A Reusable Camera Interface for Rovers

Daniel S. Clouse, Issa A. D. Nesnas, and Clayton Kunz

Abstract—**The CLARAty camera interface is used for the acquisition of images and control of cameras on research rovers for a number of NASA-supported research projects. A third revision of the camera interface was recently released. This paper traces the evolution of this interface over the past five years and outlines the design and implementation of the latest version.**

I. Introduction

CLARATY (Coupled Layer Architecture for Robotic Autonomy) is an object-oriented software infrastructure for the development and integration of new robotics algorithms, principally for use on rovers [1]. Its purpose is to provide a common interface to a number of heterogeneous robotics platforms to simplify the initial development and reuse of robotics algorithms in the areas of vision[2], manipulation, navigation[3], and planning[4]. CLARAty currently supports research rover platforms developed and used at the Jet Propulsion Laboratory, Ames Research Center, Carnegie Mellon University, and the University of Minnesota.

In this paper, we present, in detail, the design of a new camera interface for use within the CLARAty framework. CLARAty has been under development since 1999, and included a camera interface from early on. The first version of the interface [5] was basic and offered limited functionality that was not sufficient to support the increasing demands of robotic image acquisition for planetary exploration. The primary limitation was that the camera abstraction was single threaded and did not allow for functional extensions except along the hardware axis.

The second revision allowed for use in multiple threads by locking individual cameras during image acquisition. However, the design did not incorporate locking in the camera settings placing that burden on the application developers to keep track of these settings. This left no way for a thread to insure that the settings it requested were applied when acquiring a particular image. It addressed the extendibility limitation by introducing a bridge pattern [6] that separated the camera hardware specialization axis from the camera functional specialization. It also introduced a capability for logging camera information using a generic mechanism that has been deployed across all devices. These changes led to a design that became significantly more complex than the first revision and was hence harder to maintain over time. The separation of the internal state from the device abstraction to allow logging created a design with numerous classes that was hard to adapt and extend. It also required changes to multiple places in the software for adding new capability. Needless to say, we were not satisfied with the complexity and lack of maintainability of that interface. The third revision was intended to capture the best of the first two revisions while adding better support for multi-threaded applications. The remainder of this paper is devoted to a description of the new camera interface and the process by which it was conceived and developed with occasional reference to the two earlier revisions.

Section II describes the requirements imposed on the new interface by the rover environment. Section III discusses and justifies some of the design decisions made in the development of the new interface. Section IV discusses some of the problems encountered in developing an implementation of the new interface for a particular camera type. Section V summarizes the lessons learned from this enterprise.

Manuscript received December 15, 2006.

D. S. Clouse (Daniel.Clouse@jpl.nasa.gov) and Issa Nesnas are with the Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, USA.

C. Kunz is with Ames Research Center, Mountain View, CA, USA, and with Massachusetts Institute of Technology, MA, USA.

II. Requirements of the New Interface

In the early stages of the design process for the third revision, we developed an extensive list of requirements for the new interface. Though most of these requirements were eventually supported in the final design, some were more influential than others. In this section we discuss the short list of most influential requirements.

Common tasks are easy. We have many transient users, and many users who are not nearly as interested in the minutia of camera control as they are in working out the details of their algorithms. These users just want to take single and multiple synchronized pictures so they can get on with their work. Therefore setting up the cameras, adjusting parameters, and taking pictures must be easy to do.

Full access to hardware. Another class of users is very interested in having fine control over the camera, as this is essential to making some algorithms work well. These users need access to the full functionality of the underlying hardware, and are willing to put in the extra effort to learn how to use it.

Implement what is commonly used. Our experience with the original camera interface motivates our desire to avoid implementing features in the generic interface that will not be commonly used across applications particularly if doing so complicates the implementation. Simplicity is necessary not only to keep the interface easy to use, but also to keep it easy to port to new platforms, and maintain. Implementing the union of all possible functions leads to a system that is harder to adapt and maintain. Implementing the minimal set (intersection of all functions) leads to a system that requires extensions for every practical application, thus increasing complexity. We adopted an approach that lies somewhere between these two extremes. For specialized features, waiting until a feature is needed allows the development of a better understanding of the requirements of that feature. In particular, a number of features have been identified as not immediately necessary for our purposes. These include video, timeout on image acquisition, and returning the same image to two or more tasks when image requests appear simultaneously. Adding new features as they are needed may be accomplished through inheritance, or by adding new member functions to the base class.

