Proceedings of the 5th WSEAS/IASME Int. Conf. on SYSTEMS THEORY and SCIENTIFIC COMPUTATION, Malta, September 15-17, 2005 (pp310-318)

A Programming Method for Building Component-Based Commercial Software for Image Processing

FERIEL BENHISSI*, SALEM NASRI**
*Faculté des Sciences de Monastir – Monastir - TUNISIA
**Ecole Nationale des Ingénieurs de Monastir – Monastir – TUNISIA
Feriel_Benhissi@yahoo.fr- salem.nasri@enim.rnu.tn

Abstract: - This paper discusses a consistent method for building component-based commercial software for image processing. The resulting framework greatly facilitates the incorporation into the software of any processing function without worrying about the memory storage and generalizing its usage to any image format while focusing on the algorithmic behavior of the function. Also it allows the software to be expandable, maintainable, understandable and stable. It implements handles and rep objects, and use templates so a single version of the code will work with any type of pixel. Thanks to the handles, we are able to manage the number of copies of a given image and so avoid memory leakage and fragmentation. The method deals separately with the issues related to image processing problems such as memory management and memory alignment. We also present a practical implementation of the method in which we demonstrate how to apply C++ to solve the problems inherent in building commercial software for image processing. The method can be easily applied to any object oriented language.

Keywords: - commercial software, component-based, interfaces, object oriented language, image processing

1 Introduction

Component-based [6] software is an emergent discipline that is generating tremendous interest due to the development of plug-and-play reusable software [1]. Under this new setting, constructing commercial software now involves the use of prefabricated pieces, perhaps developed at different times, by different people unaware of each other, and possibly with different uses in mind. The ultimate goal is to be able to have software with the following characteristics: expandable, maintainable, understandable, and stable [8].

The expandability can be defined as the ability to add new features quickly and extend existing features in commercial software. Put simply, we have to think ahead when designing and writing code. Just implementing to a specification, with no thought to future expansion or maintainability, makes future changes much more difficult. A further complication is that most software won't allow features to be deprecated once the product is in use.

The goal of maintainability is to ensure through rigorous testing and preparation that any post-release bugs have minor consequences to the product. What becomes crucial is that these bugs be easily and quickly corrected for the next release.

Commercial software often includes visible software interfaces. If we are building an embedded system library, it needs to be understandable to the other members involved in the application construction. If we are building a software library, the interface we present must be easily understood by the customers so that they can use it to build their own applications. This doesn't just mean that naming conventions must make sense, but more importantly that our design and our use of language elements, like templates, are clear and appropriate.

Finally, commercial software must be stable; that is, it must be able to run for extended periods of time without crashing, leaking memory, or having unexplained anomalies.

On the other hand, many new algorithms in image processing have emerged in the last decade due to the great development in the domain of applied mathematics especially in the applied statistics field [7]. The inclusion of these algorithms in given software is sometimes very difficult not because they are still in the experimental stage, but because their code is not written in an efficient way to allow them to be easily embedded with other algorithms in the same software. In fact the code of the algorithm and that of memory access are usually written in the same block, and the resulting code is optimized in function of the number of arithmetic operations performed leaving the other issues such memory management and memory alignment as well as the management of image copies partially unresolved. These issues are generally fully resolved when it comes to the hardware implementation of the algorithm, which is quite difficult to maintain and expand. Also, the

algorithm generally applies to a restrained set of image formats, because it is presented in the context of a specific field of image processing such as medical image or computer vision in which only some well-known formats of image are available.
However thanks to the emergence of component-based software along with many new object oriented languages we are able to deal with all the issues mentioned above separately, and to incorporate them consistently in optimized software.
Our work is about applying C++ to solve the problems inherent in building commercial software for image processing. Its aim is to discuss a consistent method for building commercial software for image processing based on the implementation of components and interfaces. The resulting framework greatly facilitates the incorporation into the software of any processing function without worrying about the memory storage and generalizing its usage to any image format while focusing on the algorithmic behavior of the function. As a first step, we define the requirements of a memory allocation object and we describe its class hierarchy. Next, we present the design of the components of our image framework and its class hierarchy. In these components, there is a separation of the image storage classes from the image processing functions, and the templates [12] are used so that we will produce a single version of code that works with any type of pixel. We will also address the implementation of handles and rep objects as memory alignment. Thanks to the handles, we will be able to manage the number of copies of a given image and so avoid memory leakage and fragmentation. As to the memory alignment, it boosts the performance of the software because many image processing algorithms can be optimized for particular memory alignments. We will also tackle the locking with exception throwing, necessary for a multithread functioning.
Software for image processing won't be useful if it doesn't support any of the popular formats of image storage. We have designed a simple file delegate interface so new file formats can be added with little difficulty. As an example, we have extended this interface to the JPEG format which is the most popular. But the idea is easily applicable to any image format. After having discussed the essentials of the components that compose the image software, all that remains is to add global processing functions. We concentrate on the processing of single image. The principle can be easily extended to the case of two images. In the final section, we present the implementation of image processing software. In the where an image sharpening and texture enhancement of JPEG color images are performed.

