NGAO Software Architecture
KAON 675: Tasks

Author	Modified	Notes
Douglas Morrison	7/30/09	NGAO Task Design

1.0 Introduction

Distributed systems and applications built with the Keck Common Services Framework (KCSF) are based on the control and management of devices. These software components provide a low level control interface to their associated hardware, but do not typically implement complex or coordinated tasks individually, or between multiple devices and systems. Building a system control application with individual devices would be a difficult undertaking. As the functional requirements of the application grow, the development and maintenance of such an application would quickly become unmanageable. An intermediate layer of control needs to be introduced to bridge the low level devices and high level applications. This document discusses the design and use of KCSF Tasks.

2.0 Overview

At the lowest level of the Keck Common Services Framework are controllers and devices. These distributed software objects interface directly with hardware and other systems. A number of these devices and systems will need to work in concert to configure and prepare the system for observing, and during observing. Tasks provide application developers with the means to control and coordinate devices at varying levels of granularity through a common interface.

Tasks offer developers the flexibility to build layers of increasingly complex functionality from simpler self contained operations. Tasks of all levels present a simple uniform interface to the application developer, effectively abstracting away the underlying functionally and device complexity. Tasks are designed to provide a scalable solution to controls development through loose coupling, reusability and consistent use of the Command design pattern.

2.1 Approach

All tasks implement the Command pattern, where a command object encapsulates an action and its parameters. The framework provides a base task class that abstracts services for processing status and other common infrastructure activities. Upon this is built and provided a set of "atomic" tasks for telescope and instrument control through the distributed devices. A set of "container" tasks based on common sequential and concurrent command processing paradigms is also included.

All tasks share the same exact interface; it is straightforward to build up compound tasks by plugging simple tasks into container tasks and container tasks into other containers, and so forth. In this way various advanced astronomical workflows can be readily created, with well controlled behaviors. In addition, since tasks are written in Java it is easy for astronomers to subclass and extend the standard observatory tasks with their own custom extensions and behaviors.

A key concept is how rigorous use of the command pattern can make complex astronomical workflows realizable via reusable observation components in much the same way that complex graphical user interfaces can be created by leveraging the commonality of widgets.

Advantages of this approach are,

- Easy to extend the system with new command structures and primitives.
    - Software/AO group can build up a collection of simple and compound tasks that is useful for the vast majority of typical observation activities.
    - Advanced users can create their own tasks to add to the set, possibly sub-classing existing tasks.
- Has the advantage of a full-featured, object-oriented, widely-supported programming language.
- Easy to build a UI/ RAD tool around.
- Easy to modify or replace existing tasks.

2.1.1 Command Pattern

The key feature of the Command Pattern is that software objects are used to represent actions and tasks. Classes provide developers with the ability to encapsulate actions and parameters, as well as hide the underlying framework related requirements (such as connecting to distributed components).

As objects, tasks provide a convenient temporary storage for procedural parameters and can allow a user to assemble a command some time before it is actually needed.

Subclassing tasks allows developers to add parameters and members custom for a specific operation. This may include information such as the task name, who started the task, and how long it has been running. In addition, treating tasks as objects will allow developers to implement additionally control mechanism to a task such as pause, cancel and resume.

Utilizing a generic task interface, developers will be able to implement compound and hierarchical commands where a set of tasks may be managed by a higher level implementation. Developers can use these meta-control structures to build complex command systems, without increasing the complexity of management, or burdening the user with understanding how a particular task instance functions. From the perspective of the developer all task instances look the same and are commanded in an identical fashion.

2.1.2 Execution Environment

Task functionality and execution are considered two separate and distinct aspects within system control and commanding. Task development is focused on the functional requirements of an operation. The lifecycle management and execution of a task (including resource allocation and scheduling) are the responsibility of a task Executor. This design is similar to the Container Component Model (CCM) used in the deployment and execution of device controllers: containers are responsible for managing the technical requirements, while components are responsible for implementing the functional requirements.