Stable interfaces. To the extent possible, we would like our interface to be defined well enough that users are not required to change code when they change the type of camera. This is especially difficult when it comes to controlling the myriad camera parameter settings: shutter speed, gain, frame rate, etc. Despite recent efforts to standardize camera control [7][8], there are no widespread standards for controlling both digital and analog cameras (with their corresponding frame grabbers). In existing standards, the meanings of the parameter values are often left to the camera manufacturers. For the most common of these settings we would like to define the meanings of the parameters in a common way.

Synchronization. Many algorithms require images taken by multiple cameras to be precisely synchronized in time. This is particularly evident for stereo algorithms that operate while the robot is moving. This synchronization requirement is not limited to stereo processing or to two stereo cameras however. For example, an algorithm that hands off a feature that is tracked in one stereo pair to another [9] may need to precisely synchronize all four cameras.

Transient camera groups. Precise synchronization requires hardware support for simultaneous image acquisition. The level of support for this synchronization varies with the hardware. In particular frame-grabber hardware is generally capable of synchronizing a large number of cameras with a micro second latency, while the Firewire camera interface that we use on some platforms is capable of synchronizing only 4 cameras at a time. This Firewire interface does however allow any 4 cameras on the same bus to be synchronized. Since our rovers often have more than 4 cameras, a method is needed in the software interface for indicating which subset of cameras is currently to be synchronized. We must be able to change this subset often and with little overhead during processing.

Task Safety. Generally, algorithms are developed in isolation. At times, we do need to run these algorithms simultaneously in separate tasks, however. We would like these algorithms to know as little as possible about each other, but still be able to cooperate in the use of these shared camera resources.

III. Interface Design Decisions

In this section, we discuss a number of design problems and how they were addressed in the final design to make the interface reusable.

A. Task Safety

One of the most challenging aspects in designing the new camera interface is the requirement that two or more independent tasks be able to share the cameras without explicit cooperation. There are essentially two kinds of operations a user performs in interacting with cameras. First, the user may set or query certain parameters such as image format, image size, shutter speed, and gain. Second, the user acquires an image. If two processes are using the same camera, it is possible that process A will change the settings on the camera between the time process B sets its parameters and the time it acquires its image. Some method is needed to avoid this kind of interaction.

The first revision of the camera classes did not address task safety. It was limited to a single-threaded implementation. The second revision made the camera devices multi-thread safe using locks that were inherited from a generic device abstraction. If a process grabbed the lock on a particular camera, no other process was allowed to use that camera until the lock was released. Generally, the implementations for various camera types used this locking mechanism to lock only the image acquire operation, leaving users with no way to lock in a set of camera parameter settings to be used for a particular image. One way to work around this defect would be to give the user direct access to the device lock. This would allow a thread to guarantee its settings are in effect at the time the image is taken. However, there are problems with this method.

First, to guard against another task changing the camera settings you must write your code to reestablish your settings every time you grab the lock. This requires extra code to be written at the application level to use the cameras in a multitasking environment.

Second, it assumes users will write code in a cooperative manner, releasing the lock when a camera is not in use. If this assumption fails, one process can easily starve all the others. It is tempting to write this kind of non-cooperative code to avoid the extra work of reestablishing the camera settings for every image acquisition.

Third, when multiple cameras are involved, it is possible for the system to become deadlocked. For example, process A holds a lock on camera 1 while waiting for the lock on camera 2, while process B holds the lock on camera 2 while waiting for the lock on camera 1.

To address these problems, in the third revision of the design we introduced the concept of separating logical and physical cameras. This concept has since been incorporated into the implementation of other devices within CLARAty [10].

