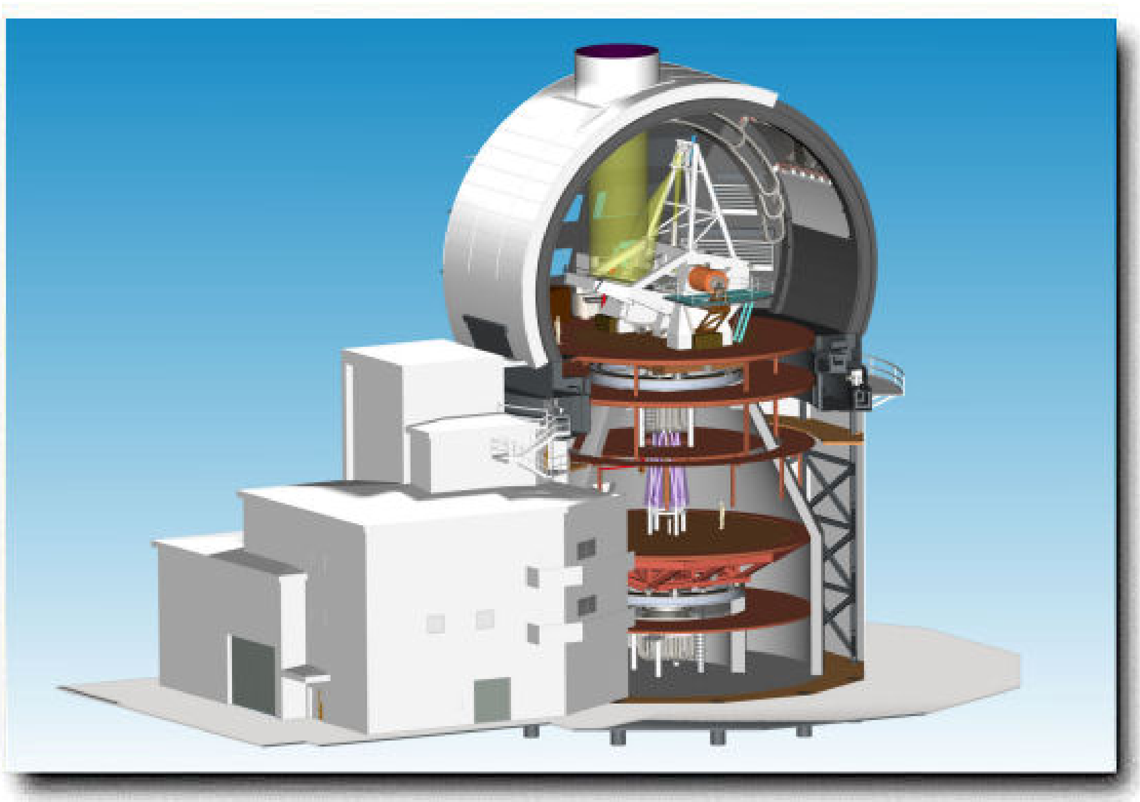
Project Documentation
SPEC-0022-1
Revision A-7

ATST Common Services Users' Manual



ATST Software Group

Panguitch-1P7
February 12, 2007
Tucson, Arizona

Advanced Technology Solar Telescope 950 N. Cherry Ave Tucson, AZ 85719
Phone 520-318-8102 atst@nso.edu http://atst.nso.edu Fax 520-318-8500

-- SteveWampler - 1 Feb 2006

to top

Table of Contents

Table of Contents

ATST Common Services
Software Design Document

ATST Common ServicesSoftware Design Document

1 Introduction

1.1 Overview

1.1.1 Background

Early in the conceptual design process ATST undertook a survey of observatory software control systems to determine the best approach to take on software design and implementation. A great deal of useful information was obtained as a result of this survey, one of which is that large, distributed, software projects can reduce their overall development, integration, and maintenance costs by basing as much software as possible on a standard infrastructure.

There are several viable models for this infrastructure in use in modern observatory systems. ATST has elected to use a *Common Services* model similar to that used for the ALMA project Common Software (ACS). The ATST Common Services (ATSTCS) attempts to be more streamlined than the ACS and also less dependent on a specific *middleware* structure. This approach should allow the fundamental characteristics of ATSTCS to be preserved as new middleware technologies are developed during the operating lifetime of ATST.

The benefits of a common services model for infrastructure include:

- All major system services are provided through standard interfaces used by all software packages. A small support team can thus support a number of development teams more easily.

- The separation of the functional and technical architectures provided by the common services model means that a significant amount of the technical architecture can be provided through the common services, allowing developers to concentrate on providing the functional behavior required of their software.

- There is a uniform implementation of the technical architecture across all systems. So long as the access to this technical architecture remains consistent, the implementation of the technical architecture can be modified with minimal impact on the development teams.

- Since application deployment is a technical issue and hence implemented within the common services, the software system as a whole is more easily managed within a distributed environment. This makes the use of less expensive, commodity computers more feasible.

Another infrastructure approach is a controls model that combines the communications and controls aspects of observatory control. Two illustrations of this model are LabVIEW, used by SOAR, and EPICS, used by Gemini, JACH, and many particle physics accelerators. Both of these control systems provide a rich development environment and are well-suited for real-time control systems.

1.1.2 Structure

The ATST Common Services are grouped into several broad categories:

- *Deployment support* – implemented based on a *Container/Component Model,* this support allows the uniform management of applications in a distributed system without regard to the functionality they provide. Base implementations for software *components* and *controllers* are provided as part of the deployment support. ***All application functionality is implemented on top of these base implementations.***

- *Communications support* – services that are necessary or useful in a distributed system. These include:

    - *Connection services* that allow applications to communicate directly with other applications, including commanding them to perform specific actions
    - *Notification services* that allow applications to publish/subscribe to broadcast messages (*events*) without explicit knowledge of their recipients/publishers
    - *Logging services* that allow applications to record historically useful information
    - *Alarm services* that allow applications to broadcast alarm and health messages.

- *Persistence support* – services that allow applications to store and retrieve property information whose lifetimes exceed that of a single instance of the application.

- *Tools* – libraries of software modules that are of common use across multiple system packages.

- *Application support* – support for writing ATST applications. The base implementation (i.e. Component) provides the connection framework to Common Services. An extension (i.e. Controller) handles multiple, simultaneous configurations in a *Command/Action/Response* model. Either may be extended by developers to add specific functionality and subclasses are already provided to assist in *sequencing of actions* and *real-time device control*.

All common services are available for use in three languages: Java, C++, and Python although the access to the services varies with the language.

1.1.3 Design Highlights

Most of the design of the ATST Common Services is of little interest to software development teams using the common services. However, a quick look at some of the key design features can be informative and also help illustrate some of the power and flexibility provided by ATSTCS. Detailed information on the *use* of these, and other common services features, can be found in later sections of this document.

1.1.3.1 Communications-Neutral Architecture

While ATST has selected ICE as the communications middleware that is the foundation for intra-application communications, ATSTCS is designed to operate as independently as possible from the choice of *communication middleware*. The role of third-party middleware is carefully defined and bounded. This allows ATST to remain flexible on its choice of middleware, such as ICE or CORBA, and to more easily replace one choice with another should it prove advantageous to do so in the future. Component developers should not be concerned with the choice of communications middleware - they reference no middleware-specific features, extend no middleware-specific classes, etc.

1.1.3.2 Separation of Functional and Technical behavior

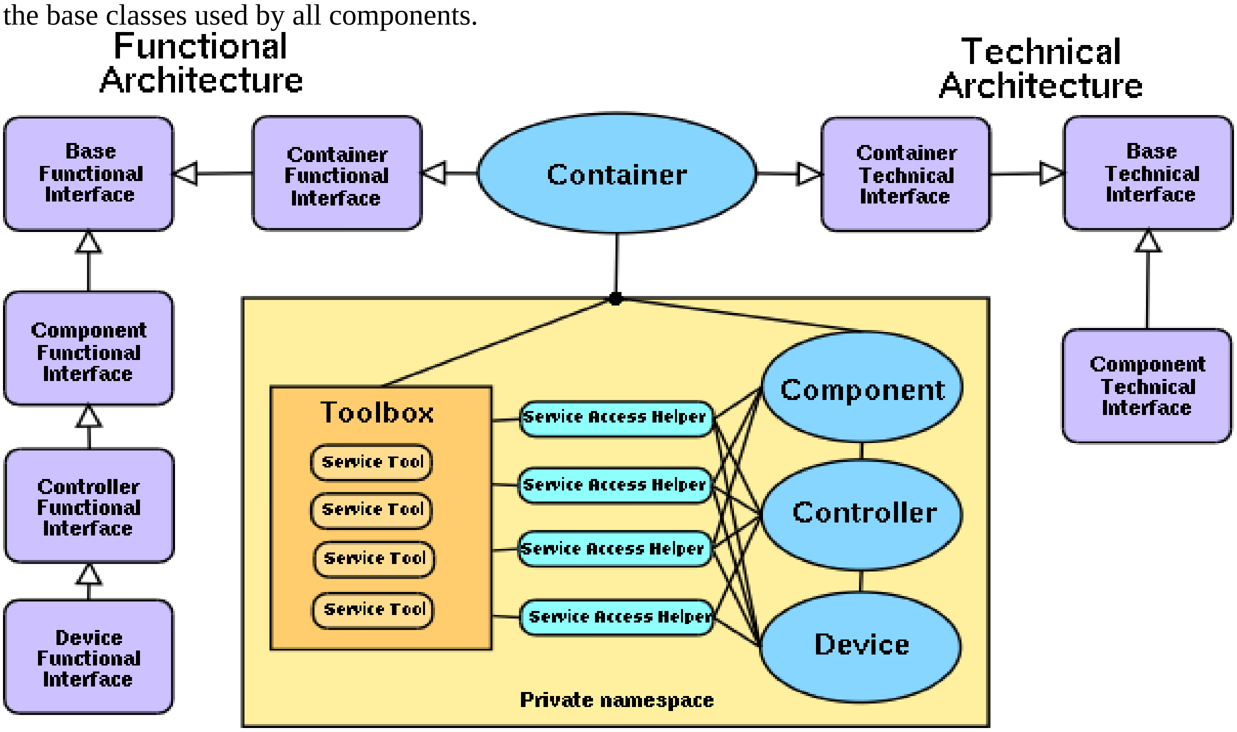
The ATST software design distinguishes between *functional* and *technical* behavior. Functional behavior describes the actions taken to directly implement ATST operations and can be contrasted with the technical behavior - the actions required of the infrastructure needed to support the functional behavior. For example, logging a specific message into a persistent store is functional behavior - only the application developer can determine what (and when) messages should be logged. The underlying mechanism that performs the logging, however, is technical behavior. By establishing a clear distinction between functional and technical behavior, and providing the technical behavior through the ATST common services, the application developer can concentrate on providing the required functionality.

1.1.3.3 Configuration-Driven Control

A fundamental precept of the ATST software design is the use of *configurations* to drive ATST control behavior. A configuration is a set of logically-related, named values the describe the target condition of a subsystem. Control of a subsystem is accomplished by directing the subsystem to *match* the target conditions described by each configuration. The set of available commands is thus kept small and generic - amounting to little more than "match this configuration". Subsystems are responsible for determining how to match the target - all details of sequencing subsystem components are isolated in the subsystem. Subsystems announce that they have met (or *cannot meet!*) the target using broadcast *events*.

1.1.3.4 Container/Component Model

One feature of ATSTCS is its adoption and support of a Container/Component Model (CCM). This approach, also used in the ALMA common services, is based upon the same fundamental design principles as Microsoft's .NET and Java's EJB architectures and simplifies application deployment and execution within a distributed environment. In the CCM, the deployment and lifecycle aspects of an application are separated from the functional aspects of the application. In particular, management applications (*containers*) are responsible for creating, starting, and stopping one or more functional applications (*components*). Containers are implemented as part of the common services, as are the base classes used by all components.



The *lifecycle* interfaces (appearing on the right of the above diagram) are implemented within the common services design by the underlying infrastructure. This means that developers can concentrate on providing code for performing actions visible through the *functional* interfaces (on the left of the above diagram). In the vast majority of cases this means subclassing the **Controller** class, overwriting or implementing any interface methods that access added functionality, and then writing support methods implementing that added functionality.

Components

Components are the foundation for all ATST applications as all ATST *functional behavior* is implemented within Component subclasses such as *Controllers*. (A Controller adds configuration management support to a Component.) The bulk of the ATST software effort is in designing and implementing the functionality provided by Components and

Controllers.