This separation between the functional and technical requirements of a task allow for greater development and deployment flexibility and independence. A task developer can focus solely on the design and functionality of a task, while an application developer only needs to consider how the tasks are managed and executed. A number of task executor solutions can be developed to satisfy a wide range of runtime scenarios, without requiring prior knowledge about how individual tasks work and what operations they perform.

As with the CCM, the functional / technical separation of tasks can be achieved by using a well defined and generic interface between the Task and Executor definitions. The interfaces (discussed later in this document) are minimal; consisting of only a few key methods to provide the basic operations needed to manage the life cycle and execution of tasks.

2.1.3 Core Task Library

The base Task implementation and its derived classes will be maintained in a Java module called the Task Library. This library will also contain auxiliary classes and utilities that can aid developers in using and managing tasks. As additional tasks and capabilities are developed they can be added to the Task Library. Ideally the library will provide all of the functionality required by the system through the task implementations. The goal is to provide a highly reusable and flexible task infrastructure which users and developers can leverage to build NGAO applications.

3.0 Task Library

Tasks can be used to efficiently implement an application's control functionality by interfacing with KCSF components to execute their specific business logic. A cornerstone of the Command design pattern is that all tasks must implement a common control interface. This not only makes tasks appear generic from the point of view of the execution environment, but allows developers to build a hierarchy of nested tasks.

Tasks can be arbitrarily complex: from simply slewing a device to closed loop image processing and control. Although a single task can be developed to perform very complex actions, the Command design pattern encourages developers to break a complex problem down into smaller discrete tasks. These tasks can in turn be grouped into higher-level task controllers (known as compound tasks) to perform the desired overall sequence. The benefit of this approach is two fold:

- Simpler tasks allow for greater reuse and provide a high degree of flexibility for variations in the system.
- Bugs, code changes, and upgrades will be confined to smaller portions of the software, allowing for faster and more reliable updates with less time required for testing and deployment.

There are two general types of compound task controllers that will be provided with the Task library: Sequential tasks and Concurrent tasks. Sequential tasks are designed to iterate through a list of task objects, executing and waiting for each task to complete before moving on to the next. Once all of the subtasks have been executed the sequential task will be considered complete. Concurrent tasks are designed to execute a set of subtasks simultaneously. After the last task has finished executing the concurrent task will be considered complete.

Essentially these two compound classes provide developers with serial and parallel task execution. As these compound task controllers themselves implement the Task interface one could nest a sequential task within a concurrent task, and vice-versa, to any desired depth. It is with these capabilities that a developer can build a complex control system with simple tasks.

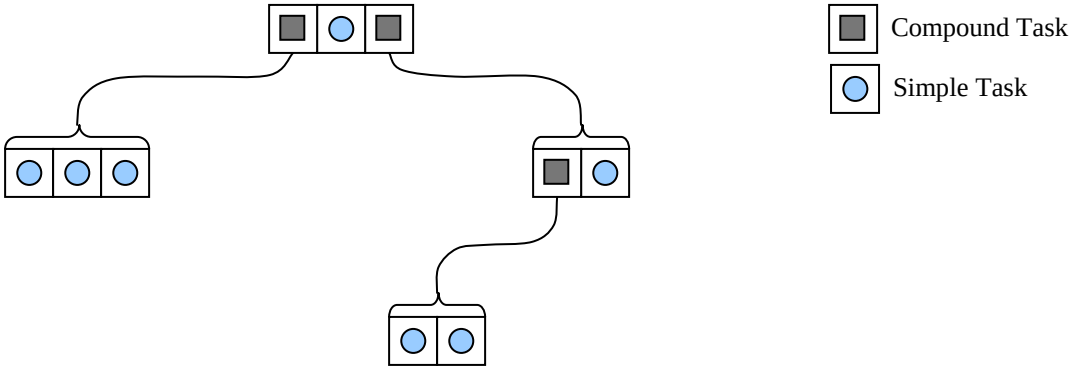

Figure 2: Nested Tasks

3.1 Interface

The base task interface is shared by all task subclasses, and implements the expected set of control methods in addition to the optional thread management routines.

```
public interface ITask {
   public void initialize(Task parent);
   public void start();
   public IAttributeList wait(int timeout);
   public void run();
   public void stop();
   public void pause();
   public void resume();
   public void done(IAttributeList result);
   public void registerCallback(ITaskCallback callback);

   protected IAttributeList execute();
}

public class Task implements ITask {
   public Task(ITaskExecutor executor);

   // Implementation of ITask interface.
   ...
}
```