There are two kinds of camera objects in the new interface, logical cameras, and physical cameras. Implementation of new camera hardware will require writing one of each. A physical camera represents the camera hardware. Its API (application programming interface) is unique to that specific camera type, and reflects the capabilities and limitations of that particular camera hardware. Our intent is that no more than one physical camera object exists for any camera on the system. This object is shared among all threads. A physical camera is responsible for:

1. getting/setting camera parameters within the physical device,
2. acquiring an image from the physical device,
3. implementing a locking mechanism which allows a logical camera to block any other logical camera from setting camera parameters, or acquiring an image.

In contrast, a logical camera represents the user's view of the camera. Parameter settings made in one logical camera do not affect the state of another logical camera, even if both refer to the same piece of camera hardware. The implementation of a logical camera is required to maintain a local cache of the current user's view of the camera state. The acquire member function of the logical camera affects an atomic operation which both sets the physical camera parameters to match the user's view of the camera state and acquires a single image.

The following code demonstrates how to declare the required objects and acquire images.

```
1: Image<uint8_t> img1, img2;
2: X_Hw_Camera hw_cam1(id_unique_to_hardware);
3: X_Camera  cam1(hw_cam1);
4: cam1.set_brightness(0.35);
5: cam1.acquire(img1);
6: cam1.acquire(img2);
```

Line 2 declares the physical camera for a camera of type X. The constructor takes an identifier to specify to which camera hardware this object refers. It is an error to call this constructor more than once with the same identifier across all threads. In line 3, the physical camera object serves as an argument to the logical camera constructor. More than one such constructor call may reside in different threads, each referring to the same physical camera. In line 4, the settings affecting the brightness of the returned image are apparently changed. This change does not actually take effect until line 5 when the acquire member function is called. Since the acquire call is an atomic operation that both sets the camera parameters and takes the picture, we are assured that the brightness setting will be 0.35 when the image is taken. If some other process were to change the brightness setting on the physical camera between lines 5 and 6, it would be restored to 0.35 before the image was taken a second time in line 6.

Note that using this interface, the user need not write any special code to maintain task safety. The code written for single-threaded operation is identical to that required for cooperation between multiple threads.

This does complicate the implementation of the logical camera however. The logical camera must maintain a cache containing the state of all the camera settings. This cache is updated by the set_brightness member function. During the execution of the acquire member function, the values of all the settings are transferred to the physical camera while the physical camera in under lock. To avoid unnecessary overhead, code is included to make sure this transfer only occurs if the cached values or the camera settings have actually changed.

B. Design Pattern

The second revision of the Camera interface (Fig 1) separates the hardware adaptation from the functional adaptation using a bridge pattern [6]. The Camera class defines the user interface. However, the implementation of a specific camera is derived from a generic Camera_Impl class. A Camera contains a pointer to a Camera_Impl and all interaction with the camera hardware is effected via Camera_Impl member functions.

This design was chosen to make it easy to add new functionality. For example, to add video one could write a Video_Camera that inherits from the Camera class, adding new member functions, start_video, end_video, and acquire_video. If these new members could be written using calls through the Camera_Impl interface, then video would be added for all types of camera hardware without changing all the various camera implementations.

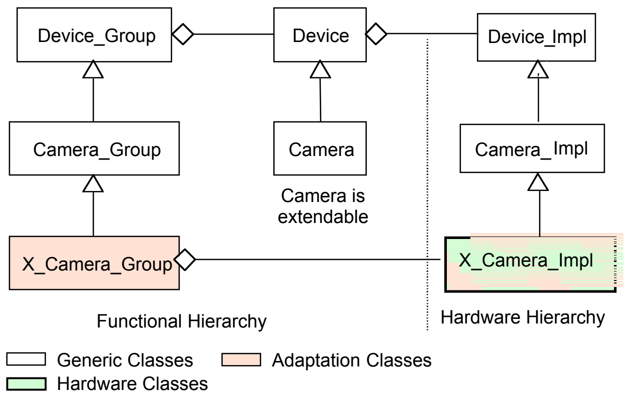

Fig 1. Revision 2 Camera Hierarchy.