Containers

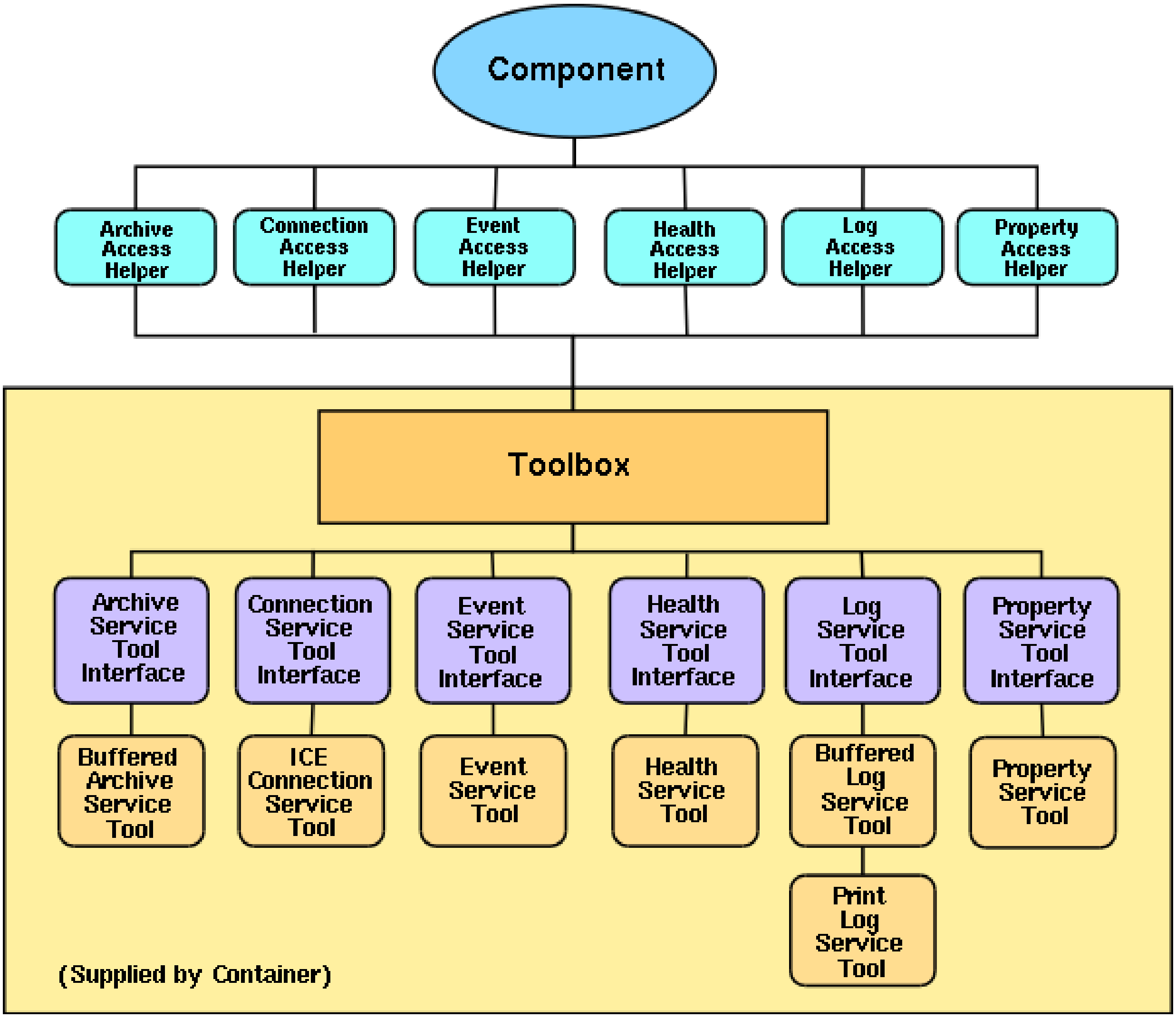
Containers provide a uniform means of deploying and controlling the technical aspects of Component operations. Component developers can develop Components without a detailed understanding of Containers.

1.1.3.5 Service Toolboxes

In ATST, access to services is provided to components through the container. Some services may be shared among components while others might be unique to individual components. It is the container's responsibility to ensure that services are properly allocated. When a component is deployed to a container, that container assigns a unique service *toolbox* to that component, and places *service tools* into that toolbox. Service tools are modules that understand how to access specific common services. Typically, these tools are are small, with well-defined tasks. However, the tools are designed to be *chained* so that several simple tools can be used to perform complex actions on service access. For example, message logging may be accomplished by chaining a log database tool that logs messages to a persistent store with a filter tool that looks for specific message characteristics. When a message with those characteristics is found, the filter tool might route that message to an operator's console. As a more extreme example (though not one likely to be used in ATST), it is possible to chain a connection service tool for ICE with a connection service tool for CORBA, allowing components to connect seamlessly and simultaneously to both ICE-aware and CORBA-aware modules! A container also retains access to each component's toolbox, permitting dynamic reconfiguration of tools without involving the component itself.

An important characteristic of the toolbox and service tools is that *all* component specific information needed by the various service tools is maintained in the toolbox, not in the specific service tool. This allows toolboxes to contain service tools that can be *shared* among components if it is advantageous to do so. For example, message logging may be more efficient if a common logging tool is shared among all the components within a container. It also makes it possible for Containers to retain access to the service tools assigned to a Component, adjusting the services as needed.

While a component is free to directly access the service tools in its toolbox, by far the most common way to access services is through static *service access helper* classes that are also provided by common services. These classes encapsulate access to the toolbox and its tools within easy to use static methods. It is this access through these access helpers that is discussed in detail in later sections of this document. *Direct access to service tools and the toolbox is intentionally not covered*.



-- SteveWampler - 12 Jan 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.25 | > | r1.24 | > | r1.23 | Total page history | Backlinks
Main.AtstCsSDDOverview moved from Main.AtstCsManualOverview on 18 Feb 2005 - 16:57 by SteveWampler - put it back
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsSDDOverview

to top

Skip to topic | Skip to bottom

Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
    - News
    - Design
    - Common Services
    - Application base
    - OCS
    - DHS
    - TCS
    - ICS
    - ICDs
    - Docs
    - Meeting Notes
    - Problem Tracking
    - Changes

- **Misc**
    - Users
    - Groups
    - Offices
    - Topic list
    - Search

- **TWiki Webs**
    - Know
    - Main
    - Sandbox
    - TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsManualCommunicationsr1.24 - 25 Aug 2006 - 21:18 - JanetTvedttopic end

---

Start of topic | Skip to actions

2 Infrastructure

2.1 Communications

The ATST software system is very much a *distributable* system. This means that the communications infrastructure plays a critical role and is provided as part of the ATST Common Services. Much of the implementation of the communications infrastructure is based upon services and support features found in third-party communications middleware packages. However, the common services isolates the dependence on third-party middleware from the rest of the ATST software system so that replacing the middleware is always a viable option.

Some of the key features of the ATSTCS communications infrastructure are:

- middleware isolation -- as mentioned above, ATSTCS isolates the third-party communications middleware from the rest of the ATST software.

- multiple communication methods:
  - ♦ peer-to-peer *command* messaging allows arbitrarily complex messages to be sent directly from one application to another.
  - ♦ publish-subscribe *event* messaging allows generating and receiving messages by applications without regard to the intended message recipients or sources.
  - ♦ bulk data *streaming* allows large data sets to be routed efficiently.

- simple connection support -- all that is needed for an application to establish a peer-to-peer connection to another application is the *name* of the target application. The communications infrastructure will locate the target application (*possibly* starting it if it isn't running).

- heartbeat monitoring -- applications are watched and an alarm is raised if an application unexpectedly stops responding.

2.1.1 Key Data Structures

All messages passed between applications share a common structure based upon standard ATST-defined data types.

2.1.1.1 Attributes

Attributes are the atomic representation of an data item that can be communicated between ATST applications outside of the bulk data transport. An Attribute is a (name,value) pair, where the *value* field is a sequence of strings. There is no limit to the size of this sequence nor to the length of the individual strings. In practice, most attribute values have a single string field within the sequence. ATSTCS provides support for converting between the strings used within Attributes to hold values and the basic data types used by ATST.

While ATSTCS itself imposes no restrictions on an Attribute's name field, ATST requires that all Attributes used in interapplication communication be uniquely named. The convention is to use the hierarchical structure of the ATST software element producing *fully-qualified Attribute names*, e.g. **`atst.tcs.ecs.az.pos`** names the enclosure azimuth position attribute used by the Enclosure Control System.

The common services provides a limited form of automatic validity checking on Attribute values. Associated with every Attribute is a *metadata* description of that attribute. This metadata is maintained in a persistent store by ATSTCS and can be accessed by an application for use in validity checking and type conversions. See the Property Service for details.

2.1.1.1.1 Java representation

The class **`atst.cs.data.Attribute`** defines an Attribute. Most references, however, use the matching *interface* **`atst.cs.interfaces.IAttribute`** with the following methods:

- **`String getName()`** -- produce the Attribute's name
- **`void setName(String name)`** -- set the name of the Attribute
- **`String[] getValue()`** -- produce the Attribute's value
- **`void setValue(String[] newValue)`** -- set the Attribute's value

2.1.1.1.2 C++ representation

The class **`atst.cs.data.Attribute`** defines an Attribute. To create an Attribute, use one of the following factory methods:

```
tr1::shared_ptr<Attribute> Attribute::create();
tr1::shared_ptr<Attribute> Attribute::create(const string& name,
const vector<string>& value);
tr1::shared_ptr<Attribute> Attribute::create(const string& name,
const string& value);
tr1::shared_ptr<Attribute> Attribute::create(const string& name,
long value);
tr1::shared_ptr<Attribute> Attribute::create(const string& name,
double value);
```

Since *smart pointers* are required in many of the ATST C++ interfaces, the return value of each create method is a smart pointer. See the C++ Implementation Notes for additional information on the use of the tr1::shared_ptr.

Most references to Attributes, however, use the matching *interface* **`atst.cs.interfaces.IAttribute`** which contains the following basic methods:

```
// Set(change) the name of the attribute
void setName(const string& newName);

// Get the name of the attribute
string getName() const;

// Set the value of the attribute
void setValue(const vector<string>& newValue);

// Return the value of the attribute as a container of strings (default)
vector<string> getValue() const;
```

Additional methods are provided for convenience, including:

```
// Set the name of the definition for this attribute
void setDefinition(const string& entryName);

// Return the name of the definition for this attribute
string getDefinition() const;

// Return the value of the attribute as a single string value
```

```
// delimited by comma(s)
string getValueList() const;

// Display the attribute in pretty form
void show(const string& s) const;

// Return a string representation of the Attribute
string toString() const;
```

2.1.1.1.3 Python representation

TBD

2.1.1.2 Attribute Tables

Attributes can be grouped into *sets of Attributes*, called *AttributeTables*. AttributeTables are *unordered*, but can be searched efficiently. Note that this is a *set* and not a *multiset*: at most one instance of an Attribute with a given name may exist within an AttributeTable. Inserting a new Attribute with the same name *replaces* the old one.

In ATST, the most common subclass of AttributeTable is the *Configuration*.

2.1.1.2.1 Java representation

The class **`atst.cs.data.AttributeTable`** defines an AttributeTable. Most references, however, use the matching *interface* **`atst.cs.interfaces.IAttributeTable`** with the following basic methods:

- **`boolean contains(String attributeName)`** -- does the table include this Attribute?
- **`void insert(atst.cs.interfaces.IAttribute attribute)`** -- insert Attribute
- **`atst.cs.interfaces.IAttribute remove(String attributeName)`** -- remove (and return) an Attribute
- **`atst.cs.interfaces.IAttribute get(String attributeName)`** -- produce an Attribute

Additional methods are provided for convenience, including:

- **`int size()`** -- number of Attributes in table
- **`String[] getNames()`** -- produce the names of all Attributes
- **`atst.cs.interfaces.IAttributeTable extractOnPrefix(String prefix)`** -- all Attributes with names sharing a common prefix
- **`atst.cs.interfaces.IAttributeTable extractOnSuffix(String suffix)`** -- all Attributes with names sharing a common suffix
- **`void merge(atst.cs.interfaces.IAttributeTable aTable)`** -- insert all elements in **`aTable`**

2.1.1.2.2 C++ representation

The class **`atst.cs.data.AttributeTable`** defines an AttributeTable. Use the following factory methods for creating AttributeTables:

```
// Create a new AttributeTable
```

```
tr1::shared_ptr<AttributeTable> AttributeTable::create();

// Create an IAttributeTable from an existing one
tr1::shared_ptr<IAttributeTable>
AttributeTable::clone(tr1::shared_ptr<IAttributeTable> table);

// Create an AttributeTable from an existing one
tr1::shared_ptr<AttributeTable>
AttributeTable[2]::clone(tr1::shared_ptr<AttributeTable> table);
```

Most references to AttributeTables, however, use the matching *interface* **`atst.cs.interfaces.IAttributeTable`** with the following basic methods:

```
// Does the table include this Attribute?
bool contains(const string& attributeName) const;

// Insert a new Attribute into the table
bool insert(tr1::shared_ptr<IAttribute> newAttribute);

// Remove (delete) an Attribute
bool remove(const string& attributeName);

// Retrieve (and return) an Attribute
IAttribute* get(const string& attributeName);
```

Additional methods are provided for convenience, including:

```
// Return the number of Attributes in the table
int size() const;

// Return the names of all Attributes
vector<string> getNames() const;

// Return all Attributes with names sharing a common prefix
tr1::shared_ptr<IAttributeTable> extractOnPrefix(const string prefix) const;

// Return all Attributes with names sharing a common suffix
tr1::shared_ptr<IAttributeTable> extractOnSuffix(const string suffix) const;

// Insert all elements from source table
void merge(const IAttributeTable* source);

// Return stringified table
string toString() const;

// Display table to standard output
void displayAttributes() const;

// Display table to standard output with heading
void show(const string heading) const;
```

2.1.1.2.3 Python representation

TBD

2.1.1.3 Configurations

While Attributes are the atomic unit of ATSTCS communications, the fundamental data structure is the *Configuration*, an extension of `AttributeTable`. Configurations are used in ATST commands to describe the conditions that must be met by the target Component to satisfy the command. Components report back their success, or *failure*, in matching those conditions.