The following methods are defined for the Task base class:

- *initialize*(parent): this method is used to initialize a task from the parent task (if any), immediately prior to execution. Because a task can be instantiated arbitrarily long before it is actually started, this method performs any dynamic initialization needed just prior to execution, based on current conditions. It is also used to reinitialize a task object if it is reused. Normally this is an inherited method that performs some initialization in the task infrastructure and does not need to be implemented or overridden by the subclass.
- *start*: a method of no parameters that starts the task executing that must return quickly (the intention being that it does not perform the task, but merely initiates it). The mechanics of how this works is dependent on the implementation of the execution environment.
- *wait* (timeout=None): a method with an optional timeout parameter. This waits for the executing task to finish and returns that task's result. If a timeout is passed (a float) then the caller will wait at most timeout seconds for the task to finish. If the task does not finish by that time a TimeoutError exception is raised. If no timeout is passed, then the caller will wait indefinitely for the task to finish. If an exception is raised by the child task, it will be re-raised in the parent on a wait.
- *run*: this is essentially a convenience function and as a combination of *start* and *wait*.
- *stop*: halt and cancel a task. The implementation of this method is optional and may not be appropriate for all tasks.
- *pause*: temporarily interrupt an executing task. The implementation of this method is optional and may not be appropriate for all tasks.
- *resume*: continue a paused task. Any task that implements the *pause* method must also implement *resume*.
- *registerCallback(callback)*: allows the user to define a callback object that will receive status information when the task completes. (See *ITaskCallback* interface below for more information on the callback signature.)
- *execute*: implements the task logic. This method is executed by the thread pool after the task has been started. The returned AttributeList should indicate the success or failure of the operation, and will automatically be forwarded to the *done* method for processing.

- *done* (result): when a task is ready to terminate normally, it must call this method internally with its result value (defined in an AttributeList). A task normally calls this as its final act. The method itself is usually inherited from the parent class.

The ITaskCallback interface defines a simple object to receive status information when the task completes through an IAttributeList instance.

```
public interface ITaskCallback {
   public void taskComplete(IAttributeList taskStatus);
}
```

As discussed earlier a set of compound task base classes will also be provided. These compound tasks are designed to manage the execution of multiple tasks in sequential or concurrent fashion.

```
public class SequentialTask extends Task {
   public SequentialTask(List<Task> tasks);

   public void step();
}
```

A SequentialTask simply iterates through the list of tasks provided, and executes each one in order. If any of the subtasks fail the sequential task will terminate and return error information through the wait method or a registered callback. The sequential task also adds a new method called *step* to the task interface. This method is used in conjunction with *pause* to allow the user to step through the execution of the sequence one task at a time. Calling *resume* will automatically return the sequential task to standard execution.

To perform task execution in parallel use the *ConcurrentTask* implementation.

```
public class ConcurrentTask extends Task implements ITaskCallback {
   public ConcurrentTask(List<Task> tasks);

   protected void taskComplete(IAttributeList taskStatus);
}
```

Concurrent tasks are designed to execute all of the provided tasks simultaneously, and then wait for each of the tasks to complete. The ConcurrentTask class implements the *ITaskCallback* interface to act as the callback for each of the supplied tasks. This will allow the task to monitor the status of each subtask and determine when to signal overall completion. The concurrent task will wait for all subtasks to complete, even if one or more fail.

3.2 Implementing Tasks

Application development should focus on the development of simple tasks. A simple task is one that is atomic in some respect. This definition is intentionally loose, but basically refers to implementing an activity that from the OCS's perspective cannot be broken down into simpler steps. Simple tasks might be used to implement a basic instrument or telescope command such as opening the shutter of a camera, or moving the telescope. Simple tasks form the basic building blocks of the set of activities provided by the application.