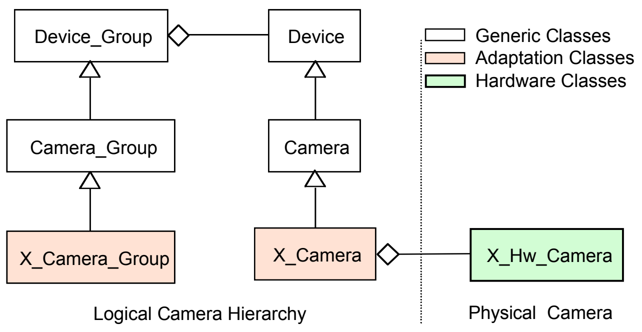

Fig 2. Revision 3 Camera Hierarchy.

The bridge pattern turned out to be a poor design choice. It introduced significant complexity without offering much in the way of flexibility. On the complexity side of the equation, the pattern resulted in parallel hierarchies with similar APIs. The API of the Camera_Impl ended up being as detailed as the Camera API, effectively doubling the complexity of the design. On the flexibility side of the equation, it turned out that there was little need to add new functionality to the interface. In addition, it is not clear that adding functionality as complicated as video could be accomplished without adding new functionality to the Camera_Impl interface as well, so the presumed advantage of the bridge pattern went unrealized.

In the new design, revision 3, (Fig 2) the implementation of a physical camera class is left completely unconstrained by any base class. You can think of a physical camera as a device driver. The physical camera need only support those functions required by the logical camera implementation specific to a particular piece of camera hardware.

In contrast, all logical cameras are derived from the base Camera class by simple inheritance. The Camera class defines a set of member functions used to perform all the common operations on a camera (Fig 3). This allows high-level code to be written using the Camera interface without knowing the type of physical camera being used.

```
class Camera : public Device {
public:
  enum IMAGE_FORMAT {MONO8, YUV411, YUV422,
                     YUV444, RGB8, MONO16, RGB16};

  virtual bool    set_contrast(double gain_frct) = 0;
  virtual double get_contrast() const = 0;
  virtual bool    set_brightness(double offset_frct) = 0;
  virtual double get_brightness() const = 0;
  virtual bool    set_exposure(double seconds) = 0;
  virtual double get_exposure() const = 0;

  virtual bool    set_format(IMAGE_FORMAT format,
                             int width_pixels,
                             int height_pixels) = 0;
  virtual IMAGE_FORMAT get_format() const = 0;
  virtual int      get_width()  const = 0;
  virtual int      get_height() const = 0;

  virtual void acquire(Image<uint8_t> & image,
                       Time*       timestamp = NULL,
                       Feature_Map* feat_map = NULL) = 0;

  virtual void acquire(Image<uint16_t> & image,
                       Time*       timestamp = NULL,
                       Feature_Map* feat_map = NULL) = 0;
};
```

Fig 3. Simplified Definition of Camera Class.

Each derived logical camera implementation must provide enough state to remember the settings desired for this camera, and the glue logic to turn the base Camera class member functions into calls to the physical camera interface. Low-level operations specific to a particular type of camera may also be supported by a logical camera implementation by adding new member functions.

We hope that sacrificing some of the unused flexibility of the bridge pattern will result in a design which is simpler, more maintainable, and easier for users to understand.

C. Camera Parameter Settings

In keeping with our requirement to implement only what is commonly used, we have chosen to include only three parameter setting member functions in the base Camera class: set_contrast, set_brightness, and set_exposure. Brightness is a multiplicative parameter controlling gain, whereas contrast is an additive one. Set_exposure controls the exposure time. These parameters are the most commonly used, and are supported in some measure on most camera hardware. Setting these parameters will be especially easy for the user because the same interface is used for all cameras.

To meet the requirement of well-defined interfaces, we are providing common meanings for all of these parameters. Exposure time is defined in seconds. For each camera, brightness takes on values between 0.0 and 1.0, which span the entire range of gain settings. Similar to brightness, contrast takes on values from 0.0 to 1.0 spanning the entire range of values available for any particular camera.

For the benefit of power users, additional parameters may be supported by specific camera implementations to allow full control of the hardware.

D. Support for Image Formats

The set_format member function allows selection of the image format to be returned by acquire. The set_format function also specifies the size of the image, as the range of available image sizes is often determined by the format. If a particular camera does not support the specified image format, false is returned, and the format is left unchanged.