2 Memory Allocation Object

Images require a great deal of memory storage to hold the pixel data. It is very inefficient to copy these images, in terms of both memory storage and time, as the images are manipulated and processed. We can easily run out of memory if there are a large number of images. In addition, the heap could become fragmented if there isn't a large enough block of memory left after all of the allocations.
Instead of designing an object that works only for images, we create a generic object that is useful for any application requiring allocation and management of heap memory. Here's the list of requirements for our generic memory allocation object:

- Allocates memory off the heap, while also allowing custom memory allocators to be defined for allocating memory from other places, such as private memory heaps.
- Uses reference counting to share and automatically delete memory when it is no longer needed.
- Employs locking and unlocking as a way of managing objects in multithreaded applications.
- Has very low overhead. For example, no memory initialization is done after allocation. This is left to the user to do, if needed.
- Uses templates so that the unit of allocation is arbitrary.
- Supports simple arrays, `[]`, as well as direct access to memory.
- Throws Standard Template Library (STL) exceptions when invalid attempts are made to access memory.
- Aligns the beginning of memory to a specified boundary.

2.1 Memory Allocator Object's Class Hierarchy

The class hierarchy for the memory allocator is shown in Fig. 1.

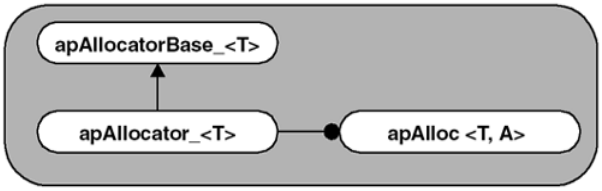

Fig. 1 Class hierarchy of the memory allocator.

It consists of a base class, a derived class, and then the object class, which uses the derived class as

one of its arguments. All three classes use templates.

The apAllocatorBase_<> base class contains the raw pointers and methods to access memory. It provides both access to the raw storage pointer, and access to the reference count pointing to shared storage, while also defining a reference counting mechanism. apAllocatorBase_<> takes a single template parameter that specifies the unit of memory to be allocated. The full base class definition is shown here.

```
template<class T> class apAllocatorBase_
{
public:
   apAllocatorBase_ (unsigned int n, unsigned int align)
   : pRaw_ (0), pData_ (0), ref_ (0), size_ (n), align_
  (align){}
   virtual ~apAllocatorBase_ () {}
   operator      T* ()      { return pData_;}
   operator const T* () const { return pData_;}
   unsigned int size  () const { return size_;}
   unsigned int ref   () const { return ref_;}
   unsigned int align () const { return align_;}
   void addRef () { ref_++; }
   void subRef ()
   {
      --ref_;
      if (ref_ == 0) delete this;
   }
protected:
   virtual void allocate   () = 0;
   virtual void deallocate () = 0;
   T* alignPointer (void* raw);
   apAllocatorBase_   (const apAllocatorBase_& src);
   apAllocatorBase_& operator= (const
   apAllocatorBase_& src);
   char*       pRaw_;
   T*          pData_;
   unsigned int size_;
   unsigned int ref_;
   unsigned int align_;
};
```

Memory alignment is important because some applications might want more control over the pointer returned after memory is allocated. Most applications prefer to leave memory alignment to the compiler, letting it return whatever address it wants. We provide memory alignment capability in apAllocatorBase_<> so that derived classes can allocate memory on a specific boundary. On some platforms, this technique can be used to optimize performance. This is especially useful for imaging applications, because image processing algorithms can be optimized for particular memory alignments.