Simple tasks fall into one of two categories: asynchronous and synchronous. These correspond to typical patterns of command processing in distributed systems. An asynchronous task is one whose "business logic" is handled by some externally active subsystem (e.g. an "open shutter" command issued to an instrument control system). A synchronous task on the other hand is one whose logic is implemented in the task code itself (e.g. calculating a point spread function on an image region). Externally, the two types of tasks can operate the same way with non-blocking start and wait capabilities. The difference manifests itself internally in the implementation of the task methods, and which methods are overridden from the base class.

3.2.1 Asynchronous Task Development

An asynchronous task is one that relies entirely on external components to perform the system control. The task is responsible for invoking a command on the remote component(s), and then providing a mechanism to notify the user when the commands have completed. Development of asynchronous tasks focuses primarily on the *start* and *wait* method. The following pseudo-code outlines an asynchronous command.

```
class PointTelescope extends Task {

   public PointTelescope(IAttributeList args) {
         super(args);
   }

   public void initialize(Task parent) {
         // Connect to TCS controller.
         this.TCS = ...
   }

   public void start() {
         this.command = new CommandSet(this.params, “point”);
         try {
               this.TCS.execute(command, null);
         catch(...) {
               // Failed command execution.
         }
   }

   public IAttributeList wait(float timeout) {
         AttributeList res;
         try {
               res = this.TCS.wait(this.command, timeout);
               done(res);
         }
         catch(...) {
               // Set res to failed.
         }

         return res;
   }
}
```

This PointTelescope class only overrides the base class methods for *initialize*, *start* and *wait*. The class constructor will perform the required preparation for the task. The constructor arguments are passed up to the superclass to be saved in the *params* member of the base class. As an attribute list

these arguments can be passed directly to target components in the system, or processed for specific formatting prior to command execution.

In this simplified example we assume that the initialize method connects the task with the TCS controller. This provides the task with a handle for interfacing with the telescope control system. When the task is ready for execution the *start* method is called. This will create a command set from the class arguments (containing the pointing coordinates for the telescope), assign the action, and issue the call to the TCS controller. The call to *start* will then immediately return as required by the Command pattern. The application can then wait on the completion of the task by executing the *wait* method. In the wait method the task again interfaces to the TCS controller, in this case to wait for the completion of the command or the timeout (if one was specified - the default is 0). If an exception occurs the return argument will be properly formatted to indicate the call failed.

The important point here is not how the methods are implemented, but that they are implemented, and obey the proper behavior. *start* is always required to return "as soon as possible" after initiating the activity of a task. In asynchronous tasks this is not too onerous because (as shown) they are generally invoking an asynchronous command in some external subsystem and then returning. The *wait* method is a little more complicated, but basically needs to synchronize with the external subsystem for the end of that command, and return the result, if any. The timeout parameter complicates this effort, but it must be obeyed.

Asynchronous tasks are ideal for external or rapidly executed commands. As asynchronous tasks do not use the internal thread pool, developers must ensure that the application control does not block or suspend activity for any significant period. Using asynchronous tasks effectively can improve application performance by reducing system resource usage and the number of concurrently active threads.

3.2.2 Synchronous Task Development

A synchronous task is one where the business logic is implemented entirely within the task class, potentially with command and execution of external controllers as well. Unlike asynchronous tasks, synchronous tasks have the potential to block the application control for a substantial duration while the command logic is executed. As the Command pattern calls for rapid non-blocking method execution, *start* needs to be able to invoke that logic and yet return immediately, without significantly impacting application responsiveness.

The standard recipe for such a requirement would be to create an auxiliary thread to do the computation, which *start* would be responsible for initiating. The *wait* method could then wait on, or join, the thread and collect the result. However, requiring all synchronous-style tasks to perform this sort of thread management is tedious and quickly leads to various sorts of concurrent programming errors, in addition the overhead required to create and destroy threads regularly. This complexity cannot be eliminated completely, but the task framework does provide some help in the form of task executors. These classes implement an execution environment for tasks allowing the user to start and forget about the task. Executors are responsible for the lifecycle and asynchronous execution of tasks, allowing the developer to focus on the functional details of the task instead.