Configurations used with directives and Configurations transmitted as the values of events may include a few standard Attributes. The values of these attributes are managed by ATSTCS, but Component developers must avoid name collisions with these standard Attributes. At the current time, only two standard Attributes are defined:

- *headerTag* -- this Attribute identifies the data set that is associated with the values found in this configuration. Typically, the value of *headerTag* is the ID of the observation that is currently being processed. It is used, among other things, to determine if a Component needs to respond to a specific *getHeaderData* event.
- *configId* -- every Configuration maintains a unique identification.

The act of matching a Component's condition to a Configuration drives the Configuration through a one-way sequence of *configuration lifecycle* stages. These stages are:

- *initialized* - the configuration has been created, but not yet acted on.
- *running* - the configuration is in the process of being matched.
- *done* - the matching process has complete (successfully or unsuccessfully).

When a Component cannot match a configuration, that configuration is *aborted* as unsuccessful. A Component that is matching a configuration can be aborted by an external directive or by some internal condition within the Component that prevents a successful match. Details can be found in the Controllers section.

2.1.1.3.1 Java representation

The class **`atst.cs.data.Configuration`** defines a Configuration. Most references, however, use the matching *interface* **`atst.cs.interfaces.IConfiguration`**. Component developers should restrict their use of the **`IConfiguration`** interfaces to those methods inherited from **`atst.cs.interfaces.IAttributeTable`**; the methods added to the **`IConfiguration`** interface are for internal use by the ATSTCS.

2.1.1.3.2 C++ representation

TBD

2.1.1.3.3 Python representation

TBD

2.1.2 Commands and Events

The two fundamental mechanisms for communication between ATST Components are *Commands* and *Events*. Commands are used in peer-to-peer communication while Events are used in publish/subscribe communication.

2.1.2.1 Commands

There are two basic classes of commands used in ATST:

- Lifecycle commands -- commands used by ATST system management to control the *lifecycle* characteristics of applications. Users generally do not need to be concerned with the lifecycle commands because they are implemented by the underlying ATST infrastructure. They are introduced, however, in the section on Containers.
- Functional commands -- commands that implement the specific functional characteristics of a Component. Because the ATST uses configurations and a narrow command interface, the number of functional commands are quite small.

ATST functional operation is based on the *Command/Action/Response* model that isolates the transmission of the command from the resulting action that is performed. When an application receives a command, it validates any Configuration associated with that command and immediately *accepts* or *rejects* the command. If the command is accepted, the application then initiates an independent internal action to meet the conditions imposed by the command. Once those conditions have been met, an *event* is posted signifying the successful completion of the action (or the unsuccessful completion if the conditions can not be met).

The functional commands depend upon the type of each Component and are covered in detail in the sections on Components and Controllers.

2.1.2.2 Events

Events are the basis of the publish/subscribe communications system provided by ATSTCS. Any application may post events and/or subscribe to events posted elsewhere. The ATSTCS event service is robust and high performance.

An event consists of a **name** and a **value**. A sequence of events with the same name is referred to as an *event stream*. The name of the event is used to identify the event stream to subscribers. The value is an arbitrary Configuration. The convention is that all events in a stream share the same Configuration structure (i.e. the same Attribute names within the Configuration).

The event service has the following general properties:

- An event stream represents a many-to-many mapping: events may be posted into the stream from more than one *source* and received by zero or more *targets*. (Typically, however, most event streams will have a single source.)

- Events posted by a single source into an event stream are received by all targets in the same order as they were posted.

- Delivery of events to one subscriber cannot be blocked by the actions of another subscriber.

- An event stream is an abstract concept: a subscriber may subscribe to an event stream using a *wildcarded* name in which case the event stream it receives is the merging of all published event whose names match that wildcarded name.

- Events are not queued by the service. A "late" subscriber will not see earlier events.

- The event service does not *drop* events. A published event will be delivered to all subscribers.

- The event service supports arbitrary event names. However, ATST itself imposes a *hierarchical* naming convention on event names.

- Events are *automatically* tagged with the source and a timestamp.

See the Event Service section for details.

-- SteveWampler - 18 Jan 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.24 | ≥ | r1.23 | ≥ | r1.22 | Total page history | Backlinks
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsManual > AtstCsManualCommunications

to top

Skip to topic | Skip to bottom

Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
  - News
  - Design
  - Common Services
  - Application base
  - OCS
  - DHS
  - TCS
  - ICS
  - ICDs
  - Docs
  - Meeting Notes
  - Problem Tracking
  - Changes

- **Misc**
    - Users
    - Groups
    - Offices
    - Topic list
    - Search

- **TWiki Webs**
    - Know
    - Main
    - Sandbox
    - TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsManualContainersr1.12 - 21 Aug 2006 - 21:56 - JanetTvedttopic end

Start of topic | Skip to actions

2.2 Containers

ATST software is based on the *Container/Component Model*. In this model, one or more Components are deployed into each Container. There are separate Containers for Java, C++, and Python Components. Typically there is one Container per host but there is no requirement restricting Containers in this fashion.

Containers are responsible for managing the *lifecycle* characteristics of any Component that they contain — they create the Component, start it running, shut it down, and remove it from the system. This approach means that there is a uniform method for managing Components across the ATST system. Furthermore, Component developers may safely ignore the majority of lifecycle characteristics and can focus on the development of the specific functionality required of each Component.

ATST Containers also provide each Component with access to the services. A Container creates a separate *ToolBox* for each Component and then populates that ToolBox with the *service tools* that provide access to the common services. The ToolBox allows the container to retain access to the service tools so the Container can, if needed, dynamically adjust this access (for example, the service tool supporting event receipt by a subscribing Component could be adjusted to log received events during debugging). ToolBoxes also provide a common location for service tools to hold commonly-used information about the Component.

When a Container creates a Component, it provides that Component with a *private* namespace that is separate from that of other Components in that Container; Component developers do not have to be concerned about namespace collisions with other Components. Service tools, on the other hand, may be loaded either within each Component's private namespace (i.e. a separate copy of the service tool is needed for each Component) or in a *shared* namespace (i.e. one service tool is used by all Components in the Container). The Container determines which service tools are private and which are shared. The choice is completely transparent to the Components.

To be managed by a Container, all Components must adhere to a common standard. This standard is implemented by the base Component class and is discussed in detail in the section on Components. A quick overview is presented below.

2.2.1 How a Container manages a Component

While the details aren't particularly important to Component developers, it is instructive to see the major steps that a Container takes when managing a Component's lifecycle.

2.2.1.1 Deploying a Component

The Container takes the following steps when asked by a Container Manager to deploy a Component:

1. The Component is instantiated by invoking its *default* constructor in a new namespace.
2. A ToolBox is created, populated with tools, and attached to the Component.
3. The **`init`** method is called on the Component

Note that the functional operation of the Component is not started as part of the deployment. This allows multiple Components to be deployed and configured before any are started operating. Also note that access to the common services is **not available** to the default constructor. For this reason, the default constructor for a Component does as little as possible, deferring any initialization actions until the **`init`** method (when common services access is available). The expectation is that the Component's **`init`** method is a *software-only* operation - no mechanisms should move as a result of calling **`init`**. Furthermore, the Component is not assumed to be ready to operate upon completion of the **`init`**.

2.2.1.2 Starting a Component

At some point after deployment of a Component, the Container Manager instructs the Container to start the Component running. The Container does so by invoking the Component's **`startup`** method. At this point the Component is considered *functionally* active and ready to accept commands.

2.2.1.3 Stopping a Component

When a Container Manager instructs a Container to do so, the Container may stop the functional operation of a Component by invoking the Component's **`shutdown`** method. The **`shutdown`** method is the logical inverse of the **`startup`** method. Once a Component has been shutdown, it can only be restarted with the **`startup`** method, or removed from the system.

2.2.1.4 Removing a Component

The following major steps are taken when a Component is to be removed from the system:

1. The Container verifies that the Component has been shutdown.
2. The Container accesses the Component's ToolBox and releases all service tools.
3. The Container removes the Component's instance.

-- SteveWampler - 26 Jan 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.12 | ≥ | r1.11 | ≥ | r1.10 | Total page history | Backlinks
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsManual > AtstCsManualContainers

to top

Skip to topic | Skip to bottom

Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
    - News
    - Design
    - Common Services
    - Application base
    - OCS
    - DHS
    - TCS
    - ICS
    - ICDs
    - Docs
    - Meeting Notes
    - Problem Tracking
    - Changes

- **Misc**
    - Users
    - Groups
    - Offices
    - Topic list
    - Search

- **TWiki Webs**

- Know
- Main
- Sandbox
- TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsManualServicesr1.7 - 22 Aug 2006 - 20:52 - JanetTvedttopic end

---

Start of topic | Skip to actions

---

3 Services

3.1 Introduction to the Services

The services provided by ATSTCS can be loosely divided into two categories:

- The **major** services (also simply referred to simply as *services*) provide functionality that is required for the successful operation of *any* Component. The major services are:
    - The connection service provides support for connecting Components and Controllers with each other including uniform access to ATST's *Command/Action/Response* system.
    - The event service provides access to ATST's publish/subscribe communications system.
    - The logging service provides a standard means of logging many types of messages.
    - The health service provides uniform handling of system error conditions.
- The **minor** services (all referred to as *tools*) provide support useful for implementing functionality within Components. They are also used internally to implement the major services. The minor services are discussed in the section on Tools. (Tools in this context should not be confused with *service helper tools* used to implement many services found in a Component's ToolBox.)

Access to many of the services are implemented by chaining small *service helper tools* together to perform compound actions. Furthermore, these chains of service tools can be dynamically reconfigured by a Container during system operation. Consequently, the actions performed in practice for a given Component may differ from the *normal* behaviors described below. However, the actual action is *always* an extension of the normal action.

Access to both services and tools by Components is simplified through the use of *helper* classes that mask most of the details of the services from the Component developer. These helper classes are covered in detail in the following sections. The actual implementation of service access using the ToolBox and service helper tools is deliberately *not covered*.

The following sections present introductions to the use of each of the services. The full documentation on the service APIs (and the rest of the ATST software) is found at the ATST source code documentation.

-- SteveWampler - 27 Jan 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.7 | ≥ | r1.6 | ≥ | r1.5 | Total page history | Backlinks
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsManual > AtstCsManualServices

to top

Skip to topic | Skip to bottom

Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
    - News
    - Design
    - Common Services
    - Application base
    - OCS
    - DHS
    - TCS
    - ICS
    - ICDs
    - Docs
    - Meeting Notes
    - Problem Tracking
    - Changes

- **Misc**
    - Users
    - Groups
    - Offices
    - Topic list
    - Search

- **TWiki Webs**
    - Know
    - Main
    - Sandbox
    - TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsManualConnectionservicer1.26 - 22 Aug 2006 - 20:55 - JanetTvedttopic end

Start of topic | Skip to actions

3.2 Connection Service

The connection service supports Component registration with the ATST communications system to allow other Components direct *peer-to-peer* access. The service also supports the connection to other Components. The following basic actions are provided by the connection service:

- Registration -- the current Component is registered by name with the ATST communications system.
- Deregistration -- the current Component's registration is removed.
- Connection -- the current (*source*) Component connects to another (*target*) Component using the name of the target.
- Disconnection -- the current Component's connection to the named target is removed.

Registering and unregistering Components are handled automatically by the ATST common services. Component developers do not need to perform these two actions in their code.

3.2.1 Commands

Once a connection has been made by a source Component to a target Component, the source Component may issue *commands* to the target Component. (*Whether or not those commands are accepted by the target Component depends upon the* ***access policies*** *in effect.*) The specific commands available on the target Component depend upon the interface selected as part of the connection to the target. The available interfaces are:

- **`IComponent`** - a simple target Component
- **`IController`** - a target Controller (a subclass of **`Component`**)

These interfaces are selected by *downcasting* the object produced by connecting to a target Component. Only legal downcasts are allowed, for example, a non-Controller target connection can not be downcast to **`IController`**.

Commands from unauthorized sources (i.e. commands originating from sources that do not have permission to control the targeted Component) are ignored. Only ATST's Observatory Control System can authorize sources. The dropping of unauthorized commands is handled automatically by ATSTCS - a target component does not see unauthorized commands.

All commands are automatically tagged with a *header tag*. This tag identifies that the command is a result of some behavior driven by a specific observation. The command system remembers this tag and compares it with subsequent *header collection events* to determine if the event needs a response from the Component.

