all* plugins need to use exactly the same version of the runtime(s) *and* must not statically link to them.

The use of `boost::shared_ptr` (also part of C++ TR1 and the future C++0x standard library) solves this problem by carrying a *deleter function pointer* that will call the appropriate deallocator.

## 2.3 Plugin Interconnection

A distinguishing peculiarity of the desired framework in comparison to other plugin architectures is that plugins primarily *not* work with some interface on the host application side, but (also) with other plugins with (probably) variable interfaces.

This generic interconnection issue could be solved by following methods:

---

<sup>2</sup>heap: `malloc, free`; free store: `new, delete`

<sup>3</sup>Picture by Alex Blekhman

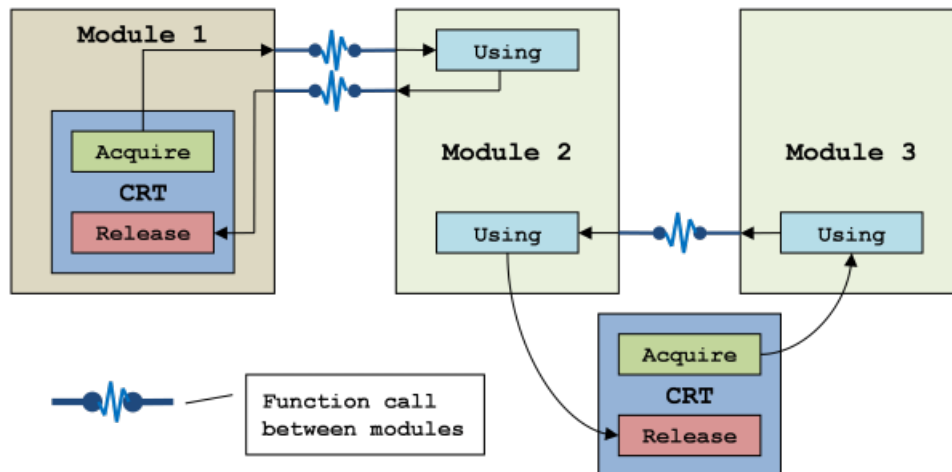


Figure 3: C,C++ Runtimes and Modules

**Dynamic types** by enabling a runtime configurable connection between plugins, similar to a scripting language - a *Module interconnection language* - being able to evaluate and integrate code at runtime.

**Connector plugins** that have knowledge of the static types of the actual interfaces and by this leaving the host application independent of the actual plugins, but moving this burden into a separate plugin. Drawbacks include that this is an indirect approach and the number of connectors may become quadratic in the number of plugins.

An interesting mixed approach is presented as a *Generic function-binding interface* in [8] by Scott Bilas:

On the Windows platform, at runtime the export table of modules can be read out and using Microsoft supplied helper DLLs, names can be unmangled and used further processing. This allows to reflect a modules/plugins interface at runtime and could be used to dynamically setup connections.

The author supplied skeleton implementation code to assert feasibility, but this method is obviously highly platform dependent and builds on a very complex basis.

### 3 Existing architectures

Various attempts to tackle the mentioned problems have been made by prominent in-use plugin architectures.

#### 3.1 QT Plugins

The popular QT C++ library[5] by Trolltech (Nokia) provides platform-independent mechanisms for the runtime loading of plugins to enable the specialization of library behavior and for extending a QT user application.

The interconnection between application and plugins is (only) supported through predefined C++ interfaces[6], which can be specialized by plugins. Necessary runtime types are enabled through a *Meta-Object System*[7] that basically builds upon

a `QObject` base class. Furthermore a *Meta-Object Compiler* as a source-code pre-processor is used to support the architecture.

Though an industry-quality, solid library, the above mentioned methods are very intrusive but settle on a middle-ground between flexibility and complexity regarding plugins.

## 3.2 Adobe Photoshop Plug In Modules

Filter Plugins in Photoshop fulfill a similar purpose like the targeted plugins as they both process image data.

The interface is one pure C method through which all calls, requests and parameters are multiplexed:

```
1  DLLExport MACPASCAL void PluginMain ( const short selector ,
2                                          void *filterParamBlock ,
3                                          long *data ,
4                                          short *result)
```

All command codes and data structures are defined in host application side versioned headers. This shows a solution where all the real interface parts were moved into the runtime and the actual interface is implicit in the code parsing parameters, casting to structures and manipulating those.

This ensures one stable static interface but a highly opaque dynamic interface, where all the versioning is performed during runtime. This approach enables widespread upward- and downward compatibility with flexible behavior, at the drawback of the necessity to multiplex the real interface.

As a note it is worth mentioning that Photoshop supports scripting Application Actions via the Microsoft COM interface.

## 4 Summary

A first plunge has been taken into the complexity of interconnecting plugins at a C++ shared library level without falling back to existing complex component technologies. It was observed that the runtime boundary for C++ applications is much harder to take than source code cooperation due to the *absence of reflection* technology, the *standard scope* that focusses on the source level and the *delegation of module and library concepts* to operating system and compiler vendors.

Though all of those points are realizable, great efforts for general solutions are required and have been taken by eg. Microsoft with COM. Also, many of those points are already covered in more recent languages. The challenge to be faced here is to find a reasonable and realizable midway.

## References

- [1] <http://www.microsoft.com/com/default.aspx>.
- [2] [http://www.boost.org/doc/libs/1\\_38\\_0/libs/gil/doc/index.html](http://www.boost.org/doc/libs/1_38_0/libs/gil/doc/index.html).
- [3] <http://msdn.microsoft.com/en-us/library/ms235460.aspx>.
- [4] <http://blogs.msdn.com/oldnewthing/archive/2006/09/15/755966.aspx>.
- [5] <http://www.qtsoftware.com>.

- [6] <http://doc.trolltech.com/4.5/plugins-howto.html>.
- [7] <http://doc.trolltech.com/4.5/metaobjects.html>.
- [8] Mark DeLoura. *Game Programming Gems 1*. Charles River Media, 2000.