The base Task class is all set up to enable synchronous-style tasks, where the subclass only needs to provide an execute method. If a task subclass does not override it, the Task's *start* method simply adds the task instance (itself) to the desired task executor. When the resources are available to

process the task the executor will activate and execute the task. Similarly, if not overridden, *wait* understands how to listen and block for the result of a task and obey the timeout parameter.

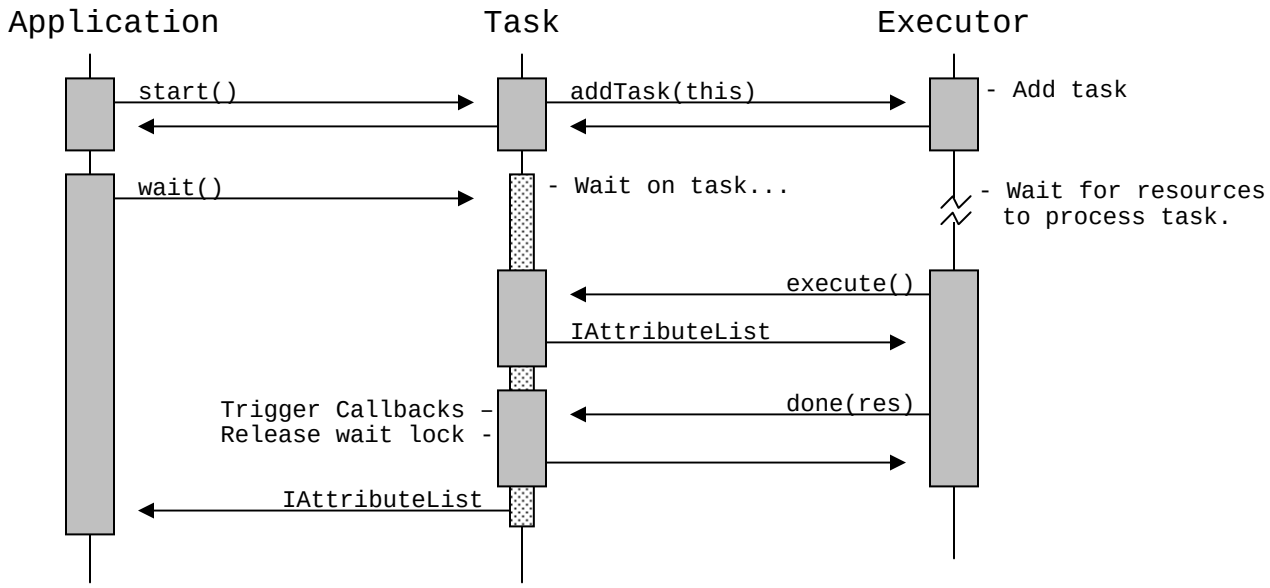

Figure 3: Task Execution

As a result of this design, implementing a synchronous-style task can be as simple as defining a single method. The following example pseudo code outlines the creation of a synchronous task.

```
class PointSpread extends Task {

   public PointSpread(IAttributeList params) {
         super(params);
   }

   public void initialize(Task parent) {
         // Connect to a camera
         this.camera = …
   }

   public IAttributeList execute() {
         IAttributeList res = new AttributeList();

         // Get image data…
         double [] image = this.camera.get(…);

         // Calculate point-spread function on image.
         …

         return res;
   }
}
```

This task inherits the *start* and *wait* methods from base Task, while overloading the *initialize* and *execute* methods to implement the task logic. By inheriting most of the default methods, synchronous simple tasks can clearly express the business logic of a task without much extraneous task-related detail cluttering up the code. The full range of the Java standard library is available, and

the task is free to define other methods to subdivide up the problem and make the program structure more manageable, provided they conform to the Command pattern and don't conflict with the task interface method names.

3.3 Task Executors

Task Executors are responsible for managing the execution and lifecycle of tasks. Typically an executor will be designed to queue or schedule tasks as they arrive, and when resources are available or an event occurs, the task will be removed and processed.