The apAllocator_<> class, which is derived from apAllocatorBase_<>, handles heap-based allocation and deallocation. Its definition is shown here.

```
template<class T> class apAllocator_ : public
apAllocatorBase_<T>
{
public:
   explicit apAllocator_ (unsigned int n, unsigned int align
   = 0) : apAllocatorBase_<T> (n, align)
   {
      allocate ();
      addRef ();
   }
  virtual ~apAllocator_ () { deallocate();}
private:
   virtual void allocate () ;
  virtual void deallocate ();
  apAllocator_              (const apAllocator_& src);
  apAllocator_& operator= (const apAllocator_& src);
};
```

The apAllocator_<> constructor handles memory allocation, memory alignment, and setting the initial reference count value. The destructor deletes the memory when the object is destroyed.

apAlloc<> is our memory allocation object. This is the object that applications will use directly to allocate and manage memory. The definition is shown here.

```
template<class T, class A = apAllocator_<T> >
class apAlloc
{
public:
  static apAlloc& gNull ();
  apAlloc  ();
  explicit apAlloc  (unsigned int size, unsigned int
  align=0);
  ~apAlloc ();
  apAlloc           (const apAlloc& src);
  apAlloc& operator= (const apAlloc& src);
  unsigned int  size   () const { return pMem_->size ();}
  unsigned int  ref    () const { return pMem_->ref ();}
  bool          isNull () const { return (pMem_ ==
  gNull().pMem_);}
  const T* data () const { return *pMem_;}
  T*       data ()       { return *pMem_;}
  const T& operator[] (unsigned int index) const;
  T&       operator[] (unsigned int index);
  virtual A* clone ();
  void duplicate ();
protected:
       A*   pMem_;
      static apAlloc* sNull_;
};
```

Parameter T specifies the unit of allocation. Parameter A specifies how and where memory is allocated. It refers to another template object whose job is to allocate and delete memory, manage reference counting, and allow access to the underlying data.

The null allocation, an allocation with no specified size, is of special interest because of how we

implement it. We saw that the apAllocator_<> object supported null allocations by allocating one element. It is possible that many (even hundreds) of null apAlloc<> objects may be in existence. This wastes heap memory and causes heap fragmentation.

Our solution is to only ever have a single null object for each apAlloc<> instance. We do this in a manner similar to constructing a Singleton object [4]. Singleton objects are typically used to create only a single instance of a given class. We use a pointer, sNull_, and a gNull() method to accomplish this:

```
template<class T, class A>
apAlloc<T,A>* apAlloc<T, A>::sNull_ = 0;
```

This statement creates our sNull_ pointer and sets it to null. The only way to access this pointer is through the gNull() method, whose implementation is shown here.

```
template<class T, class A>
apAlloc<T,A>& apAlloc<T, A>::gNull ()
{
  if (!sNull_)
    sNull_ = new apAlloc (0);
  return *sNull_;
}
```

3 Implementation considerations

Here we present the design for the image framework. The components of the framework are shown in Fig. 2.

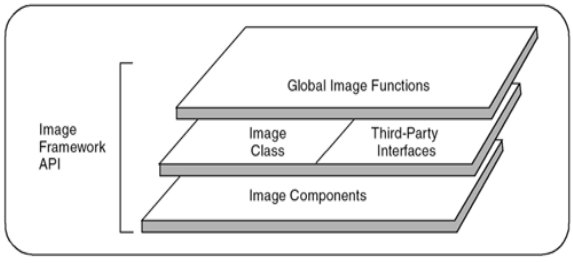

Fig. 2 Image Framework Components

They follow the following principles:

- Image storage should be separate from the image processing functions. The image storage classes are independent of the image processing functions (but not vice versa).
- Templates should be used for a more efficient design, allowing us to: produce a single version of code that works with any pixel type; optimize performance where needed by using specialization; and adapt our image storage component to use other memory allocators.