Commands return immediately but the actions that are initiated as a result of a command may take some time to complete. When the action completes, an *action status event* is posted that includes the completion status of that action. The component generating the command monitors this status event prior to issuing the command on the remote system. While the monitoring is performed automatically by the command system, Component developers may need to attach a *callback* to perform processing on action completion. This callback may be **`null`** if no processing is needed.

The status event's name is prefixed with the name of the Component posting the event, as is the name of the *status Attribute* contained in that event's value. For example, if the Controller **atst.tcs.mcs** posts an action status event, the name of that event is **atst.tcs.mcs.status** which matches the name of the action status attribute within the event's value.

The details of the available commands for each of these interfaces are language specific and covered in the appropriate *helper* section below.

3.2.2 Java helper

The class **atst.cs.services.App** provides Java-based Components with access to the Connection service. The static methods involved are:

- **atst.cs.interfaces.IRemote App.connect(String targetComponentName)**
- **void App.disconnect(String targetComponentName)**

Note that **App.connect** returns an **IRemote**. The object that is returned can be *downcast* to the same correct interface. So, for example if **atst.tcs.mcs** is the name of a Controller registered with with the naming service then a Component may connect to **atst.tcs.mcs** with:

```
atst.cs.interface.IController mount =
   (atst.cs.interface.IController)App.connect("atst.tcs.mcs");
```

An inappropriate downcast results in a **ClassCastException**.

The interfaces available for downcast are:

- **atst.cs.interfaces.IComponent** - a simple target Component
- **atst.cs.interfaces.IController** - a target Controller

The **atst.cs.services.App** class provides an additional static method that is useful to Component developers:

- **String getName()** -- returns the (registered) name for this application

3.2.2.1 Java commands for IComponent

The following command methods are defined by the **atst.cs.interfaces.IComponent** interface:

- **void set(atst.cs.interfaces.IAttributeTable attributes)**
- **atst.cs.interfaces.IAttributeTable get(atst.cs.interfaces.IAttributeTable attributes)**

The **set** method passes a set of Attributes to the target Component which uses this set to adjust its internal parameters. Typically, a target Component ignores Attributes that have no meaning to the Component.

The **get** method accepts a set of Attributes with empty or irrelevant values - the return value is the same set of Attributes with the values adjusted by the target Component. Attributes that are unknown to the target Component are left unchanged.

3.2.2.2 Java commands for `IController`

The **`atst.cs.interfaces.IController`** interfaces extends **`IComponent`** with the following methods:

- **`void submit(atst.cs.interfaces.IConfiguration config, atst.cs.interfaces.ICmdCallBack callback) throws atst.cs.data.BadConfigException`**
- **`void cancel(String configID)`**
- **`void pause(String configID)`**
- **`void resume(String configID)`**

The **`BadConfigException`** is thrown if the target Controller determines that the Configuration is incomplete, inconsistent, or otherwise invalid.

3.2.2.3 Java command callbacks

When a remote Controller completes the action initiated with a call to the **`IController`** interface method **`submit`**, the connection service receives the completion status event from the remote Controller and invoke the command call back action. The **`atst.cs.interfaces.ICmdCallback`** interface defines the following method:

- **`void onCompletion(String actionStatus, atst.cs.interfaces.IConfiguration config)`**

Controller developers should implement this method, adding any operations they need to have performed on action completion.

3.2.2.4 Java example

The following example demonstrates how a developer might use the connection service. Note that the current implementation of the ATST communications system requires the Internet Communication Engine (ICE) and its companion services. Consult the Reference Manual for instructions on starting the ICE services.

Assume that a subclass of Component exists on a remote machine and has been registered with the connection service. Registration of a Component with the connection service happens automatically when it is loaded into a container. Then using the component's name, one can obtain a reference to the component and invoke operations on it as if it was a local object.

```
// Obtain a reference to the component and downcast appropriately
String componentName = "system.subsystem.device"
atst.cs.interfaces.IComponent component = (IComponent)App.connect(componentName);

// Invoke operations
try {

        // Construct a list of status items to report on -- position, rate, ...
        IAttributeTable status = new AttributeTable();
        status.insert(new Attribute("system.subsystem.device.position", ""));
        status.insert(new Attribute("system.subsystem.device.rate", ""));

        // Obtain the values from the device
        IAttributeTable list = component.get(status);
        list.show("device status is: ");

}
```

```
catch(Exception e)
{
        e.printStacktrace()
}
```

3.2.3 C++ helper

The class **atst::cs::services::App** provides C++ based Components with access to the Connection service. The static methods involved are:

```
IRemote* App::connect(const string& targetComponentName);
void App::disconnect(const string& targetComponentName);
```

Note that **App::connect** returns an IRemote* which can be downcast to the correct interface (i.e. IController). So, for example, if **atst.tcs.mcs** is the name of a controller registered with the connection service then a Component may connect to **atst.tcs.mcs** with:

```
IController* mcsPtr = dynamic_cast<IController*>
App::connect("atst.tcs.mcs");
```

A future implementation of **App::connect** may return a smart pointer and eliminate the explicit downcast making the following possible:

```
tr1::shared_ptr<IController> mcsSptr =
App::connect<IController>("atst.tcs.mcs");
```

In either example, if App::connect fails to connect to the desired target component the returned pointer is set to zero.

The interfaces available for downcast are:

```
<atst::cs::interfaces::IComponent>  // a simple target Component
<atst::cs::interfaces::IController> // a target Controller
```

The **atst::cs::services::App** class provides an additional static method that is useful to Component developers:

```
// Returns the (registered) name for this application
string App::getName();
```

3.2.3.1 C++ commands for IComponent

The following command methods are defined by the interface **atst::cs::interfaces::IComponent**:

```
tr1::shared_ptr<IAttributeTable>
get(tr1::shared_ptr<IAttributeTable> attributes);
void set(tr1::shared_ptr<IAttributeTable> attributes);
```

The **set** method passes a set of Attributes to the target Component which uses this set to adjust its internal parameters. Typically, a target Component ignores Attributes that have no meaning to the Component.

The **get** method accepts a set of Attributes with empty or irrelevant values. The return value is the same set of Attributes with the values adjusted by the target Component. Attributes that are unknown to the target Component are left unchanged.

3.2.3.2 C++ commands for `IController`

TBD

3.2.3.3 C++ command callbacks

TBD

3.2.3.4 C++ example

The following example demonstrates how a developer might use the connection service. Note that the current implementation of the ATST communications system requires the Internet Communication Engine (ICE) and its companion services. Consult the Reference Manual for instructions on starting the ICE services.

Assume that a subclass of Component exists on a remote machine and has been registered with the connection service. Registration of a Component with the connection service happens automatically when it is loaded into a container. Then using the component's name, one can obtain a reference to the component and invoke operations on it as if it was a local object.

```
IController* mcsPtr = dynamic_cast<IController*>
App::connect("atst.tcs.mcs");
tr1::shared_ptr<Attribute> posn =
Attribute::create("atst.tcs.mcs.pos","");
tr1::shared_ptr<Attribute> rate =
Attribute::create("atst.tcs.mcs.rate", "");
tr1::shared_ptr<AttributeTable> status = AttributeTable::create();
status->insert(posn);
status->insert(rate);
tr1::shared_ptr<IAttributeTable> list;
if(mcsPtr) list = get(status);
list->show("atst.tcs.mcs status");
```

3.2.4 Python helper

TBD

-- SteveWampler - 27 Jan 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.26 | > | r1.25 | > | r1.24 | Total page history | Backlinks
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsManual > AtstCsManualConnectionservice

to top

Skip to topic | Skip to bottom

Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
  - News
  - Design
  - Common Services
  - Application base
  - OCS
  - DHS
  - TCS
  - ICS
  - ICDs
  - Docs
  - Meeting Notes
  - Problem Tracking
  - Changes

- **Misc**
  - Users
  - Groups
  - Offices
  - Topic list
  - Search

- **TWiki Webs**
  - Know
  - Main
  - Sandbox
  - TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsManualEventservicer1.16 - 25 Aug 2006 - 21:13 - JanetTvedttopic end

Start of topic | Skip to actions

3.3 Event Service

3.3.1 Events

Publish/subscribe messaging is provided through ATST's *event service*. The event service allows Components to post messages and to perform actions upon the receipt of messages, both without having to connect directly to other Components. The event service provides, through a *helper* class, support for these basic operations.

Events are received by attaching a callback to a subscription. The event service, upon receipt of an event, invokes this callback in a separate thread. However, *all events* received from the same subscription use the same thread so delivery order is preserved within the callback processing. If events are being received faster than the callback processing, the unprocessed events are locally queued within the event service. *This is a potential problem*, but represents a trade-off of mutually exclusive goals. Component developers are encouraged to write callbacks that process quickly. Numerous approaches are available to handle the case where the required action cannot be performed quickly - the best approach to use is dependent upon the nature of the specific task and is thus the responsibility of the Component developer.

3.3.1.1 Java helper

The class **`atst.cs.services.Event`** provides Java-based Components with access to the event service. The static methods provided by this class are:

- **`void subscribe(String eventName, atst.cs.interfaces.IEventCallback callback)`**
- **`void unsubscribe(String eventName, atst.cs.interfaces.IEventCallback callback)`**

- **`void post(String eventName, atst.cs.interfaces.IAttributeTable value)`**
- **`void post(String eventName, long value)`**
- **`void post(String eventName, double value)`**
- **`void post(String eventName, String value)`**
- **`void post(String eventName, atst.cs.interfaces.IAttribute value)`**

The **`post`** methods deserve some explanation. *Internally* the ATST event service uses just the **`IAttributeTable`** value on all posted events. This is the first form of **`post`** shown above. The other **`post`** methods convert the value passed into this internal representation. In the case of **`long`**, **`double`**, and **`String`**, the resulting **`IAttributeTable`** contains a single **`IAttribute`** with the name matching the name of the event. That Attribute's value is the **`String`** representation of the actual value. In the case of value parameter being an **`IAttribute`** (the last of the above methods), that Attribute is wrapped into an **`IAttributeTable`**. See the section on Event callbacks for a discussion on the support available when receiving such events.

3.3.1.2 C++ helper

The class **`atst::cs::services::Event`** provides Java-based Components with access to the event service. The static methods provided by this class are:

```
void Event::subscribe(const string& eventName, const IEventCallback& callback);
```

```
void Event::unsubscribe(const string& eventName, const IEventCallback& callback);
void Event::post(const string& eventName, tr1::shared_ptr<IAttributeTable> value);
void Event::post(const string& eventName, long value);
void Event::post(const string& eventName, double value);
void Event::post(const string& eventName, String value);
void Event::post(const string& eventName, tr1::shared_ptr<IAttribute> value);
```

The **`post`** methods deserve some explanation. *Internally* the ATST event service uses just the **`IAttributeTable`** value on all posted events. This is the first form of **`post`** shown above. The other **`post`** methods convert the value passed into this internal representation. In the case of **`long`**, **`double`**, and **`String`**, the resulting **`IAttributeTable`** contains a single **`IAttribute`** with the name matching the name of the event. That Attribute's value is the **`String`** representation of the actual value. In the case of value parameter being an **`IAttribute`** (the last of the above methods), that Attribute is wrapped into an **`IAttributeTable`**. See the section on Event callbacks for a discussion on the support available when receiving such events.

3.3.1.3 Python helper

TBD

3.3.2 Event callbacks

Components wishing to receive events perform actions on the basis of those events by attaching a *callback* object to the event subscription. Arbitrary actions may be performed by this callback object (but see the caveat in the section on the Event service).

3.3.2.1 Java helper

Component developers need to inject *functional* behavior into event callbacks. ATSTCS provides an adapter class **`atst.cs.services.event.EventCallbackAdapter`** that must be overridden by Component-specific code. Developers should override the single method:

- **`public void callback(String eventName)`**

which is invoked by ATST's event service when a subscribed-to event is received. Note that the actual event value is not passed in as a parameter to this method. Instead, helper methods are provided by **`EventCallbackAdapter`** to obtain the value. These helper methods provide a means of inverting the actions performed by the various **`Event.post()`** methods described in the Event service section.

These helper methods are:

- **`String getString(String attributeName)`** -- produce the named Attribute's value as a `String`
- **`long getLong(String attributeName)`** -- produce the named Attribute's value as a `long`
- **`double getDouble(String attributeName)`** -- produce the named Attribute's value as a `double`
- **`atst.cs.interfaces.IAttributeTable getAttributeTable()`** -- produce the event's full value as an **`AttributeTable`**

Each may be used to fetch the desired value of any attribute in the event's value. If the **`attributeName`** is set to the name of the event itself, then each method acts as an inverse of the corresponding **`post`** method shown above. For example, given a **`long`** value posted with (say) a call:

```
Event.post("atst.ocs.obsCount",5);
```

the **getLong**, **getDouble**, and **getString** methods can all be used inside the **callback** method to obtain the value of the named field as (respectively) a **Long**, **Double**, or **String**. If an illegal conversion is asked for, each method returns **null**.

