ImageIO: Design of an Extensible Image Input/Output Library

by *Parag Chandra* and *Dr. Luis Ibanez, Ph.D*.

Introduction

Computer graphics and image processing techniques have had useful applications in fields as diverse as computer vision, geographic information systems, and medical imaging. These applications provide researchers with new ways to visualize their data and gain insight into its structure. Their design also poses sizeable challenges to software engineers. One such challenge concerns file input and output: How can these software systems be designed so that they will work with data stored in a variety of different file formats stored on disk?

At first, the solution may seem easy enough: decide on a few formats to support, and hard-code a few routines to handle those formats. This approach may be valid for applications that are intended for use by a single team working on a very specific problem. However, these development scenarios arise infrequently. In the research community, where teams are scattered across different institutions around the globe, each team often has a vested interest in its own file format(s). Even if all of these teams could decide on a common file format to use, they would lose the ability to use their existing datasets with this new code base, unless they wrote conversion routines.

All of these factors render this simplistic approach inappropriate. Consequently, it is necessary to develop a framework for file input and output. We have developed one such framework, as part of Insight: The Visible Human Image Processing Toolkit, a project funded by the National Library of Medicine. The design of our file input and output framework and the details of our C++ implementation are presented in this article.

Design Framework

A good design for an image input/output (image I/O) library should strive to meet the following criteria:

Portability - The library should operate on multiple computer architectures. For our work, this meant supporting at least Windows, Linux, and SunOS. All of the driving applications our group develops are written in C++, and hence the library was also written in C++. There are good, standards-compliant compilers available for most platforms, and because image I/O is a high level function that does not make use of any system-specific routines, the library is platform-independent.

Modularity - The library should be self-contained and decoupled from the other parts of the application, so that it can be reused in other software systems. The reader should note that, for the purposes of this discussion, images exist in two forms: externally as files on disk, and internally as data structures in memory. By making this library responsible only for loading and saving images to and from disk, I/O functionality is completely decoupled from any application-specific notion of an image. This division of responsibility mandates that an intermediary class be used to convert between an image as stored in the I/O class, and the image data structure in memory that will ultimately be used by the application

Extensibility - Support for new file types can be added by third parties without modifying existing code. It is not possible to anticipate what additional image formats will arise in the future, nor is it feasible for one party to provide support for all existing formats. Therefore, it is essential that the image I/O framework be "open to extension, but closed to modification" [5] . This goal is met through the use of a compound design pattern known as the pluggable object factory, and by defining a standard interface through an abstract superclass. An object factory provides a mechanism for easily exchanging one class (or family of classes) for another, provided that the two are derived from the same superclass (parent). The pluggable object factory extends this notion by allowing new subclasses to be added, or "plugged in", to the framework at runtime.

Transparency - Ideally, application programmers should be able to pass a filename to an appropriate class and then access the image data stored in the file, without having to worry about the details of a particular file format. Transparency is achieved through two mechanisms. First, features that are common to all image formats are "factored out" and made members of an abstract superclass, from which format-specific subclasses are derived. In this way, an application developer can work with any supported image through a common interface. Secondly, the pluggable object factory can be used to automatically instantiate (create) the correct subclass that handles a given filetype. The details of this mechanism are given below.

There are two key concepts the reader should keep in mind before proceeding. First, we have decoupled (separated) the notion of an image on disk from that of an image in memory. Since applications must work with images in memory, we use an intermediary class, FilterIOToImage, to convert between the two representations. Second, we have used a software design pattern known as the pluggable object factory to dynamically instantiate the appropriate subclass that handles a given file format. Whenever support for a new file format is needed, a subclass of ImageIO is derived to handle the new format's implementation details. Additionally, a factory that can produce instances of this new subclass is also created and is registered with a master dictionary. This master dictionary can then be used to select, at runtime, the appropriate ImageIO subclass to instantiate for a particular filename. Next we present the details of the class hierarchy used to implement this framework.