The following details the base task executor interface which all task executors must implement.

```
public interface ITaskExecutor {
   public boolean addTask(Task task);
   public boolean removeTask(Task task);
   public int pendingTasks();
}
```

The executor interface defines a set of methods to add and remove a task, as well as report the total number of tasks waiting to be executed. How the executor manages and executes tasks is up to the developer. The Task Library however, provides a thread pool executor implementation which may be satisfactory for most task processing requirements.

3.3.1 Command Thread Pool

The Task library command thread pool executor abstracts the implementation of a group of threads responsible for processing information from a shared queue. Hidden behind a class interface, there are methods for adding, modifying, and removing work objects from the queue. Threads compete to read items from the queue, process them, and iterate back to the queue.

The executor creates a thread pool object whose worker threads are all blocked waiting for available tasks from the queue. A worker thread will pick up a new task reference when it arrives, and try to invoke its execution method. By convention, the execute method does whatever work needs to be done and returns the result, which is stored away in the task object. The worker thread then returns its attention to the queue.

The conceptual model of a thread pool is simple: the pool starts threads running; work is queued to the pool; available threads execute the queued work. In our case all tasks will be handled identically because of the standardized interface.

In the thread pool pattern, a number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. Generally a thread pool allows a server to queue and perform work in the most efficient and scalable way possible. As soon as a thread completes its task, it will request the next task from the queue until all tasks have been completed. The thread can then terminate or sleep until there are new tasks available.

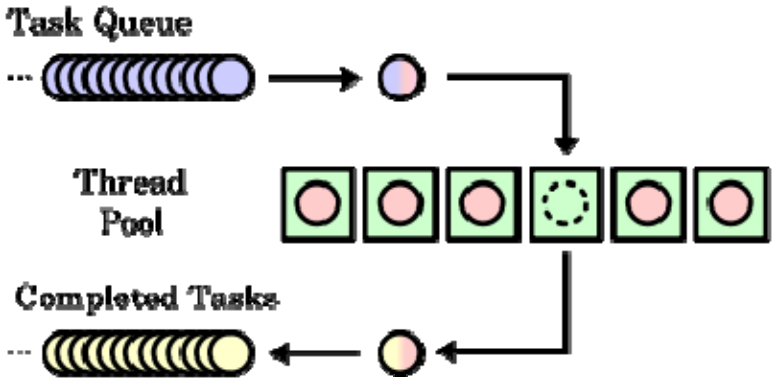

Figure 1: Simple Thread Pool

The number of threads used is a parameter that can be tuned to provide the best performance. Additionally, the number of threads can be dynamic, based on the number of waiting tasks. The cost of having a larger thread pool is increased resource usage. The advantage of using a thread pool over creating a new thread for each task is that thread creation and destruction overhead is negated, which may result in better performance and better system stability. As with any technique that utilizes threads, the task queue and task class must be developed with thread safety and read / write synchronization in mind.

Using a thread pool also allows us to develop additional capabilities to tune and control the execution of tasks in a control system. These include:

- Adding a priority to tasks
- Utilize thread priorities, per task or per group
- Removing a task before it has been dequeued
- Tasks can be scheduled for execution
- Compound commands can distribute tasks over multiple threads

The following pseudo-code details the CommandThreadPool executor interface.

```
public class CommandThreadPool implements ITaskExecutor {

   private class ThreadControl implements Runnable {
         public CommandHandler();
         public void run();
   }

   public CommandThreadPool(int poolSize);

   public boolean addTask(Task task);
   public boolean removeTask(Task task);
   public int pendingTasks();
   public void flushQueue();

   protected ExecutorService threadPool;
   protected BlockingQueue<Task> taskQueue;
}
```

The CommandThreadPool class makes use of the Java *ExecutorService* to create and manage an internal pool of threads of a desired size. Each of the threads created by the executor will be started in its own instance of the task processing control class, *ThreadControl*. This class implements the

individual thread logic to pop a task from the queue, call *execute*, and finally invoke the task's *done* method before returning for new tasks. The queue is implemented as a blocking queue, where thread access to items is synchronized and blocking is performed until items are available for processing.