3.3.2.2 Java example

The following Java code fragments demonstrate how one might use the event service to subscribe/publish events to a named event stream.

MyEventListener overrides the callback method of the **EventCallbackAdapter** class. This class demonstrates how one would deduce the data type of the attribute named after the event. In practice, the subscriber would expect a specific data type and would only call the conversion method appropriate for that data type.

```
import atst.cs.services.event.EventCallbackAdapter;

public class MyEventListener extends EventCallbackAdapter {

        public void callback(String eventName) {

                // Identify who sent the event
                System.out.println("event '"+eventName+"' received");

                // Is it a long value?
                Long lValue = getLong(eventName);
                if (null != lValue) {
                        System.out.println("long value is: "+lValue);
                        return;
                }

                // Is it a double value?
                Double dValue = getDouble(eventName);
                if (null != dValue) {
                        System.out.println("double value is: "+dValue);
                        return;
                }

                // Finally, is it a string value?  (Must be null if not!)
                String sValue = getString(eventName);
                if (null != sValue) {
                        System.out.println("string value is: "+sValue);
                        return;
                }

                System.out.println("Value of '"+eventName+"' not found in event!");

        }
}
```

The following code fragment demonstrates how an application would subscribe to a named event stream using an instance of **MyEventListener** as the remote callback object.

```
MyEventListener listener = new MyEventListener();
Event.subscribe("system.subsystem.device.status", listener);

System.out.println("Press Q to quit");
try {
```

```
        int ichr = 0;
        do {
                ichr = System.in.read();
        } while( (ichr != 'Q') && (ichr != 'q'));
}
catch(java.io.IOException ioexcept) {
        System.err.println("java.io.IOException occurred");
        ioexcept.printStackTrace();
}

System.out.println("unsubscribing");
Event.unsubscribe("system.subsystem.device.status", listener);
```

The following code fragment demonstrates how one would publish events of each supported data type to a named event stream.

```
String eventName = "system.subsystem.device.status";

// Post strings
System.out.println("posting strings ...");
for (int i = 0; i < 100; i++) {
        Event.post(eventName, ""+i);
}

// Post doubles
System.out.println("posting doubles ...");
for (double d = .1; d < 10.0; d+=.125) {
        Event.post(eventName, d);
}

// Post longs
System.out.println("posting longs ...");
for (long l = -100000; l < 100000; l+=5000) {
        Event.post(eventName, l);
}
```

3.3.2.3 C++ helper

Component developers need to inject *functional* behavior into event callbacks. ATSTCS provides an adapter class **`atst::cs::services::event::EventCallbackAdapter`** that must be overridden by Component-specific code. Developers should override the single method:

```
void callback(const string& eventName);
```

which is invoked by ATST's event service when a subscribed-to event is received. Note that the actual event value is not passed in as a parameter to this method. Instead, helper methods are provided by **`EventCallbackAdapter`** to obtain the value. These helper methods provide a means of inverting the actions performed by the various **`Event::post()`** methods described in the Event service section.

These helper methods are:

```
string getString() throw (IllegalConversionException);
long getLong() throw (IllegalConversionException);
double getDouble() throw (IllegalConversionException);
tr1::shared_ptr<IAttribute> getAttribute() throw (IllegalConversionException);
```

Each may be used to fetch the desired value of any attribute in the event's value. If the **attributeName** is set to the name of the event itself, then each method acts as an inverse of the corresponding **post** method shown above. For example, given a **long** value posted with (say) a call:

```
Event::post("atst.ocs.obsCount",5);
```

the **getLong**, **getDouble**, and **getString** methods can all be used inside the **callback** method to obtain the value of the named field as (respectively) a **Long**, **Double**, or **String**. Each method will throw an **IllegalConversionException** if an illegal conversion is asked for.

3.3.2.4 C++ example

TBD

3.3.2.5 Python helper

TBD

-- SteveWampler - 28 Jan 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.16 | > | r1.15 | > | r1.14 | Total page history | Backlinks
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsManual > AtstCsManualEventservice

to top

Skip to topic | Skip to bottom

Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
  - News
  - Design
  - Common Services
  - Application base

- OCS
- DHS
- TCS
- ICS
- ICDs
- Docs
- Meeting Notes
- Problem Tracking
- Changes

- **Misc**
  - Users
  - Groups
  - Offices
  - Topic list
  - Search

- **TWiki Webs**
  - Know
  - Main
  - Sandbox
  - TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsManualLogservicer1.26 - 16 Dec 2006 - 16:11 - SteveWamplertopic end

---

Start of topic | Skip to actions

3.4 Log Service

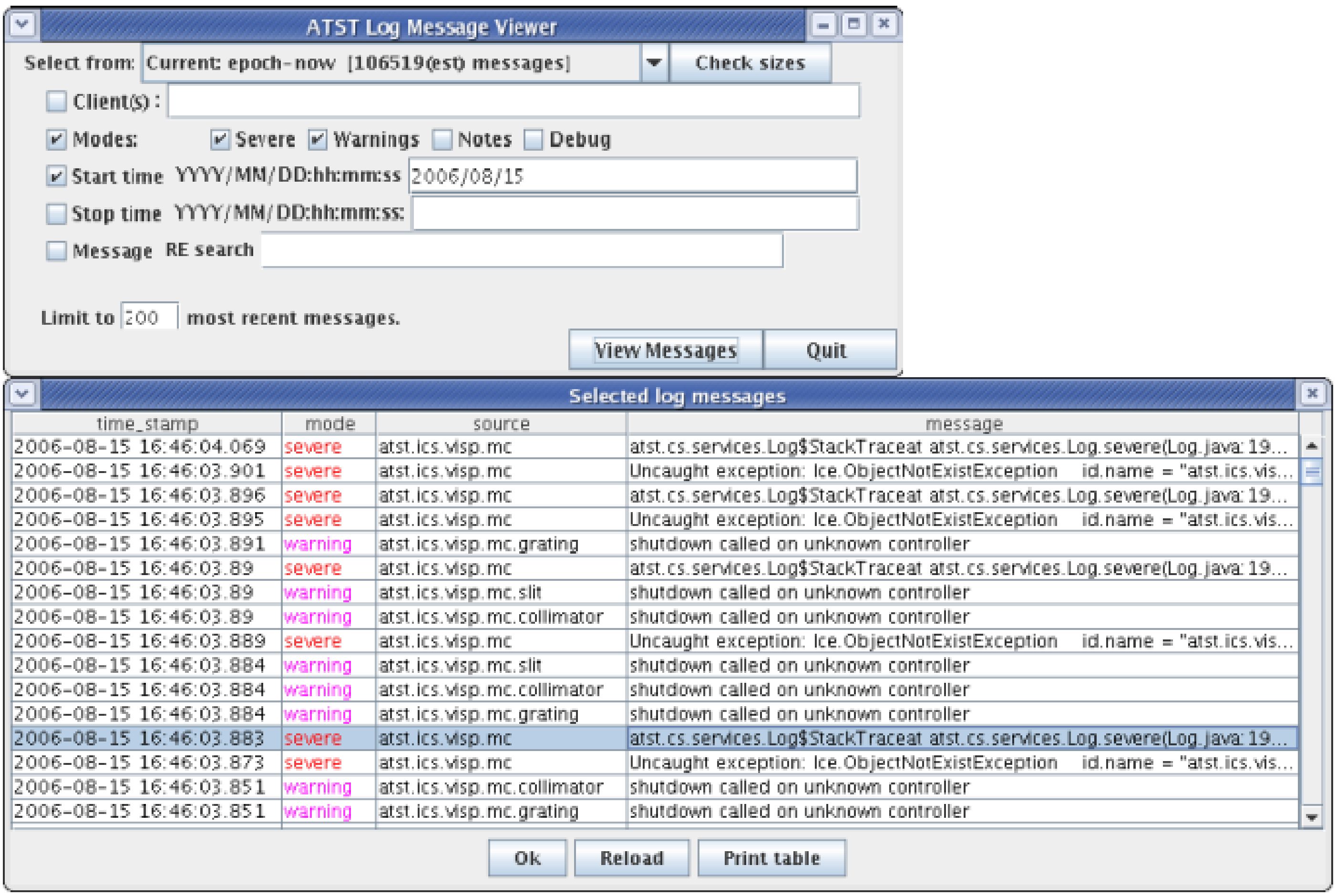
ATST maintains a permanent record, or *log*, of all system activity. Information is recorded into this log as *messages*. There are two types of log messages:

- **status --** messages that one would reasonably expect to *always* be logged.
- **debug --** messages that are only logged during system diagnostics

ATST Common Services provides a robust, high-performance logging system for recording both types of log messages. All log messages are stored in a relational database and automatically timestamped and the ATST Component generating the log message is automatically identified. Log messages may be arbitrarily large but performance is improved if most messages are kept short. The log service provides numerous support operations including the ability of generating a *stack trace* to be logged as an aid in debugging. This stack trace shows the the call tree hierarchy from the point of the stack trace back to the root and so can be used to answer the question *"How did I ever wind up here?"*.

3.4.1 Viewing log messages

The **`LogView`** GUI application may be used to search and display messages that have been recorded into the logging database. Searches may be restrained by a number of criteria.



3.4.2 Message categories

The log service supports grouping messages into broad, cross-Component *categories*. The log service defines the following standard categories for use by Component developers:

- **default** -- general log messages
- **flow** -- trace program flow (call/return, branches, and loops)
- **init** -- trace initialization of hardware/software
- **timer** -- trace timers and delays
- **xfer** -- trace transfers
- **system** -- trace system calls
- **io** -- trace input/output calls
- **reset** -- trace reset actions
- **variable** -- trace actions to key variables

- **event** -- trace events received or generated
- **user1** -- user assigned
- **user2** -- user assigned
- **user3** -- user assigned

Status messages are usually logged using the **default** category.

Developers can add their own categories, but are cautioned to do so *only* after consulting with the ATST software team as there may well be an existing category that is suitable.

Other predefined categories are used to identify messages used within the ATSTCS. These categories, while available for Component developers, are intended more for use within ATSTCS itself:

- **health** -- log changes to system health
- **alarms** -- log alarm conditions
- **connection** -- log connections
- **command** -- log commands and responses
- **lifecycle** -- log Component lifecycle behavior
- **actionstate** -- log changes to the action state
- **container** -- log Container actions
- **commonservices** -- general common services messages
- **cs_connections** -- trace the internal operations of the connection service
- **cs_commands** -- trace the internal operations of the command interface
- **cs_database** -- trace the internal operations involving the ATST persistent stores
- **cs_archive** -- trace internal operations of the archiver service
- **cs_property** -- trace internal operations of the property service
- **cs_monitor** -- trace internal operations of the monitor service
- **cs_gui** -- trace internal operations of the GUI support

Categories are also useful in controlling actions outside the log system. The section on the Archive service includes an example where a custom log category is used to control archiving of data.

3.4.3 Status messages

A status message may belong to one of three classes:

- **note** -- messages that are informative but do not indicate a particular problem should belong to this class.
- **warning** -- messages indicating a problem that is not yet severe enough to interfere with proper operation of the Component belong in this class. If the current debug level is set to non-zero, the message *automatically* append the current method name, source file name, and line number within that file of the call.
- **severe** -- messages reporting conditions under which the component is unable to perform normal operation belong in this class. Severe messages *automatically* include a strack trace.

Status messages are *always* logged.

Note that unconstrained use of status messages does little to enhance the usefulness of the system log and may impose unnecessary strain on resources. Most of the places where status messages are appropriate are within ATSTCS - there is very little need for Component developers to add status messages. Places where ATSTCS automatically produces status messages include:

- lifecycle changes

- health changes
- connection changes
- commands and responses
- alarms

Nevertheless, status messaging is available to Component developers should specific Component functionality suggest their use.

3.4.4 Debug messages

Debug messages provide information useful when attempting to track down problems and are normally disabled. An operator may enable/disable debug messages at any time on a per category basis. Debug messages are identified by log category, as described above, and by *level*.

3.4.4.1 Debug levels

The log service requires debug messages to be identified by a *level*. In general, the higher the level, the more information is provided for debugging. A *current debug level* setting determines which debug messages are generated. All messages whose level is less than or equal to the current debug level are produced, so setting the debug level to **`2`**, for example, would cause messages at levels **`1`** and **`2`** to be produced. The current debug level is not by set by Component code, but is managed by the Component's *lifecycle* interface. Every log category maintains its own debug level.

Developers are cautioned that unconstrained debug use can put a strain on system resources and are thus encouraged to set the debug levels of messages judiciously. Putting level **`1`** messages inside tight loops, for example, is probably *not* a good idea. Otherwise, determining the appropriate level for a particular debug message is probably more art than science. ATST offers the following recommendations:

- level **1**: at the entry/exit of major code sections (code modules)
- level **2**: at the entry/exit of methods and procedures (unless these are expected to be called within tight loops) and in object constructors (with the same caveat)
- level **3**: at key points within methods and procedures (again, outside of tight loops)
- level **4**: within tight loops (should be small messages with critical information only)

For example, in the **flow** category, level **3** messages should be used to identify branches taken in the code and the start of major loops. It is *always* a bad idea to include stack traces in level **4** debug messages!