Class Structure

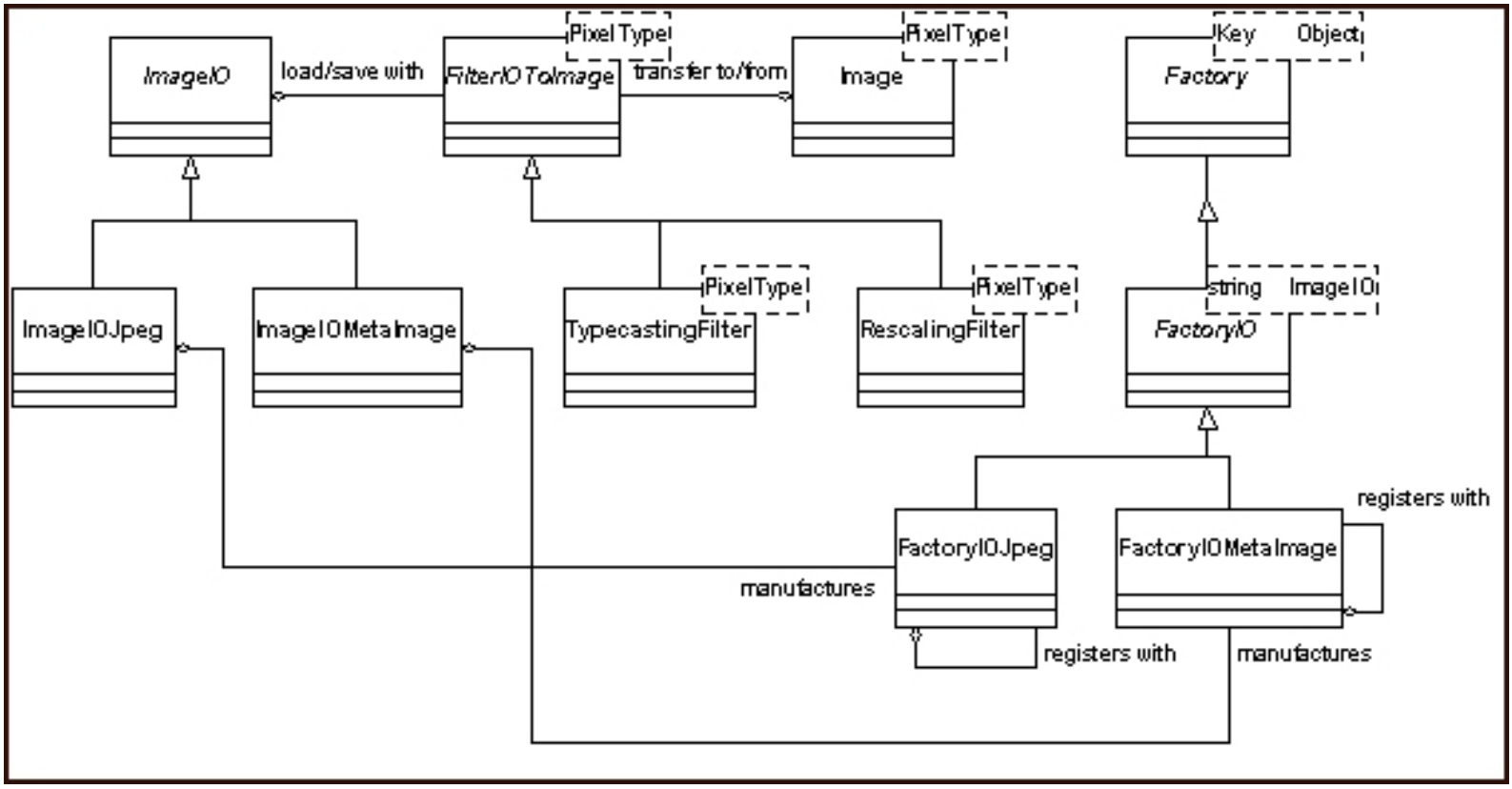

Figure 1. Class Hierarchy

In the above diagram, each class is shown as a box. Abstract class names are italicized, while concrete subclass names are not. Connections terminating with arrowheads indicate inheritance ("is a") relationships, while those ending in diamonds indicate aggregation ("has a") relationships.