Two versions of the acquire member function are supported. One version returns Images containing 8-bit elements. The other returns 16-bit elements. The image is resized, if necessary, to match the geometry expected by the camera hardware.

For each version of acquire two optional parameters are included in the prototype. If timestamp_ptr is not NULL, it points to an object that will return the time at which the image was taken. If feature_map_ptr is not NULL, it points to a Map object for returning the settings of camera parameters in effect when the image was taken.

CLARAty defines a generic Image data type that can support color images in more than one way. Rather than allow the Camera class to determine the specific representation for all color images, we decided to return images using the memory layout defined by the physical camera. For example the first 4 bytes of the Image returned from acquire when the camera is in YUV422 format mode will contain the U, Y1, V, and Y2 components in that order. Using these components, it is possible to build up the first two pixels in an RGB image. That translation from YUV to RGB is not performed within the Camera class however. Making this translation an explicit function call outside of the Camera class allows us to keep the Camera class relatively simple, and will make it easy to extend the class to new formats in the future.

E. Camera Groups

Our solution for specifying the set of cameras to synchronize is embodied in a new class hierarchy based on class Device_Group. You will notice in Fig 2 the similarity between this new Camera_Group hierarchy and the Camera hierarchy.

The Device_Group base class (Fig 4) implements what is essentially a container that holds a group of pointers to

```
class Device_Group  {
public:
  unsigned int  size() const;
  void        append(Device & device);
  void        remove(Device & device)
                    throw(invalid_argument);

  template <class Sub, class Arg>
  int for_each(bool Sub::*memb_func(const Arg &),
               const Arg & value);
};
```

Fig 4. Simplfied Definition of Device_Group class.

```
class Camera_Group : public Device_Group {
public:

   virtual int acquire(vector<Image<uint8_t>*>& im_vec,
                       vector<Time> *              ts_vec = NULL,
                       vector<Feature_Map*> *    fm_vec = NULL);

   virtual int acquire(vector<Image<uint16_t>*>& im_vec,
                       vector<Time> *              ts_vec = NULL,
                       vector<Feature_Map*> *    fm_vec = NULL);
};
```

Fig 5. Simplified Definition of Camera_Group Class

Devices. For our purposes, these Devices will be Cameras. Member functions are supplied to append and remove Devices from the container. The template function, for_each, supplies a way for some member function of Camera to be called for every camera in the group. This allows, for example, set_brightness to be called to set each camera to the same brightness level. The value argument to for_each specifies the brightness level.

The Camera_Group class (Fig 5) supplies new acquire member functions that take synchronized images from all cameras in the group. Two virtual functions allow acquisition of 8 and 16 bit images. The caller must supply a vector of Image pointers in which the images are returned. Optional vectors return timestamps, and maps of the camera parameter settings at the time of acquisition. A default implementation of the group acquire grabs images from the cameras in the group in sequential order without true hardware synchronization. For true synchronization of image acquisition for a new camera type X, you need to derive a new class (X_Camera_Group) from the Camera_Group base class. The two virtual acquire member functions will need to be written to support synchronized acquire for the new X cameras.

The following code demonstrates how to declare the required objects and acquire synchronized images from a pair of cameras.

```
1: X_Hw_Camera pcam1(id1);
2: X_Hw_Camera pcam2(id2);
3: X_Camera   cam1(pcam1);
4: X_Camera   cam2(pcam2);
5: X_Camera_Group grp(cam1, cam2);
6: grp.for_each(set_brightness, 0.35);
7: vector<Image<uint8_t> > images(2);
5: grp.acquire(images);
```

Lines 1 and 2 declare two physical cameras of type X. Lines 3 and 4 create logical cameras referring to the physical cameras just constructed, just as in our previous example. In line 5, a Camera_Group containing the two cameras is constructed. The X_Camera_Group class is a Camera_Group object specific to cameras of type X. This class allows a group of two cameras to be built by simply passing them as arguments to the constructor. In line 6, the settings affecting the brightness of the returned images are changed using the for_each function. Line 7 declares a vector of two images which are passed to the group acquire function in line 8. Acquire resizes the images to match the physical cameras and returns synchronized images acquired by the two cameras.