3.4.5 Convenience methods

Additional convenience methods are available that provide features. These are described in the language-specific *helper* sections, below.

3.4.6 Java helper

The class **`atst.cs.services.Log`** provides a rich set of static methods for interacting with the logging system:

- **`boolean isEnabled(String category)`** - is debugging enabled in the category?
- **`boolean isDebuggable(String category, int level)`** - will a debug message be logged at the given category and level?

- **`int getDebugLevel(String category)`** - produce the category's current debug level

- **`void note(String category, String message)`** - log a note in the category
- **`void warn(String category, String message)`** - log a warning in the category
- **`void severe(String category, String message)`** - log a severe condition in the category
- **`void debug(String category, int level, String message)`** - log a debug message if level <= current debug level

Since the expectation is that status messages (**note**, **warn**, and **severe**) produced by Component developers will likely belong in the **default** category, the following convenience methods are defined:

- **`void note(String message)`** - log a note in the default category
- **`void warn(String message)`** - log a warning in the default category
- **`void severe(String message)`** - log a severe message in the default category

The next few convenience methods are independent of logging category:

- **`String curLoc()`** - returns a string containing the method name, source file name, and line number within that file of the call.
- **`String getStackTrace()`** - returns a string containing a full stack trace from the current location in the source code.
- **`String getStackTrace(Throwable ex)`** - returns, as a string, the full stack trace for the exception **`ex`**.

The **`getStackTrace`** methods can be used to add a stack trace to *any* log message. Remember, though, that **`severe`** messages have a stack trace added automatically.

The following are the standard category definitions available to Java Component developers through the log service:

- **`String Log.DEFAULT`** -- default logging category
- **`String Log.FLOW`** -- log flow-control block entry/exit
- **`String Log.INIT`** -- trace initialization of hardware/software
- **`String Log.TIMER`** -- trace timers and delays
- **`String Log.XFER`** -- trace transfers
- **`String Log.SYSTEM`** -- trace system calls
- **`String Log.IO`** -- trace input/output calls
- **`String Log.RESET`** -- trace hardware/software reset actions
- **`String Log.VARIABLE`** -- trace key variable assignments
- **`String Log.EVENT`** -- trace received/generated events
- **`String Log.USER1`** -- user assigned
- **`String Log.USER2`** -- user assigned
- **`String Log.USER3`** -- user assigned

while the categories used within the Common Services are:

- **`String Log.HEALTH`** -- log changes to system health
- **`String Log.ALARM`** -- log alarm conditions
- **`String Log.CONNECT`** -- log connections
- **`String Log.COMMAND`** -- log commands and responses
- **`String Log.LIFECYCLE`** -- log Component lifecycle behavior
- **`String Log.ACTSTATE`** -- log changes to the action state

- **String Log.CONTAINER** -- log Container actions
- **String Log.CS** -- general common services messages
- **String Log.CS_CONNECT** -- trace the internal operations of the connection service
- **String Log.CS_COMMAND** -- trace the internal operations of the command interface
- **String Log.CS_DB** -- trace the internal operations involving the ATST persistent stores
- **String Log.CS_ARCHIVE** -- trace internal actions of the archiver service
- **String Log.CS_PROPERTY** -- trace internal actions of the property service
- **String Log.CS_MONITOR** -- trace internal actions of the monitor service
- **String Log.CS_GUI** -- trace internal GUI support operations

If no category is given, **Log.DEFAULT** is assumed.

3.4.7 Java example

The following code sample demonstrates the usage of a few capabilities of the logging service.

The entry and exit points of the function are marked with level 2 debug statements in the default category. These debug statements will be logged only if the current debug level for the default category is 2 or greater.

In the arguments parsing, the warning status message is logged if the "startup.mode" argument is missing regardless of the current debug level. This warning message also appends an optional stacktrace by calling Log.getStackTrace().

The Log.INIT category is used for messages relating to hardware initialization. So the debug statement "initializing hardware controller" will be logged in the Log.INIT category if the debug level for that category is set to 3 or above. The severe status message will be logged if there is an error during hardware initialization regardless of the debug level. A stacktrace is automatically appended by the logging service for severe status messages.

```
public void doInit(IAttributeTable args) {

        // Entry point
        Log.debug(2, "initializing. In "+Log.curLoc());

        // Parse arguments
        IAttribute mode = args.get("startup.mode");
        if(mode == null) Log.warn("startup mode unspecified -- stacktrace: "+Log.getStackTrace());

        // Hardware initialization
        Log.debug(Log.INIT, 3, "initializing hardware controller");
        if((status = controllerInit(defaultMode)) != OK)
        Log.severe(Log.INIT, "Error initializing hardware controller, status = "+status);

        // Exit point
        Log.debug(2, "initialization complete");
}
```

3.4.8 C++ helper

The class **atst::cs::services::Log** provides a rich set of static methods for interacting with the logging system:

```
// Is debugging enabled in the category?
bool Log::isEnabled(const string& category);

// Will a debug message be logged at the given category and level?
```

```
bool Log::isDebuggable(const string& category, int level);

// Produce the category's current debug level
int Log::getDebugLevel(const string& category);

// Log a note in the category
void Log::note(const string& category, const string& message);

// Log a warning in the category
void Log::warn(const string& category, const string& message);

// Log a severe condition in the category
void Log::severe(const string& category, const string& message);

// Log a debug message if level <= current debug level
void Log::debug(const string& category, int level, const string& message);
```

Since the expectation is that status messages (**note**, **warn**, and **severe**) produced by Component developers will likely belong in the **default** category, the following convenience methods are defined:

```
// Log a note in the default category
void Log::note(const string& message);

// Log a warning in the default category
void Log::warn(const string& message);

// Log a severe message in the default category
void Log::severe(const string& message);
```

The next few methods are independent of logging category:

```
// Returns a string containing a stack trace from the current location in the source code
string Log::getStackTrace();

// Returns a string containing the stack trace for the exception
string Log::getStackTrace(const std::exception& ex);
```

The **`getStackTrace`** methods can be used to add a stack trace to *any* log message. Remember, though, that **`severe`** messages have a stack trace added automatically.

The following are the standard category definitions available to C++ Component developers through the log service:

```
const string Log::DEFAULT;  // The default is no category
const string Log::FLOW;       // Program flow
const string Log::INIT;       // hardware/software init
const string Log::TIMER;    // timers and delays
const string Log::XFER;       // xfers
const string Log::SYSTEM;   // system calls
const string Log::IO;       // input/output calls
const string Log::RESET;    // hardware/software reset
const string Log::VARIABLE; // Variables
const string Log::EVENT;    // Events
const string Log::USER1;    // User defined
const string Log::USER2;    // User defined
const string Log::USER3;    // User defined
```

while the categories used within the Common Services are:

```
const string Log::ALARM;   // Alarms
const string Log::HEALTH;   // Health
const string Log::CONNECT;   // Connections
const string Log::COMMAND;   // Commands/Responses
const string Log::LIFECYCLE;   // Lifecycle
const string Log::ACTSTATE;     // Action states
const string Log::CONTAINER;   // ICE internals
const string Log::CS;      // General CommonService
const string Log::CS_DB;   // Database internals
const string Log::CS_CONNECT;   // connection internals
const string Log::CS_COMMAND;   // command internals
const string Log::CS_ARCHIVE;   // Archive internals
const string Log::CS_PROPERTY;   // Property internals
const string Log::CS_MONITOR;   // Monitor internals
const string Log::CS_GUI;   // GUI support internals
```

If no category is given, **`Log::DEFAULT`** is assumed.

3.4.9 C++ example

TBD

3.4.10 Python helper

TBD

-- SteveWampler - 01 Feb 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.26 | > | r1.25 | > | r1.24 | Total page history | Backlinks
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsManual > AtstCsManualLogservice

to top

Skip to topic | Skip to bottom

Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
  - News
  - Design
  - Common Services
  - Application base
  - OCS
  - DHS
  - TCS
  - ICS
  - ICDs
  - Docs
  - Meeting Notes
  - Problem Tracking
  - Changes

- **Misc**
  - Users
  - Groups
  - Offices
  - Topic list
  - Search

- **TWiki Webs**
  - Know
  - Main
  - Sandbox
  - TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsManualHealthservicer1.11 - 22 Aug 2006 - 17:51 - JanetTvedttopic end

---

Start of topic | Skip to actions

3.5 Health Service

The *health service* is used by ATST to maintain the "health" condition of a Component. The health may be one of (presented in worsening order):

- **`GOOD`** - no problems have been detected by the Component

- **`ILL`** - problems have been detected, but they do not prevent observing. Data *quality*, however, may be affected. It may also be the case that operation of the Component will fail soon if corrective action is not taken.
- **`BAD`** - severe problems have been detected. The Component is unable to operate correctly. Corrective action is required.
- **`UNKNOWN`** - the Component is not responding. It may or may not be operating. This health value is not set by the Component (obviously) but may be set by the health service.

The health of a Component is not dependent upon the health of any components that it may be connected to. The health *is*, however, dependent upon the health of the connections themselves. A *failed* connection forces the health to be at least **`ILL`**. Component developers do not need to worry about these connections however, checking the health of connections is handled by the health service itself.

Strictly speaking, a Component does not set its own health status. Instead, Component developers must implement a method, **`doCheckHealth`**, that detemines the Component's health. This method is used by the health service in determining the health status. The health service also provides convenience functions useful in the implementation of **`doCheckHealth`**.

The health service operates as a separate task (or thread) that runs in parallel with the Component. At a periodic interval the health service *polls* the Component using the above method. The polling interval is controlled by the health service but may be different for different Components and can change dynamically. A *timeout* mechanism is used during polling to avoid hanging the health service. Repeated timeouts automatically worsen the Component's health condition.

The health service automatically posts an event showing *changes* to the Component health and logs a `warning` on worsening health and a `note` on improving health. When a health condition worsens to **`BAD`** or **`UNKNOWN`** the log message switches from `warning` to `severe`.

3.5.1 Java helper

The **`atst.cs.services.Health`** class provides Component access to the health service. The following constants represent the above health settings:

- **`Health.GOOD`** -- good health
- **`Health.ILL`** -- ill health
- **`Health.BAD`** -- bad health
- **`Health.UNKNOWN`** -- unknown health. This value is not to be used by Component developers.

Component developers must implement a method that, when called, produces one of the above values:

- **`void doCheckHealth()`** - check the health of the Component

When implementing `doCheckHealth()`, the following static methods can be useful:

- **`String Health.getHealth()`** - produce the current health of the named Component
- **`void Health.setHealth(String health, String reason)`** - record what the current health should be, and why. This method ***must*** be called within `doCheckHealth()` to record the health.

The `getHealth` method is only useful in places where you care in your tests what the current health is. The last, **`setHealth()`**, is required to be called at some point within `doCheckHealth()`.

3.5.2 Java example

The following is an example of a health check function for a subsystem whose health depends on the availability of an internal pool of resources. The health of the subsystem is deemed:

- **BAD** — if there are no more resources available in the pool.
- **ILL** — if there is only one resource available in the pool.
- **GOOD** — if there are two or more available resources.

```
public void doCheckHealth() {

        switch (resourcePool.numAvailable()) {
                case 0: Health.setHealth(Health.BAD, "no resources left in pool");
                break;
                case 1: Health.setHealth(Health.ILL, "low resources left in pool");
                break;
                default:Health.setHealth(Health.GOOD, null);
        }

}
```

3.5.3 C++ helper

The **`atst::cs::services::Health`** class provides Component access to the health service. The following static constants represent the above health settings:

```
const string Health::GOOD; //good health
const Health::ILL;  //ill health
const Health::BAD;  //bad health
const Health::UNKNOWN; //unknown health -- not to be used by Component developers
```

Component developers must implement a method that, when called, produces one of the above values:

```
void doCheckHealth();  // check the health of the Component
```

When implementing `doCheckHealth()`, the following static methods can be useful:

```
// Produce the current health of the named Component
string Health::getHealth();

// Record what the current health should be and why
void Health::setHealth(const string& health, const string& reason);
```

The ***Health::setHealth*** method must be called within `doCheckHealth()` to modify the stored value of the Component's health. The `getHealth` method is only useful in places where you care in your tests what the current health is.

In Sacramento-1, Components do not implement the doCheckHealth method. This will be corrected in Sacramento-2.

3.5.4 C++ example

TBD

3.5.5 Python helper

TBD

-- SteveWampler - 02 Feb 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.11 | > | r1.10 | > | r1.9 | Total page history | Backlinks
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsManual > AtstCsManualHealthservice

to top

Skip to topic | Skip to bottom

Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
    - News
    - Design
    - Common Services
    - Application base
    - OCS
    - DHS
    - TCS
    - ICS
    - ICDs
    - Docs
    - Meeting Notes
    - Problem Tracking
    - Changes

- **Misc**
    - Users
    - Groups

- Offices
- Topic list
- Search

- **TWiki Webs**
  - Know
  - Main
  - Sandbox
  - TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsManualToolsr1.3 - 11 Mar 2005 - 20:56 - SteveWamplertopic end

---

Start of topic | Skip to actions

---

4 Tools (Minor Services)

4.1 Introduction to the Tools

The ATST common service *tools* (aka "*minor services*") provide generally-useful support for Component developers. In many cases these tools expose functionality used internally within the common services.