ImageIO is the heart of the library. This abstract class handles images stored on disk, and defines the interface that application programmers can use to interact with the library. It provides methods for loading and saving images, as well as for reading any pertinent header information (image dimensions, color depth, etc.) about a file. Notice that in addition to default Load() and Save() methods, we define specialized versions of these methods that allow us, for example, to use 2D image formats to assemble 3D volumes, and to extract a single slice or a range of slices from a 3D image.

```
class ImageIO {
public:

     virtual void Load() = 0;
     virtual void Save() = 0;
     virtual void Save2DSlice(int sliceNumber) = 0;
     virtual void Assemble3DVolume() = 0;
     /* This method returns the file extensions corresponding to the
     filetypes that this class can handle */
     virtual string GetSupportedFileExtensions() const = 0;

private:
```

```
     void* imageData;

};

class ImageIOJpeg {
public:

     string GetSupportedFileExtensions() const { return ".jpg"; }

};
```

The member imageData is defined as a pointer to void so that it can store image data of any type. The Load and Save methods typecast this pointer to the appropriate data type. For example, ImageIOJpeg treats the data as unsigned chars, because that is the data type of grayscale JPEG images.

Factory<Key, Object> is the pluggable object factory base class. The pluggable object factory [2] is essentially a composition of several design patterns described in [4]: the Abstract Factory, which provides a mechanism for instantiating classes from a hierarchy; the Singleton, which ensures that only one instance of a particular class exists and that it is globally accessible; and the Prototype, which we use to specify the abstract pluggable object factory from which other, concrete factories are derived. Factory maintains a dictionary, which maps keys (words) to instances of objects (definitions). It is parameterized (templated) over the key type and the object type.

```
template <class Key, class Object>
class Factory<Key, Object> {
public:

     static Object* Create (const Key k) {

          Factory<Key, Object>* f = (dictionary->find(k))->second;
          if (!f) return NULL
          else return f->MakeObject();

     }

protected:

     Factory(const Key k) {

          static bool dictionaryInitialized = false;
          if (!dictionaryInitialized) {

               dictionary = new Dictionary;
```

```
            dictionaryInitialized = true;

        }
        dictionary->insert(std::make_pair(k, this);

    }

    virtual Object* MakeObject() const {
    /* implemented by subclasses. return instance of object that
    subclass is responsible for */

    }

private:

    typedef Factory<Object>* FactoryPtr;
    typedef std::map<Key, FactoryPtr> Dictionary;
    static Dictionary* dictionary;

};
```

The public member Create() is the only method visible to the application programmer. Its only parameter is a key, which is used to index into the internal dictionary. If the key is found in the dictionary, the corresponding factory creates an instance of the object. The dictionary is a static member variable. Consequently, a single instance of the dictionary is shared amongst all concrete instances of the object factory.

The constructor for the abstract factory takes a key as a parameter, and inserts into the dictionary, a mapping between that key and the subclass (concrete factory) that invoked this constructor (via `this`, which C++ provides to refer to oneself). We shall refer to the process of concrete factories becoming listed in the master dictionary as *registration*. Note the use of the static boolean variable, `dictionaryInitialized`, inside the constructor. This variable is used as a flag to ensure that the dictionary is initialized exactly once, just before the first concrete factory attempts to register itself. It is necessary because the order of static initializations across different compiled source files varies between platforms and compilers, and hence cannot be known in advance. Therefore, it is necessary to have this simple check in place to ensure that the dictionary is always initialized before any registration attempts are made.

We then derive **FactoryIO<string, ImageIO>** from the **Factory** class, taking advantage of the fact that each file format will have a unique file extension associated with it (e.g. .bmp, .jpg, .png, etc.), and hence can be used as a key value. From the **FactoryIO** class we in turn derive a separate concrete factory for each file format, and each of these factories is responsible solely for creating instances of its corresponding ImageIO class. Below is an example taken from our code for loading JPEG images.

```
class FactoryIOJpeg : public FactoryIO {
public:

     FactoryIOJpeg() : FactoryIO(myObject.GetSupportedFileExtensions())
     {}
     static const ImageIOJpeg myObject;
     ImageIO* MakeObject() const { return new ImageIOJpeg; }
     static const FactoryIOJpeg registerMyself;

};
```

There are two key points to note in the implementation of this concrete factory. First, each concrete factory contains a static instance of itself, used solely for the purposes of registering the factory with the dictionary. Secondly, each concrete factory also has a static instance of the ImageIO object it is responsible for cloning. This is used simply for determining the key values to use in the registration process.

Image<PixelType> is the class we use to maintain an image in memory. Its design and implementation details are not important for this discussion. The reader should simply note that the class is parameterized on the datatype of the pixels with which the application needs to work. This is not necessarily the same as the datatype of the image as stored on disk.