IV. Implementation and Efficiency Considerations

The common Camera interface was designed for reusability. It presents to the user an interface that is simple to use and understand. The apparent simplicity of this interface hides some vexing implementation problems though. In this section we discuss some of the design decisions made for one particular implementation, that of the IEEE1394 (or Firewire) interface for VxWorks.

Three new classes are included in the implementation. A class representing a physical camera is named MR1394_Hw_Camera. A class representing a logical camera, and derived from the Camera class is named MR1394_Camera. A class derived from Camera_Group is name MR1394_Camera_Group. The MR in the names refers to the MindReady IIDC driver interface [11] that this implementation uses to communicate with the camera hardware. The IIDC interface specification [7] provides a common interface allowing MR1394_Hw_Camera to support a variety of cameras.

A. Fast Synchronized Image Acquisition

In our implementation, synchronization between two or more cameras is accomplished by putting them into video mode. The cameras then self-synchronize via the Firewire bus. Acquiring a synchronized image from a set of cameras that are already in video mode is fast. It can be accomplished at frame rates. A naïve implementation of the Camera_Group acquire operation would enter and exit video mode once for each acquire operation. This would slow things down by over 1 second per frame. To avoid this overhead, it would be nice if we could leave all the camera in video mode all the time. Unfortunately, as we will see in the following section, the scarcity of DMA channels and bus bandwidth limit the number of cameras left concurrently in video mode. Our solution is to always leave a camera in video mode at the end of a synchronized acquire. The next acquire operation determines which cameras to remove from video mode to free up enough resources to complete its function. In this way, we avoid the overhead caused by unnecessarily exiting and reentering video mode.

B. Resource Limits and Multi-Tasking

As we mentioned in the previous section, the number of DMA channels and total bus bandwidth limits the number of cameras that can be simultaneously left in video mode. Specifically, most Firewire cards support only 4 DMA channels and 320 Mbps for synchronized images. As we mentioned in section III.E, accommodation for these limitations was provided in the interface design by supporting synchronization via transient Camera_Groups.

It is a simple matter to include code in the MR1394 implementation to limit the size of any camera group to 4 cameras, and the bus bandwidth required of any single acquire opration to 320 Mbps. More difficult is the task of insuring that neither the 4 DMA channel limit, nor the bus bandwidth limitation is exceeded by any group of independent threads.

The implementation provides a function, grab_all_resources, which atomically captures all the resources needed to finish both parameter setting and image acquisition from all cameras in a Camera_Group, or from an individual camera (which is equivalent to a Camera_Group of one). Required resources include exclusive access to the individual cameras, one DMA channel per camera, and enough bus bandwidth to transfer all the images at the current frame rate. If any resource is not immediately available, the function waits until it is freed by some other thread. If, after grabbing enough DMA channels, the bus bandwidth does not allow the new acquire operation to proceed in parallel with existing acquires, the task will continue to wait for more DMA channels to be released until the total bandwidth requirements are met.

The following invariants assure us that the resources needed to complete an acquire operation will be available in a short time without the possibility of deadlock:

1. Any acquire operation that exceeds total system resources fails without grabbing resources.
2. Only one thread at a time is allowed to grab camera resources. Thus, resource grabbing is effectively an atomic operation.
3. Any camera resources which are currently unavailable are held by currently executing acquire operations.
4. All camera resources are released at the completion of the acquire operation.
5. All acquire operations are of short time duration.

Some concurrency is lost in this design because it does not allow multiple threads to concurrently grab camera resources. More importantly however, the design does allow multiple threads to acquire images simultaneously if resources are available.

C. *Camera Feature Caching Efficiency*

In Section III.A we described how, at the logical camera level, any of the feature setting member functions (e.g. set_brightness) do not immediately cause a change to the camera hardware. Rather, the value of a new setting must be cached within the logical camera class, and only applied to the physical camera at the time of the next acquire operation. The Firewire implementation does not allow the camera settings to be changed in video mode, and so a method is needed for quickly detecting if the settings have changed since the last acquire to avoid the overhead of unnecessarily removing a camera from video mode.