The details that we now address in the design include:

- **Handles and rep objects**. Our final image storage object encapsulates an apAlloc<> object along with other storage parameters. Because we aren't using handles, these storage objects get copied as they are passed. Fortunately, the copy constructor and assignment operators are very fast, so performance is not an issue because the pixel data itself is reference counted. The complexity of the additional layer of abstraction didn't provide enough of a benefit to make it into the final design.
- **Memory alignment**. apAlloc<> supports the alignment of memory on a user-specified pixel boundary. Proper alignment can be critical for efficient performance of many image processing functions. As it turns out, it is not sufficient to align the first pixel in the image as apAlloc<> does. Most image processing routines process one line at a time. By forcing the first pixel in each line to have a certain alignment, many operations become more efficient. For generic algorithms, this savings can be modest or small because the compiler may not be able to take advantage of the alignment. However, specially tuned functions can be written to take advantage of particular memory alignments. Many third-party libraries contain carefully written assembly language routines that can yield impressive savings on aligned data. Our final design has been extended to better address memory alignment.
- **Image shape**. We refer to the graphical properties of the storage as *image shape*. For example, almost all images used by image processing packages are rectangular; that is, they describe pixels that are stored in a series of rows. In our final design, we explicitly support rectangular images so that we can optimize the storage of such images, but we also allow the future implementation of non-rectangular images. For example, we might have valid image information for a large, circular region. If we store this information as a rectangle, many bytes are wasted because we have to allocate space for pixels that do not contain any useful information. A more memory-efficient method for storing non-rectangular pixel data is to use *run-length encoding*.

3.1 Class hierarchy

According to the previous section, the design partitions image storage into three pieces, as illustrated in Fig. 3.

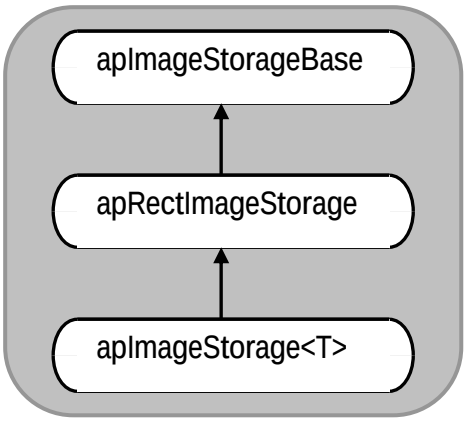

Fig. 3 Class hierarchy of the image framework

We start by looking at apImageStorageBase. This base class only has an understanding of the boundary surrounding the image storage.

```
class apImageStorageBase
{
public:
  apImageStorageBase ();
  apImageStorageBase (const apRect& boundary);
  virtual ~apImageStorageBase ();
  const apRect& boundary () const { return boundary_;}
  int  x0            () const { return boundary_.x0();}
  int  y0            () const { return boundary_.y0();}
  int  x1            () const { return boundary_.x1();}
  int  y1            () const { return boundary_.y1();}
  unsigned int width  () const { return boundary_.width();}
  unsigned int height () const { return boundary_.height();}
protected:
  apRect boundary_;
};
```

Once the rectangular boundary is specified in the constructor, the object is immutable and cannot be changed. It is designed this way because changing the boundary coordinate information would affect how this object interacts with other images that are already defined.

apRectImageStorage is the most complicated object in our hierarchy. It handles all aspects of memory management, including allocation, locking, and windowing. Here are the protected member data of apRectImageStorage:

```
protected:
  mutable apLock lock_;
  apAlloc<Pel8>  storage_;
  Pel8*          begin_;
  Pel8*          end_;
  eAlignment     align_;
  unsigned int   yoffset_;
  unsigned int   xoffset_;
  unsigned int   bytesPerPixel_;
  unsigned int   rowSpacing_;
```

storage_ contains the actual pixel storage as an array of bytes. apAlloc<> allows a number of objects to share the same storage, but the storage itself is fixed in memory. This allows us to create *image window*s. An image window is an image that reuses the storage of another image. In other words, we can have multiple apRectImageStorage objects that use identical storage, but possibly only a portion of it. To improve the efficiency of accessing pixels in the image, the object maintains begin_ and end_ to point to the first pixel used by the object and just past the end, respectively. Derived objects use these pointers to construct iterator objects, similar to how the standard C++ library uses them. bytesPerPixel_ and align_ store the pixel size and alignment information passed during object construction. rowSpacing_ contains the number of bytes from one row to the next. This is often different than the width of the image because of alignment issues. By adding rowSpacing_ to any pixel pointer, you can quickly advance to the same pixel in the next row of the image.

xoffset_ and yoffset_ are necessary for image windows. Just because two images share the same storage_ does not mean they access the same pixels. Image windowing lets an image contain a rectangular portion of another image. xoffset_ and yoffset_ are the pixel offsets from the first pixel in storage_ to the first pixel in the image. If there is no image window, both of these offsets are zero. lock_ handles synchronization to the rest of the image storage variables, with the exception of storage_ (because it uses apAlloc<>, which has its own independent locking mechanism).