The final component of this framework is **FilterIOToImage<PixelType>**. Recall that because we make the distinction between images residing on disk and those in memory, and because we want the two classes to know nothing of each other, we must use an intermediary filter class to translate image data between the two. This class performs exactly that function.

```
class FilterIOToImage<PixelType> {
public:

     FilterIOToImage(const char* fileName, Image<PixelType>* image);
     FilterIOToImage(Image<PixelType>* image, const char* fileName);
     virtual void CopyPixelsToIO();
     virtual void CopyPixelsToImage;

protected:

     ImageIO* myIO;
     Image<PixelType>* myImage;
     static const Factory<ImageIO> myIOFactory;

};
```

The **FilterIOToImage** class contains a reference to an ImageIO object and a reference to an Image object. Notice that FilterIOToImage and Image are both parameterized on PixelType. This is so that the former has knowledge of the latter's template parameter. As the names suggest, CopyPixelsToIO() and CopyPixelsToImage() translate image data from one representation to the other. By deriving from FilterIOToImage, this translation can be anything from simple typecasting or rescaling to more complex operations such as grayscale conversion. Notice that we have also defined two constructors that take in filenames and images as parameters. The first constructor loads the image and transfers the pixels *to* the image passed as a parameter, while the second constructor transfers the pixels *from* the image and then saves it to disk. Both of these constructors instantiate the appropriate ImageIO object by parsing the file extension from `fileName`, and then proceed to either load or save the image using this object. By defining these constructors, we are able to present a greatly simplified interface to the library that is appropriate for the majority of image input/output tasks (see example below). However, the default constructor (not shown) is also available to developers so they can still have full control over the I/O process.

Implementation Details

In many cases, subclassing ImageIO involves interfacing with a third-party library written in C. This will be the case for commonly used image formats, as good low-level C libraries exist for nearly all of them, and there is little sense in rewriting these routines. Whenever possible, we have leveraged this existing code by writing our ImageIO classes to effectively wrap the interface provided by these third-party libraries. By using the `extern "C"` directive, we are able to mix C and C++ code, and can therefore compile the third party library along with our C++ classes into new libraries suitable for use in this framework.

It is easiest to compile new libraries statically with compile time linking. However, we would like to advocate the use of dynamic link libraries instead. Dynamic link libraries (DLLs) are written and compiled to be loaded at runtime. Consequently, they can be loaded and unloaded on demand. Furthermore, if more than one program requests a given library, then a single copy of the library is loaded in memory once and the multiple programs share this single copy of the library resulting in a smaller memory footprint for these applications. It is possible to define a high-level class that abstracts away the platform-specific details of loading and importing the symbols of dynamic link libraries. With this class in place, it becomes possible to place new libraries supporting additional filetypes into a single directory that is scanned at program startup. This small change removes the necessity of re-linking existing applications when adding support for new image types. Furthermore, if an unsupported filetype is encountered, rather than returning a `NULL` pointer to indicate this condition, we can scan an additional library path at runtime to see if a DLL supporting this filetype does in fact exist. If it does, the DLL can be imported and the object factory for that image type will be dynamically registered with the dictionary allowing the application to repeat its request for an object to handle this filetype.

Analysis

The following sample program illustrates the simplified interface presented to the application programmer:

```
void main () {

     FilterIOToImage<float>* loadFilter, saveFilter;
     Image<float>* image = NULL;
     loadFilter = new FilterIOToImage<float>("input.jpg", image);
     // Start processing image
     ...
     // Done processing image. Save result
     saveFilter = new FilterIOToImage<float>(image, "output.bmp");

}
```

This first example simply loads an image for processing and then saves it back to disk in a different file format. Because both jpeg and bmp files are inherently 2-D, the default load and save methods are adequate for the image being processed, and hence the library's simplified interface can be used. Loading and saving the image is very intuitive and requires only a single statement for each operation.