-- SteveWampler - 02 Feb 2005
to top

End of topic
Skip to action links | Back to top

- **Misc**
    - Users
    - Groups
    - Offices
    - Topic list
    - Search

- **TWiki Webs**
    - Know
    - Main
    - Sandbox
    - TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsManualArchiveservicer1.8 - 14 Mar 2006 - 14:57 - SteveWamplertopic end

---

Start of topic | Skip to actions

4.3 Archive service

The *archive service* provides high-performance archiving of **`Attributes`** (name/value pairs). A timestamp and the name of the source Component are automatically recorded with each **`Attribute`**. The intent is to provide a means of saving bursts of engineering data for later analysis. Normally, developers should wrap calls to the archive service within tests to control the production of archived data. For example, one could (using pseudocode that only happens to look like Java):

```
Log.note("TCS_ARCHIVING","Archiving target position stream");
....
if (isEnabled("TCS_ARCHIVING")) {
        Archive.archive(mcsPosition);
}
....
Log.note("TCS_ARCHIVING","Done archiving target position stream");
```

Note that this example uses a log service *category* to control access to the archiver. The log service *debug level* can also be used to refine this control.

At the current time, there is no programmatic support in the archive service for retrieving values that have been archived. The archived data is maintained in a relational database and **SQL** commands may be used to retrieve values from the archive. Applications that support the analysis of archived data are expected to be added to the ATST common services at some point in the future.

4.3.1 Java helper

The **`atst.cs.services.Archive`** class contains some static methods to access the archive service:

- **`void archive(atst.cs.data.Attribute attribute)`** - archives **`attribute`**
- **`void archive(String name, String value[])`** - archive a name/value pair
- **`void archive(String name, String value)`** - archive a name/value pair
- **`void archive(String name, double value)`** - archive a name/value pair
- **`void archive(String name, long value)`** - archive a name/value pair

In the second case, note that **`value`** is an array of strings. This name/value pair is used to construct an **`Attribute`** for archiving, so there is no performance advantage in using the second form - it is merely a convenience method. This holds true for the last two methods also.

4.3.2 C++ helper

TBD

4.3.3 Python helper

TBD

-- SteveWampler - 02 Feb 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.8 | > | r1.7 | > | r1.6 | Total page history | Backlinks
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsManual > AtstCsManualArchiveservice

to top

Skip to topic | Skip to bottom

Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**

- Main Web Home
    - News
    - Design
    - Common Services
    - Application base
    - OCS
    - DHS
    - TCS
    - ICS
    - ICDs
    - Docs
    - Meeting Notes
    - Problem Tracking
    - Changes

- **Misc**
    - Users
    - Groups
    - Offices
    - Topic list
    - Search

- **TWiki Webs**
    - Know
    - Main
    - Sandbox
    - TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsManualPropertyservicer1.12 - 01 Nov 2006 - 22:59 - SteveWamplertopic end

---

Start of topic | Skip to actions

4.4 Property Service

The *Property Service* maintains *metadata* about Attributes in a *persistent store*. This metadata consists of:

- **`type`** - the type of the Attribute's value(s). This is an *ATST type*, not an implementation language type. For example, a simple position is a **`Position`** while a 2-D position is an **`XYPosition`**. At the current time (more types will be added) the following types exist:

- ♦ **"string"** - an arbitrary-length string
- ♦ **"integer"** - a signed, integral value of upto 64 bits.
- ♦ **"real"** - a signed, floating-point value of 64 bits.
- ♦ **"boolean"** - a simple boolean value

- **vector** - a boolean flag that is **true** if the attribute is a *vector* of **type**. Note that some types of Attributes are themselves vector-valued. This flag would denote a *vector of vectors* in such cases.
- **readonly** - a boolean flat that is **true** if the attribute is a *readonly* Attribute. Readonly attributes cannot be changed by an external command (i.e. **submit** or **set**).
- **description** - a description of the Attribute.
- **values** - the Attribute's value(s). Applications may record the current value of an Attribute to provide *persistency*.
- **defaults** - the default value(s) for the Attribute, if any.
- **limits** - bounds of legal values, if any. The values here depend upon the **type**:
  - ♦ **string** limits, if they exist, denote **exact** values that are legal strings. In essense, this means that the property denotes an *enumeration*. If there are no limits, then any string is considered legal.
  - ♦ **integer** limits, if they exist, denote a (low,high)-inclusive range. If only a single limit is given, it is the lowest value allowed.
  - ♦ **real** limits, if they exist, denote a (low,high)-inclusive range. If only a single limit is given, it is the lowest value allowed.
  - ♦ **boolean** limits don't exist since it is a self-limiting type.
- **changeDeltas** - changes below these deltas, if any, are not monitored. Not all types have changeDeltas, but those that do typically have two values: **lowDelta** and **highDelta**. Change deltas are only meaningful on numeric-valued data.

4.4.1 Properties versus Constants

The Property Service provides a mechanism that allows Components access to metadata on *application-specific* Attributes. This can be distinquished from other types of information such as *manifest constants* that are immutable across all ATST applications. Examples of such constants include the information that uniquely describes the precise location (*latitude*, *longitude*, and *elevation*) of ATST. Such manifest constants are provided by the `Constant Service`. While the implementation internals of the Constant service are built on top of the property service implementation, the two services are distinct at the Component level.

4.4.2 Component access to Attribute metadata

The **Property** access helper provides Component developers with the following methods:

- **exists(attributeName)** -- true if a property set for this attribute exists
- **getType(attributeName)** -- produce the type (as a string) of the named attribute
- **isVector(attributename)** -- returns **true** if this attribute is a vector of the indicated **type**
- **isReadonly(attributename)** -- returns **true** if this attribute is readonly.
- **getDescription(attributeName)** -- produce the description of the attribute
- **saveAttribute(attribute)** -- record the value of attribute into the persistent store.
- **getAttribute(attributeName)** -- produce an **Attribute** for the named attribute, with the saved value restored. If no values have been saved, then the *default* values are used. Note that, as an Attribute, the value is represented as an array of strings.
- **getDefault(attributeName)** -- produce an **Attribute** for the named attribute, with all values set to their defaults.
- **getValues(attributeName)** -- produce any saved values.
- **getDefaults(attributeName)** -- produce the default values.

- **getLimits(attributeName)** -- produce the limits values.
- **getDeltas(attributeName)** -- produce the change deltas.

A **null** is returned if the requested metadata item does not exist for the named attribute.

There are a series of *convenience* methods for performing various tests against the properties:

- **inRange(attributeName, simpleValue)** -- is **simpleValue** within limits?
- **inRangeString(attributeName, simpleValue)** -- is **simpleValue** (a string) within limits?
- **inRangeStrings(attributeName, simpleValues)** -- are all **simpleValues** (strings) within limits?

Several additional convenience methods are also defined to handle other common cases:

- **setValue(attributeName, value)** -- set **value** as the first saved value of the attribute's property.
- **getValue(attributeName)** -- produce the first saved value of the attribute's saved value.

Both of these leave any other values untouched (remember that any attribute can hold an array of values).

Note that there is *no support* for a Component altering any of the metadata except for the saved values.

A special method is provided for the few cases where a Component needs access to the properties of a *different* component:

- **getRemoteProperties(String appName)** -- returns a table holding all the properties for the named Component.

(This situation is expected to be limited to components that *manage* other components.)

4.4.3 Java property service helper

The implementation of the access helper is straightforward and provides Java implementations of the above methods:

- **boolean exists(String attributeName)** -- true if a property set for this attribute exists
- **String getType(String attributeName)** -- produce the attribute's **type**.
- **boolean isVector(String attributeName)** -- true if attribute is a vector of the indicated **type**.
- **boolean isReadonly(String attributeName)** -- true if attribute is readonly.
- **String getDescription(String attributeName)** -- produce the attribute's description.
- **void saveAttribute(atst.cs.interfaces.IAttribute attribute)** -- save the attribute's value into the persistent store.
- **atst.cs.interfaces.IAttribute getAttribute(String attributeName)** -- produce an `Attribute` for the named attribute using the saved values.
- **atst.cs.interfaces.IAttribute getDefault(String attributeName)** -- produce an `Attribute` for the named attribute with all values set to defaults.
- **Object[] getValues(String attributeName)** -- produce the array of saved values.
- **Object[] getDefaults(String attributeName)** -- produce the array of default values.
- **Object[] getLimits(String attributeName)** -- produce the array of limit values.
- **Object[] getDeltas(String attributeName)** -- produce the array of change deltas.
- **atst.cs.interfaces.IPropertyTable getRemoteProperties(String appName)** -- produce the properties for the named Component.

Note that casts are required to properly use **getValues**, **getLimits**, **getDeltas**, and **getDefaults**, and that the user is responsible for understanding the meaning of each array element for each.

Convenience methods are provided for performing tests against the property:

- **boolean inRange(String attributeName, String value)** -- is **value** within the limits of the named attribute?
- **boolean inRange(String attributeName, Integer value)** -- is **value** within the limits of the named attribute?
- **boolean inRange(String attributeName, Double value)** -- is **value** within the limits of the named attribute?
- **boolean inRangeString(String attributeName, String sValue)** -- is **sValue** (the String form for a value of the property's type) within limits? limits of the named attribute?
- **boolean inRangeStrings(String attributeName, String[] sValues)** -- are **sValues** within the limits of the named attribute? Returns **true** only if all are within limits.

Some additional convenience methods are included to handle the common case where the value of the attribute is a single string value:

- **void setValue(String attributeName, String value)** -- set the saved value of the named property for the named attribute to the indicated value.
- **void setValue(String attributeName, Long value)** -- set the saved value of the named property for the named attribute to the indicated value.
- **void setValue(String attributeName, Double value)** -- set the saved value of the named property for the named attribute to the indicated value.
- **void setValue(String attributeName, Boolean value)** -- set the saved value of the named property for the named attribute to the indicated value.
- **Object getValue(String propertyName)** -- return the saved value of the named property. Note that the result must be cast to the appropriate type for the named property.

In all of these convenience methods, the value is the *first* value held in the property, other values, if any are undisturbed.

4.4.3.1 Java example

The following code sample shows how one might obtain the limits for a real-valued attribute **anAttribute**. These limits could be used, for example, to verify values for this attribute in a user interface.

```
Double[] limits = (Double)Property.getLimits("anAttribute");
```

If the variable **dValue** contains a **Double**, the following code tests **dValue** against those same limits:

```
if (Property.inRange("anAttribute", dValue)) {
        // code to perform if value is within limits
}
```

similarly, an attribute containing the default values for **anAttribute** can be obtained with:

```
IAttribute anAttribute = Property.getDefault("anAttribute");
```

4.4.4 C++ helper

TBD

4.4.5 Python helper

TBD

-- SteveWampler - 02 Feb 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.12 | > | r1.11 | > | r1.10 | Total page history | Backlinks
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsManual > AtstCsManualPropertyservice

to top

Skip to topic | Skip to bottom

Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
    - News
    - Design
    - Common Services
    - Application base
    - OCS
    - DHS
    - TCS
    - ICS
    - ICDs
    - Docs
    - Meeting Notes
    - Problem Tracking
    - Changes

- **Misc**
    - Users
    - Groups
    - Offices
    - Topic list
    - Search

- **TWiki Webs**
    - Know
    - Main
    - Sandbox
    - TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsManualConstantservicer1.4 - 12 Apr 2006 - 18:12 - JanetTvedttopic end

Start of topic | Skip to actions

4.5 Constant Service

The *Constant Service* maintains ATST *manifest constants*: values that are immutable and uniform across all ATST applications. Examples of such manifest constants include the precise location (*latitude*, *longitude*, and *elevation*) of ATST.

At the current time, the following manifest constants exist:

- **`latitude`** -- position of ATST
- **`longitude`** -- position of ATST
- **`elevation`** -- position of ATST

4.5.1 Component access to manifest constants.

The **`Constant`** access helper provides Component developers with the following methods:

- **`getValue(constantName)`** -- produce the value (as a string) of the named constant.
- **`getDescription(constantName)`** -- produce the description of the named constant.

In addition, the following *convenience methods* are defined.

- **`getDouble(constantName)`** -- produce the constant value as a floating point number.
- **`getLong(constantName)`** -- produce the constant value as an integer.

A **`null`** is returned if the requested metadata item does not exist for the named attribute or if an impossible conversion is required.

4.5.2 Java property service helper

The implementation of the access helper is straightforward and provides Java implementations of the above methods:

- **`String getValue(String constantName)`** -- produce the constant's value as a String.
- **`String getDescription(String constantName)`** -- produce the constant's description.
- **`Double getDouble(String constantName)`** -- produce the constant's value as a Double, returning **`null`** if that representation isn't possible.
- **`Long getLong(String constantName)`** -- produce the constant's value as a Long, returning **`null`** if that representation isn't possible.