We also use a number of locking functions to synchronize access to both the image storage parameters and the image storage itself, as shown.

```
  bool lockState     () const { return lock_.lock();}
  bool unlockState   () const { return lock_.unlock();}
  bool lockStorage   () const { return storage_.lockStorage ();}
  bool unlockStorage () const { return storage_.unlockStorage
();}
  bool lock   () const { return lockState() && lockStorage();}
  bool unlock () const { return unlockState() &&
unlockStorage();}
```

Locking is not a difficult feature to add to an object, but it is important to consider where to use it effectively. In our design, for example, several instances of apRectImageStorage can use the same underlying pixel storage. There is no need to lock access to this storage if we are only manipulating other member variables of apRectImageStorage. lockState() is best used when the state of

apRectImageStorage changes. lockStorage() is used when the actual pixel data is accessed. lock() is a combination of the two, and is useful when all aspects of the image storage are affected. These functions are used by derived objects and non-member functions, since locking is a highly application-specific issue.
apImageStorage<T>, however, is a template class that defines image storage for a particular data type. Most apImageStorage<> methods act as wrapper functions by calling methods inside apRectImageStorage and applying a cast.

3.1.1 Exception-Safe Locking

Most functions that operate on apImageStorage<> objects require some form of record locking. This is true for functions that modify both the state of the object and the underlying pixels. Writing a function that calls lock() and unlock() is not difficult, but we need to consider how exceptions influence the design; otherwise, it is quite possible that when an exception is thrown, the lock will not be cleared because the function does not terminate properly. One solution is to add a try block to each routine to catch all errors, so that the object can be unlocked before the exception is re-thrown [10].

```
template<class T> class apImageStorageLocker
{
public:
  apImageStorageLocker (const apImageStorage<T>& image)
: image_ (image) { image_.lock();}
  ~apImageStorageLocker () { image_.unlock();}
private:
  const apImageStorage<T>& image_;
  // No copy or assignment is allowed
  apImageStorageLocker (const apImageStorageLocker&);
  apImageStorageLocker& operator= (const
apImageStorageLocker&);
};
```

Our apImageStorageLocker<> implementation locks only apImageStorage<> objects, although it wouldn't be hard to create a generic version. Here is how it works. When an apImageStorageLocker<> object is created, a reference to an apImageStorage<> object is stored and the object is locked. When the apImageStorageLocker<> object is destroyed, the lock on apImageStorage<> is released. You can see how powerful this simple technique is when it is used within another function.

3.3 Finalizing Interfaces to Third-Party Software

A decade ago, most software solutions were completely proprietary, in that all aspects of the application were developed in-house. There were plenty of software libraries available for purchase, but they were usually expensive or were considered inferior — not because they didn't perform the intended function, but because they were not developed in-house. This "not invented here" syndrome created large in-house development groups that often duplicated functionality available elsewhere. The actual expense of developing these packages was enormous, especially considering that all maintenance was performed by the organization. Most of these issues vanished due to shrinking budgets and the advent of open-source software.
Modern software takes advantage of existing libraries to speed development and minimize the maintenance issues. It is now considered good design practice to design applications with interfaces that leverage existing code. We use the word delegates to refer to third-party software packages to which we delegate responsibility.

3.3.1 File Delegates

We have created a very flexible and extensible image processing framework. However, it still lacks the capability of interacting with the outside world. Unless our package can import and export images using many of the popular image formats, our framework is of little use.
There are many image storage formats, including JPEG, GIF, PNG, and TIFF. They all have their advantages and disadvantages, so supporting a single format is of limited use. We will design a simple interface so new file formats can be added with little difficulty. This design can be used by most image formats, although it may not take advantage of all the features of an individual format. Fig. 4 provides an overview of the file delegate strategy.