The following methods are defined for the CommandThreadPool class:

- *addTask*: This method will add a new task to the command thread pool queue for processing. Typically this method will be called by the tasks themselves when *start* is executed. The method will return false if it is unable to add the task.
- *removeTask*: Allows the client to remove a task from the queue. This method can only be performed on tasks which have not already been removed for processing. If the task is not currently on the queue this method will return false.
- *pendingTasks*: Returns the total number of tasks waiting in the queue for processing.
- *flushQueue*: Removes all of the pending tasks.

3.4 Using Tasks

To realize an activity during an observation, one instantiates the task of the appropriate name with appropriate arguments (i.e. creates an object by calling the class constructor with a full populated Attribute List). For example, a PointTelescope task might be instantiated as,

```
IAttributeList params = new AttributeList();
params.set("ra", 185.0);
params.set("dec", 47.8);
params.set("equinox", 2000);

PointTelescope point = new PointTelescope(params);
```

Once a task has been created, the standard task interface is used to control it (i.e. by method calls on the object). For example, we might do the following to initialize the task, start it, and wait for the result, which is returned to the variable res.

```
point.initialize(null);
point.start();
IAttributeList res = point.wait();
```

Alternatively, one can combine the *start* and *wait* steps by using *run*.

```
point.initialize(null);
IAttributeList res = point.run();
```

The attribute list returned by the *wait* or *run* methods will contain status information for the task. The attribute list is guaranteed to contain the enumeration item "*_OperationResult*", which can have one of the permitted status values (SUCCESS or FAIL). In addition, if the task failed a string attribute, "*_Reason*", will be defined to give a human readable description as to the cause of the failure. Other custom attributes may be defined as required or provided by the task implementation.

3.5 Compound Tasks

Due to the standard interface provided by the task abstraction, it is possible to compose tasks straightforwardly. Tasks can be written that create and control other tasks - these kinds of tasks are compound task. Compound tasks share some basic attributes and methods with ordinary tasks, but also have other methods that may be unique.

3.5.1 Sequential

A common type of compound task is to execute a set of tasks in a given order (a sequence) running each one to completion before starting the next, and bailing out if there is an error along the way. This is known as a sequential compound task. Rather simplistically, the task can be implemented as follows.

```
class SequentialTask extends Task {

   public SequentialTask(List<Task> tasks) {
         super(null);
         this.taskList = tasks;
   }


   public IAttributeList execute() {
         IAttributeList res;
         for(int i = 0; i < this.taskList.length; i++) {
               this.taskList[i].initialize(this);
               this.taskList[i].start();
               res  = t.wait();

               if(res.getInt("_Status") == FAILED)
                     return res;

               if(this.taskPaused) {
                     synchronize(this.stepMutex) {
                           try {
                                 this.stepMutex.wait();
                           }
                           catch(...) {}
                     }
               }
         }

         // set res to success for Sequence task
         res = ...
         return res;
   }

   public void step() {
         if(this.taskPaused) {
               synchronize(this.stepMutex) {
                     try {
                           this.stepMutex.notify();
                     }
                     catch(...) { }
               }
         }
   }
```

```
}
```

Here is an example of how create a Sequential compound task:

```
List<Task> list = new List<Task>();
list.append(PointTelescope(…));
list.append(OpenShutter(…));
…

SequenceTask task = new SequenceTask(list);
task.initialize(null);
IAttributeList res = task.run();
```

A list of simple tasks is created and inserted into the list in the desired order of execution. A new sequential task is created passing in the list to the constructor call. The task is then initialized and run. The status of the task can be found in the Attribute List returned by the call to *run*.