To detect this situation, the MR1394_Hw_Camera class includes a feature setting counter. When a logical camera changes any of the feature settings on a camera, it increments that counter. The next time it wants to acquire an image, it checks the value of that counter to see if it has changed. If it has not, it can forgo resetting the features and thus avoid the expense of leaving video mode.

V. Lessons Learned

In conclusion, we would like to highlight some lessons learned as a result of our experiences. First, designing good reusable interfaces is not easy. We believe that the process of architecting and designing an interface is essential to producing a useable product. However, there is no substitute for the experience gained from implementing and using the interface over a period of time. When problems are discovered, rearchitecting and redesigning the interface incorporating the insight gained from experience will result in a better design.

Second, our experience with the second revision of this interface suggests that too much flexibility is not necessarily a good thing. The added flexibility may add complexity that makes the interface more difficult to understand and maintain. In the third revision, in order to simplify the design, we removed some flexibility, making it more difficult to extend the Camera class along the functional axis. That increased simplicity allows us to concentrate on supporting features, which are more likely to be needed by users, such as, improved support for multitasking.

Third, implementing a reusable interface design is necessarily more complicated than implementing a design that is not reusable. To support multiple camera types in our design, it is necessary for us to hide the details of any particular type of camera from the user. This complicates the code required to support any particular type of camera, as demonstrated in section IV. The increased complexity inherent in a reusable interface is not always justifiable. However, in cases, such as CLARAty, where a significant number of applications use the interface, the pay off for the increase complexity comes from the ability to leverage common infrastructure and algorithms across multiple platforms.

Acknowledgment

Our thanks go to Lorenzo Fluckeiger, Richard Madison, Michael McHenry, and Hans Utz for the many useful ideas they contributed. We would also like to thank I-Hsiang Shu, Jeffrey Edlund, Nik Melchoir, and Robert Steele for participating in the code review process. Our thanks also go to our sponsor, the NASA Mars Technology Program, for their support. The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology with contributions from NASA Ames Research Center and Carnegie Mellon under a contract to the National Aeronautics and Space Administration.

References

[1] I. A. Nesnas et al., "CLARAty: Challenges and Steps Toward Reusable Robotic Software," *International J. of Advanced Robotic Systems*, vol. 3. no. 1., 2006, pp. 23-30.
[2] I. A. Nesnas et al., "Visual Target Tracking for Rover-based Planetary Exploration," *Proceedings of the IEEE Aerospace Conference, Big Sky Montana, March 2004.*
[3] C. Urmson, R. Simmons, I. Nesnas, "A Generic Framework for Robotic Navigation," *Proceedings of the IEEE Aerospace Conference, Big Sky Montana, March 2003.*
[4] T. Estlin et al., "Continuous Planning and Execution for an Autonomous Rover," *Proceedings of the Third International NASA Workshop on Planning and Scheduling for Space, Houston, TX, October 2002.*
[5] R. Volpe, I.A.D. Nesnas, T. Estlin, D. Mutz, R. Petras, H. Das, "CLARAty: Coupled Layer Architecture for Robotic Autonomy." *JPL Technical Report D-19975*, Dec 2000.
[6] E. Gamma et al., "Design Patterns: Elements of Reusable Object-Oriented Software", Reading, Mass: Addison-Wesley, 1995.
[7] 1394 Trade Association, "IIDC 1394-based Digital Camera Specification", ver. 1.31, 2004.
[8] Specification of the Camera Link Interface Standard for Digital Cameras and Frame Grabbers, October 2000.
[9] R. Madison, "Improved Target Handoff for Single Cycle Instrument Placement," *Proceedings of the IEEE Aerospace Conference, Big Sky Montana, March 2006.*
[10] I. A. Nesnas, "The CLARAty Project: Coping with Hardware and Software Heterogeneity" in the *Software Engineering for Experimental Robotics*, Springer Tracts on Advanced Robotics, ed. Davide Brugali, 2006.
[11] Mindready Solutions Inc., "Instrument & Industrial Digital Camera IIDC IEEE-1394 Protocol Reference Manual", ed. 1, rev. 2, 2003.