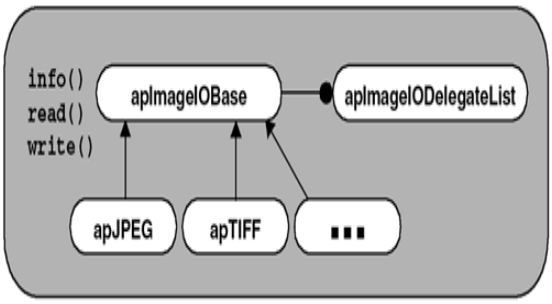

Fig. 4 File Delegate Interface Design

We create a base class, apImageIOBase, that defines the services we want and then we derive one class from apImageIOBase for every file format we want to support. apImageIOBase defines three essential methods, info(), read(), and write(), that check the file format and handle the actual reading and writing of each file format, respectively, as shown.

```
class apImageIOBase
{
public:
  virtual apDelegateInfo info (const std::string& filename) = 0;
  template<class T, class S>
  void read (const std::string& filename, apImage<T,S>&
image);
  template<class T, class S>
  void write (const std::string& filename, apImage<T,S>&
image,const apDelegateParams& params = sNoParams);
protected:
  apImageIOBase ();
  virtual ~apImageIOBase ();
  static apDelegateParams sNoParams;
};
```

info() determines whether a file is of the specified format and, if known, can provide the size. The read() function reads an image into the specified apImage<> object. This is an excellent example of using templates inside a non-template object. The user can specify an image of any arbitrary type, and read() returns an image of that type. Most applications would use info() to determine the image type before using read() to read the image data from a file. write() takes an apImage<> object and saves it in a particular image format.

3.3.2 JPEG File Delegate

One of the most common file formats is Joint Photographic Expert's Group (JPEG). JPEG can store both monochrome and color images at various levels of compression.

apJPEG is our file delegate object that creates an interface between our apImageIOBase interface and the JPEG library. Its definition is shown here.

```
class apJPEG : public apImageIOBase
{
public:
  static apJPEG& gOnly ();
  virtual apDelegateInfo info (const std::string& filename);
  virtual apImage<apRGB> readRGB   (const std::string&
  filename);
  virtual apImage<Pel8>  readPel8  (const std::string&
  filename);
  virtual bool write (const std::string& filename, const
apRectImageStorage& pixels,const apDelegateParams&
params = sNoParams);
private:
  static apJPEG* sOnly_;
  apJPEG ();
  ~apJPEG ();
};
```

3.4 Adding Global Functions

Our apImage<> class does not include the image processing functions. We decided that the image class should only contain the absolute essentials.

3.4.1 Processing Single Source Images

Single source image processing operations take a single source image and produce a single destination image. We provide a general class, apFunction_s1d1, which you can use to easily add your own single source image processing functions.

apFunction_s1d1 lets us logically divide the processing operations, each as a separate method. We have made some of those methods virtual so that we can derive new classes from apFunction_s1d1 to handle custom requirements.

The apFunction_s1d1 class is shown here.

```
template<class R, class T1, class T2,
class S1=apImageStorage<T1>, class
S2=apImageStorage<T2> >
class apFunction_s1d1
{
public:
  apFunction_s1d1 () : function_ (0) {}
  typedef void(*Function) (const R&, const
  apImage<T1,S1>& src1,apImage<T2,S2>& dst1);
  apFunction_s1d1 (Function f) : function_ (f) {}
  virtual ~apFunction_s1d1 () {};
  void run (const apImage<T1,S1>& src1,
  apImage<T2,S2>& dst1) { execute (src1, dst1);}
protected:
  Function      function_; // Our process function, if any
  apImage<T1,S1> roi1_;    // roi of src1 image
  apImage<T2,S2> roi2_;    // roi of dst1 image
  virtual apIntersectRects intersection(const
  apImage<T1,S1>& src1,apImage<T2,S2>& dst1)
  { return intersect (src1.boundary(), dst1.boundary());}
  virtual void execute (const apImage<T1,S1>& src1,
  apImage<T2,S2>& dst1);
  virtual void createDestination (const apImage<T1,S1>&
  src1,apImage<T2,S2>& dst1);
  virtual void process ();
};
```

apFunction_s1d1 has five template parameters. Four of these parameters are present because there are two images, each requiring two parameters. The parameter R is for intermediate computations.

apFunction_s1d1 can be used in two different ways, depending on how you want to specify the actual image processing operations. We can either override process() to define your processing function, or we can pass a function pointer to the constructor. We recommend the latter option because it means that there will be no changes to apFunction_s1d1, and no need to derive objects from it. It also encourages us to write stand-alone image processing operations that potentially have other uses in our application. We pass a function pointer to the constructor, as shown:

```
typedef void(*Function) (const R&, const apImage<T1,S1>&
src1,apImage<T2,S2>& dst1);
apFunction_s1d1 (Function f) : function_ (f) {}
```

The run() method is the main entry point of apFunction_s1d1, but it only calls the virtual function, execute(). The execute() method constructs the intersection and performs the image processing operation. execute() is only overridden if the standard rules for computing the image windows changes. The intersection() method does nothing but call a global intersect() function. We added numerous intersect() functions to the global name space to encourage developers to use them for other purposes. We provide the process() function to allow derived classes to define their own processing behavior, if necessary.