This next example illustrates the standard interface available to the developer when more control is needed:

```
void main () {

     FilterIOToImage<float>* loadFilter, saveFilter;
     Image<float>* image;
     const FactoryIO ioFactory;
     ImageIO* loadIO, saveIO;
     char inputFileName[] = "inputSlice*.jpg";
     char outputFileName[] = "outputSlice.bmp";

     loadIO = ioFactory.Create(ParseFileExtension(inputFileName));
     loadIO->SetFileName(inputFileName);
     loadIO->Assemble3DVolume();
     loadFilter = new FilterIOToImage<float>(loadIO, image);
     loadFilter->CopyPixelsToImage();
     // Start processing image
     ...
     // Done processing image. Save result
     saveIO = ioFactory.Create(ParseFileExtension(outputFileName));
     saveIO->SetFileName(outputFileName);
     saveFilter = new FilterIOToImage<float>(saveIO, image);
     saveFilter->CopyPixelsToIO();
     saveIO->Save2DSlice(3);
```

```
}
```

In this case, the program first assembles a 3-D volume from a series of sequentially numbered 2-D slices stored as jpeg images. After processing, a single slice of that volume is then saved to disk as a bmp file. Although a few extra lines of code are required to instantiate the filters and to transfer the image data between the two representations, the underlying framework is still reasonably transparent and developer friendly.

One drawback to this framework concerns efficiency. ImageIO often relies on a third-party library to handle the details of a particular file format, and such a library will likely maintain an internal buffer to store the image. ImageIO will in turn then copy this buffer into its own `imageData` buffer. Finally, FilterIOToImage will copy and translate the pixels over to an Image object. Three copies of the same data are used before it is finally ready for use by the application. This can become a problem when dealing with very large images. If the application is forced to page to disk for memory, then I/O operations can become slow. Fortunately, this penalty is offset because I/O is a relatively infrequent operation; usually, an image is loaded into a program at startup, and then saved to disk once processing is finished. Furthermore, we can eliminate one of these copies if we modify the design of the ImageIO class to use an external void pointer as a buffer, instead of using its own internal one.

Another caveat regards to byte ordering: the so-called "little-endian vs. big-endian" issue [1]. Some computer architectures treat the leftmost bit of a byte (or byte of a word) as the most significant, while others treat the rightmost bit (or byte) as such. Therefore, it is necessary to perform a conversion between the ordering used in the file and the ordering used in the computer's memory. Failure to do so will result in images that only appear correctly on one type of machine. We have left this issue to be resolved by the third-party image libraries that our ImageIO framework interfaces with, and experience has shown us that, at least for the image types we have encountered, such libraries perform the conversion appropriately.

In summary, the advantages this framework provides in the form of extensibility and ease-of-maintenance more than compensate for the limitations described above. Support for new image types can be added by third parties, and even integrated into applications during execution by dynamic linking. I/O classes are kept completely separate from the rest of the application, so that they can be reused in other programs. The library is written in completely portable C++ code, so it runs on both Windows and several variants of Unix with only a recompilation required. Finally, the interface we have defined in the base ImageIO class provides added flexibility by enabling 2D image formats to be used in a 3D sense, and vice versa.

References

1 Cohen, Danny. *On Holy Wars and a Plea for Peace*. http://www.goodnet.com/~gooch/docs/documents/endian.html

2 Culp, Timothy R. *Industrial Strength Pluggable Object Factories*. C++ Report Online. http://www.

creport.com/html/from_pages/view_recent_articles_c.cfm?ArticleID=1520

3

Deadman, Richard. *Pluggability Patterns*. Deadman Consulting. http://24.114.65.197/articles/pluggableArticle.html

4

Gamma, Erich et. al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Massachusetts, 1995.

5

Meyer, Bertrand. *Object-Oriented Software Construction, Second Edition*. Prentice Hall, California, 1997.

6

Roberts, Don and Johnson, R *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks.* University of Illinois Computer Science Dept. http://st-www.cs.uiuc.edu/users/droberts/evolve.html

---

This work was supported by NLM contract N01-LM-0-3501 as part of Insight: The Visible Human Image Processing Toolkit development effort.

Biography

Parag Chandra (chandra@cs.unc.edu) is a graduate student/research assistant at The University of North Carolina, where he is pursuing his master's degree in computer science and assisting with research in medical image registration. A recent graduate of Georgia Tech, he has had 6 years of software engineering experience in industry and academia. Currently he is on track to graduate in the spring of 2001.

Dr. Luis Ibanez (ibanez@cs.unc.edu) received a degree in Physics form the Universidad Industrial de Santander, Colombia in 1989, and a Ph.D. degree in Signal Processing from the Universite de Rennes I, France in 2000. He is currently a research assistant professor at The University of North Carolina. His interests include medical image processing, image guided surgery and biological morphogenesis.