3.5.2 Concurrent

A Concurrent compound task is designed to execute its subtasks in parallel. The concurrent task is designed to start all its subtasks and then collects the results as they trickle in, terminating itself when the last subtask finishes. This is possible due to another feature of the task framework: the ability to register a callback for the task's termination. The following details how aconcurrent task can be implemented.

```
class ConcurrentTask extends Task implements ITaskCallback {

   public ConcurrenTask(List<Task> tasks) {
         super(null);
      this.taskList = tasks;
      this.count = 0;
   }

   public IAttributeList execute() {
         this.count = this.taskList.length;
         for(int i = 0; i < this.taskList.length; i++)
               this.taskList[i].registerCallback(this);

         try {
               for(int i = 0; i < this.taskList.length; i++)
                     this.taskList[i].initialize(this);
                     this.taskList[i].start();

               synchronize(this.waitMutex) {
                     this.waitMutex.wait();
               }
         }
         catch(...) { }

         // Create summary status results
         IAttributeList res = ...

         return res;
   }
```

```
    protected void taskComplete(IAttributeList taskStatus) {
          try {
                this.count--;

                // Process taskStatus
                ...

                if(this.count <= 0) {
                      synchronize(this.waitMutex) {
                            this.waitMutex.notify();
                      }
                }
          }
          catch(...) {
                ...
          }
    }
}
```

Any data structures that are shared between concurrently executing tasks need to be protected via standard mutual exclusion techniques such as reentrant locks, semaphores, conditions, etc. Java's standard threading module provides a set of tools for this purpose.

3.6 Nested Tasks

With simple and compound tasks a developer will be able to create nested tasks to perform complex actions. Since all task implementations share the generic interface and follow the Command pattern we can compose compound tasks out of Sequential and Concurrent tasks.

```
List<Task> l1 = new List<Task>();
List<Task> l2 = new List<Task>();
List<Task> l3 = new List<Task>();

l1.append(PointTelescope(…));
l1.append(OpenShutter(…));

l2.append(InitCamera(…));
l2.append(Track(…));

ConcurrentTask c1 = new ConcurrentTask(l1);
SequentialTask s1 = new SequentialTask(l2);

l3.append(c1);
l3.append(s1);
l3.append(IdleSystem(…));

SequentialTask s2 = new SequentialTask(l3);
s2.initialize(null);
IAttributeList res = s2.run();
```

The key feature of the above pseudo-code is that developers will be able to create compound tasks out of other compound tasks. In this example a concurrent and sequential task are created, and appended to a list. A new sequential task is instantiated using said list as the task sequence. When 's2' is executed it will perform the standard initialize-start-wait command sequence on each of the

compound tasks, recursively executing the underlying simple tasks contained within each. The final return value is the overall status of all tasks within the hierarchy of commands.

3.7 Other Control Patterns

Based on the Command pattern principals outline so far, we can see how one could define various other common control structures as tasks: conditionals, iterators, etc. With only a few simple, general compound tasks one can create quite sophisticated control flows using these sorts of compositions. The advantage over simply using native control structures is that using the compound task approach abstracts the details of concurrency and leverages code reuse for uniform treatment of error handling, cancellation, logging, etc. However, one can always "fall back" to using Java's native control structures for particularly tricky control patterns. Regardless of which approach is used, all tasks have access to Java data structures and libraries.

Other possibilities include inserting delays between tasks, logging task invocations, gathering timing statistics on subtasks, uniform error handling of subtasks, command cancellation between subtasks, etc. By defining a class for sequencing and using it wherever possible, any bug fixes or enhancements to the class are propagated to all uses of sequences.

3.8 Handling Errors, Dynamic Task Control

The previous examples have all been somewhat simplified. In reality, tasks have to deal with issue like cancellation, error handling, and so forth. In crafting the task interface, we have tried to ensure that writing code to the interface still remains as object oriented as possible. The standard approach to error handling in Java is to make use of exceptions. Therefore the same applies to tasks: a task is considered to have succeeded unless it raises an exception (of course the return values of tasks can also be used and interpreted, if desired).

In a distributed command and control system it can be difficult to cancel or pause some commands, especially once released to an external subsystem. Nevertheless, it is good to provide a mechanism to do so for those tasks that could support it. The task interface defines a set of execution control methods that can be implemented by the developer to provide this extra level of control (*stop*, *pause*, *step*, and *resume*.) Depending on what a specific task is designed to do, some of these capabilities may not apply (for example, you can not step through a concurrent task sequence since everything runs in parallel). At the very least, if possible, all tasks should implement a technique to halt and cancel a command through the *stop* method.

References

NGAO Component API