3.5 Practical Results

After having discussed all the steps required for building a component-based commercial software for image processing, we present an example of implementation with Visual C++6.0 and API Windows. All the code that we have explained for the components and interfaces has been written along with adequate API functions necessary for the creation and view management of the different windows of the software. Fig. 5 shows a dialog box for opening images, while Fig. 6 shows a display of a cytological sample of a malign breast cancer image, on which we will do an image sharpening and texture enhancement operation.

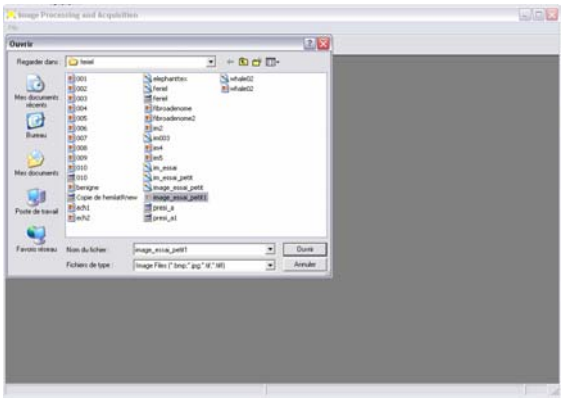

Fig. 5 Dialog box for opening images.

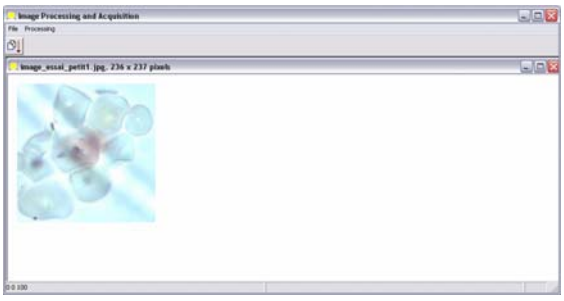

Fig. 6 A display of a cytological sample of a malign breast cancer image

3.5.1 Image Sharpening and Texture Enhancement

As an example of functions for processing single source images we decide to implement the image sharpening and texture enhancement processing. This processing is one of the most important especially in the medical imaging field. For example it can be helpful to delimit the edges of organic cells in order to count them or to identify some malign cancer zones in a given tissue. The processing of color images is nowadays a thriving field of research because there is no consistent method for comparison between vector data [3] and because the images are multi-components, and the components are processed either separately or after being mapped by a nonlinear map [11]. The aim of this operation is to emphasize those image regions or ordered image point patterns, whose spectrum mostly contains higher spatial frequencies. The aim is to emphasize details, edges, fine textures etc. without emphasizing noise.

3.5.2. Adaptive Contrast Enhancement at Edges

There are many methods for doing an image sharpening and texture enhancement [5]. However, we have chosen the one that gives an adaptive contrast because it takes into account the effect of light reflectance. This is how comes in the case of a gray image. The measure of contrast enhancement is locally adaptively controlled by an opportunely defined edge gray value E. The value E represents a weighted average of all edge pixels inside the given picture window. The local contrast C is defined by the relative deviation between E and the gray value F(i,j) of the current point. The algorithm of the processing is shown below for a picture window nxn.

```
S:=0;
Q:=0;
for j:=2 to n-1 do
        for i:=2 to n-1 do begin
                dij:=max{|F(i1,j+1)+F(i,j+1)+F(i+1,j+1)-F(i-
                1,j-1)-F(i,j-1)-F(I+1,j-1)|,|F(i-1,j+1)+F(i-
                1,j)+F(i-1,j-1)-F(i+1,j+1)-F(i+1,j)-F(i+1,j-
                1)|};
                S:=S+dij.F(i,j);
                Q:=Q+dij;
        end{for}
        E :=S/Q ;
        C:=[|F(k+1,k+1)-E|/(F(k+1,k+1)+E]^r;
        if(F(k+1,k+1)<=E) then
                w:=E(1-C)/(1+C);
        else
                w:=E(1+C)/(1-C);
        H(x,y)=adjust(w);
```

end{for}

Where r is an input parameter less than 1, and adjust() is a function that adjusts the value of w to the domain [0,255]. This method produces a contrast enhancement because the gray values close to the edges are decreased if they are inferior to the weighted edge gray value E. On the other hand, the gray values are increased if they are larger than E. The local contrast C as defined above represents a measure of deviation between the value E and the gray value F(x,y) which has to be transformed. The increase in contrast can be adjusted by the parameter r. Fig. 7 shows a window displaying the result of the operation on the image of Figure 6. We have applied the algorithm on each image component separately with a picture window 7x7.

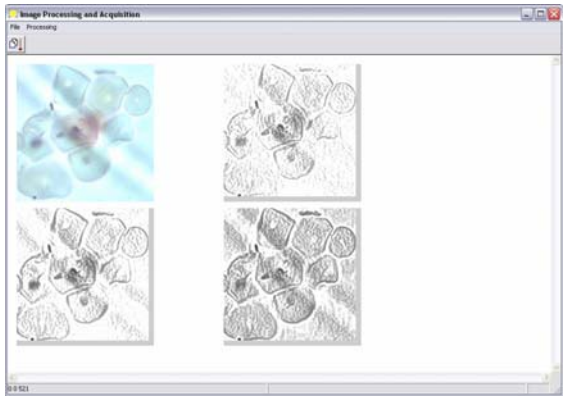

Fig. 7. A window displaying an adaptive contrast enhancement of a cytological sample of a malign breast cancer image on each image component. Top left: the original image. Top right: the result of the operation on the red component. Bottom left: the result of the operation on the green component. Bottom right: the result of the operation on the blue component.

4. Conclusion

In this paper, we have discussed a method for building component-based commercial software for image processing. It implements handles and rep objects, and use templates so a single version of the code will work with any type of pixel. Thanks to the handles, we are able to manage the number of copies of a given image and so avoid memory leakage and fragmentation. The method deals separately with the issues related to image processing problems such as memory management and memory alignment. This will enable us to enforce our focus on the algorithm. Also, thanks to file delegate the software can deal with any image format. Our work enabled to explore such issues as: What's the best way to design this application, using inheritance or templates [2]? Should we do everything at static initialization time or use a Singleton object? Does explicit template instantiation give us any syntactic or functional advantages? Does reference counting using rep objects and handles add to design? How do we partition functionality between global functions and objects? What kind of framework makes sense for handling exceptions? Does template specialization help us?

References:

[1] Bastide R., and Sy O., Towards components that plug AND play. In Vacillo A., Hernander J., Troya J. M. (eds), Proc. *ECOOP'2000 Workshops on Object Interoperability (WOI'00)*, pp. 3-12. Extremadura University Press, Cannes, France.

[2] Canal C., Pimentel E., and Troya J. M., Compatibility and inheritance in software architectures, *Science of Computer Programming*, 2000..

[3] Chanussot J., Approches vectorielles ou marginales pour le traitement d'images multi-composantes. *PhD, Thesis*, Université de Savoie, 1998.

[4] Gamma E., Helm R., Johnson R. and Vlissedes J. M., *Design Patterns*. Reading, MA: Addison-Wesley, 1995.

[5] Klette R., Zamperoni P., *Handbook of Image Processing Operators.* John Wiley and Sons, 1996.

[6] Leavens G. T., Sitaraman M. (eds) 2000. Foundations of Component-Based Systems, Cambridge University Press, Cambridge, UK.

[7] Pratt W. K. *Digital Image Processing*, Third Edition, Indiapolis, IN: Wiley, 2001.

[8] Romanik P., Mutz A., *Applied C++ Practical Techniques for Building Better Software*, Addison-Wesley, 2003.

[9] Stroustrup B., *The C++ Programming Language*, Special Edition, Boston: Addison-Wesley, 2000.

[10] Sutter H., *More Exceptional C++*. Boston: Addison-Wesley, 2002.

[11] Trémeau A., Fernadez_Maloigne C., Bonton P., *Image numérique couleur – De l'acquisition au traitement*, Dunod, Collection Sciences Sup, 2004.

[12] Vandevoode D., Josuttis N. M., *C++ Templates*, Boston: Addison-Wesley, 2003.