4.5.2.1 Java example

The following code sample shows how one might obtain the latitude for ATST:

```
String latitude = Constant.getValue("latitude");
```

4.5.3 C++ helper

TBD

4.5.4 Python helper

TBD

-- SteveWampler - 17 Jan 2006
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.4 | ≥ | r1.3 | ≥ | r1.2 | Total page history | Backlinks
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsManual > AtstCsManualConstantservice

to top

Ideas, requests, problems regarding ATST_Software? Send feedback.
Skip to topic | Skip to bottom

Jump:
Main

Main.AtstCsManualMonitorservicer1.4 - 14 Mar 2006 - 14:58 - SteveWamplertopic end

---

4.6 Monitor Service

The *monitor service* provides support for monitoring key values within a Component. It operates in conjunction with the Property Service. Component developers may attach a *monitor* to an Attribute. The monitor watches for changes to the value of that Attribute and performs arbitrary actions on the basis of those changes. The most common, and the default, action is to post an event announcing the change. Monitors consult the property service to obtain information about value hysteresis, limits, etc. to help refine the definition of a value "change".

The monitor service is a new addition to the ATST common services. At this time, most of the details are still being refined. You can expect the sparse information currently in this section to be filled out and possibly altered in future versions of this document.

4.6.1 Java helper

TBD

4.6.2 C++ helper

TBD

4.6.3 Python helper

TBD

-- SteveWampler - 02 Feb 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.4 | ≥ | r1.3 | ≥ | r1.2 | Total page history | Backlinks
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsManual > AtstCsManualMonitorservice

to top

Skip to topic | Skip to bottom

Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
    - News
    - Design
    - Common Services
    - Application base
    - OCS
    - DHS
    - TCS
    - ICS
    - ICDs
    - Docs
    - Meeting Notes
    - Problem Tracking
    - Changes

- **Misc**
    - Users
    - Groups
    - Offices
    - Topic list
    - Search

- **TWiki Webs**
    - Know
    - Main
    - Sandbox
    - TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsManualUIservicer1.4 - 14 Mar 2006 - 14:58 - SteveWamplertopic end

---

Start of topic | Skip to actions

4.7 User Interfaces Support

The ATST common services provides basic support for implementing user interfaces.

At the current time, this support is ***TBD***. For now, the only documentation can be found in the source code documentation of the **`ATST.cs.util.gui`** package, which contains a (very) limited set of Java classes.

-- SteveWampler - 02 Feb 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.4 | ≥ | r1.3 | ≥ | r1.2 | Total page history | Backlinks
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsManual > AtstCsManualUIservice

to top

Skip to topic | Skip to bottom

Jump:
Main

- **ATST Software**
- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
    - News
    - Design
    - Common Services
    - Application base
    - OCS
    - DHS
    - TCS
    - ICS
    - ICDs
    - Docs
    - Meeting Notes
    - Problem Tracking
    - Changes

- **Misc**
    - Users
    - Groups
    - Offices
    - Topic list
    - Search

- **TWiki Webs**
  - Know
  - Main
  - Sandbox
  - TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsManualMiscservicesr1.8 - 14 Mar 2006 - 14:58 - SteveWamplertopic end

Start of topic | Skip to actions

4.8 Miscellaneous Services

This sections covers a variety of minor services provided by ATST common services. In most cases they are simply introduced here. Most or either too simple or too obscure to warrant much discussion. Others are language specific. Details of their use can be found in the *source code documentation*.

4.8.1 Thread support

The ATST common services provides a few tools to support threaded programming. Specifically, there is a simple implementation of a *thread pool* used inside the common services that is exposed for use by Component developers. As an example, thread support is utilized by the Controller class to provide the action threads. Details on the use of the ATST thread pools can be found in the Java source code documentation for the **`atst.cs.util.threads`** package. Details for C++ and Python are ***TBD*** .

4.8.2 Generic pools

Besides thread pools, other types of *pools* (collections of isomorphic resources) are possible. The ATST common services provides a generic mechanism for managing such pools. Pools may be *fixed* in size or automatically *growable* as pool resources are depleted and can by dynamically switched back and forth between fixed and growable. The pool keeps track of both *active* and *unallocated* resources.

Details can be found in the Java source code documentation for the **`atst.cs.util.Pool`** class. Details for C++ and Python are ***TBD*** .

4.8.3 ID Service

The *ID service* is used internal to the ATST common services but is exposed in case Component developers need a similar functionality. This service provides identification strings that are *guaranteed* to be unique across *all* of ATST. The service is high-performance with 90,000+ IDs produced per second. The ID strings contain a number that is monotonically increasing but not necessarily sequential.

4.8.3.1 Java helper

The class **`atst.cs.services.IdDB`** provides access to the ID service. A single static method is available:

- **`String getId(String prefix)`** - returns a unique id beginning with **`prefix`**

The choice of a **`prefix`** is a convenience choice - the ID will be unique regardless of the prefix.

4.8.3.2 C++ helper

TBD

4.8.3.3 Python helper

TBD

4.8.4 Java date service

Dates and times are heavily used in ATST, as is the case with most observatories. The **`GregorianCalendar`** class provided by Java has the unfortunate property of producing *massive* serialized forms. The ATST **`atst.cs.util.AtstDate`** class provides an alternative to **`GregorianCalendar`** that produces a much smaller serialized form. Details can be found in the Java source code documentation for the **`atst.cs.util.AtstDate`** class. This service is not available for C++ and python Components.

-- SteveWampler - 02 Feb 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.8 | ≥ | r1.7 | ≥ | r1.6 | Total page history | Backlinks
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsManual > AtstCsManualMiscservices

to top

Skip to topic | Skip to bottom

Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
    - News
    - Design
    - Common Services
    - Application base
    - OCS
    - DHS
    - TCS
    - ICS
    - ICDs
    - Docs
    - Meeting Notes
    - Problem Tracking
    - Changes

- **Misc**
    - Users
    - Groups
    - Offices
    - Topic list
    - Search

- **TWiki Webs**
    - Know
    - Main
    - Sandbox
    - TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsManualComponentsr1.30 - 06 Feb 2007 - 16:13 - SteveWamplertopic end

---

Start of topic | Skip to actions

---

5 Components and Controllers

5.1 Components

The `Component` is the foundation for all applications in ATST. Most ATST applications extend the `Controller` , a subclass of `Component` that adds *configuration-management* features. Components are managed by **Containers**, which are responsible for managing the *lifecycle* characteristics of Components. Consequently, there are no **main** functions for Components - Components do not exist as standalone entities. Containers also provide Components with access to the services and tools described in earlier sections.

5.1.1 Component Lifecycles and Functionality

The ATST *Container/Component Model*(CCM) distinguishes between the *lifecycle* of a Component and the *functionality* provided by a Component. The lifecycle of Components is consistent across ATST. It is this consistency that allows Components to be managed by ATST Containers. Containers can manipulate the lifecycle characteristics of any Component, without regard to the functional behavior. The functionality of a Component, on the other hand, is unique to that Component and implements the needs required by ATST of that Component. Of course, some Components share many common characteristics. ATST software developers are generally expected to use derivatives of Components (i.e. Controller).

This split of lifecycle and functional characteristics has three key advantages:

- The lifecycle characteristics can be implemented once, by the ATST common services
- The necessary overlap between lifecycle behavior and functional behavior occurs in well-defined places
- Component developers can concentrate exclusively on implementing the functional behavior

5.1.2 Component Lifecycle

The section on Containers outlines the steps performed by a Component's Container as part of the lifecycle management of the Component. This section covers these steps from the Component developer's perspective and identifies places where functional behavior can be injected into the lifecycle operations. Additional lifecycle characteristics of Components are also covered here.

5.1.2.1 Creation

A Container creates a Component by invoking that Component's **default** constructor. At this time none of the ATST common services are available to the Component - no log service, property service, etc. For this reason, developers should avoid putting any functionality into the default constructor. In fact, the *ideal* constructor is *empty*! A Component that has been created but not initialized is said to be *loaded*.

5.1.2.2 Initialization

The next step is *initialization*. Initialization is the process of preparing a Component for operation, but stops short of starting that operation. Typical steps taken during initialization include:

- *Metadata* about the Component's Attributes is loaded
- Any special memory needs (fixed buffers, memory maps, etc) are satisfied

The first of these is performed automatically by the ATST common services (though it may be deferred through lazy evaluation, that deferral is transparent to the operation of the Component). The latter is an action based upon the functional behavior required of the Component and so must be done by the Component developer.

Under no circumstances should a Component move any physical mechanisms during initialization. The Component is not yet running and is not available for access by other components at this time.

Before initializing the Component, the Container creates a **`Toolbox`** for the Component, populates it with service helper tools, and attaches the Toolbox to the Container. The Container also binds a *name* to the component at this time, but does **not** register this name with the ATST connection service (yet). The Container then calls the Component's **init()** method. Any Component initialization that isn't performed by the common services should be added to the **doInit()** method by the Component developer. The common services are available for use by Component developers to assist in this part of initialization.

5.1.2.3 Startup

Once the Component has been initialized, the Container (and hence the Component) waits for a directive to start the Component operating. When this directive is received, the Container invokes the Component's **startup()** method. Upon completion of this method, the Component is assumed to be *running* and ready to accept *functional directives*. The Container then announces the availability of the Component by registering the Component's name with the ATST connection service and informing the Container Manager.

It should be pointed out that "ready to accept functional directives" does **not** mean ready to accept *any* functional directive. The behavior of a Component that has completed startup successfully is defined by the functional definition of that Component.

5.1.2.4 Operation

Very little lifecycle activity takes place during Component operation (i.e. while it is *running*). There are two essential tasks that are performed:

- Monitoring Component Health
- Controlling logging (*especially* debug)

While a Component is running, the ATST Health Service publishes a *heartbeat* event at a regular interval. This heartbeat includes the current **health** of the Component. The Component's Container monitors the heartbeat and reports any irregularities to the Container Manager. From a Component developer's view, all that is needed is to keep the Component health information up-to-date. (It should be noted, however, that a heartbeat in a *multithreaded* process cannot be treated as an accurate report on *all* threads. It may well be the case that the *only* functioning thread is the one reporting the heartbeat. For this reason, the heartbeat is perhaps best viewed as a monitor of the network connection and the host hardware, and not seen as a monitor of the software itself.)

Logging control - enabling/disabling categories and changing the debug levels of categories (see the section on the Log Service) - is another lifecycle management activity performed during Component operation. This is handled entirely by the Log Service and needs no Component developer action. However, a component developer can test whether or not a category is enabled and check the current debug level. Decisions in the developer's code may then be based on the results of those actions.

5.1.2.5 Shutdown

Near the end of its lifetime, a Component may be *shut down* by its Container by calling the **`shutdown()`** method. This makes the Component unavailable for functional access. That is, the shut down process is the inverse of the startup step. The Component developer is responsible for safely undoing all actions they introduced during start up (the common services, in conjunction with the base **`Component`** class undo their own actions). If a Container is directed to restart a shut down Component, the startup actions are repeated.

Besides restarting, the only other operation available on a shut down Component is the *uninitialization* of the Component.

5.1.2.6 Uninitialization

When a Component has been shutdown, it is back in the *initialized* lifecycle stage. The Component can then be moved back to the *loaded* stage by calling the **`uninit()`** method. This operation is the inverse of initialization. Typically, the next stage is the *removal* of the Component from the Container. It is also possible to reinitialize the Component.

5.1.2.7 Removal

Components that are *transient* may be removed once they have been shut down. The **`remove()`** method is called to allow the release of any resources acquired by the Component. The **`remove()`** is thus the inverse of the loading a Component. Removing a Component removes it from the Container and from the ATST Connection Service. No action can be taken with that Component from then on - although a new Component may be created in its place.

5.1.3 Functional architecture

While a Component is operating, there are only two basic functional operations that are available. (Subclasses, of course, define more operations). Another application connected to a Component may:

- request the values of Attributes
- set the values of Attributes

Both operations are subject to the standard ATST access policy. Controller Components extend this functionality by adding support for managing `Configurations`. See the section on Controllers for details. Custom components extend this functionality in other ways on a case-by-case basis.

Component developers must also provide support for determining the *health* of the Component's behavior by implementing code used by the Health Service when checking system health. This support is described in the language-specific sections below.

5.1.4 Simulated Components

ATST `Components` may be *simulations*. A simulated Components is not permited to submit configurations to Controllers and has all outgoing events tagged as coming from a simulated Component. A single instance of a Component is not permitted to switch between simulated and non-simulated. Switching is performed by unloading one instance from its `Container` and loading another another.

Once a Component has been marked as a simulated Component, all outgoing events contain an attributed named **`simulated`** with value **`true`**. This attribute is added automatically to the event by the Common Services. Similarly, Common Services enforces the prohibition on having simulated Components submit Configurations to Controllers.

5.1.5 Java-based Components

The base Component class **`atst.cs.ccm.component.Component`** is an abstract class that must be subclassed. This base class has very little support for the ATST functional architecture. Component developers must add functionality to a subclass.

There are *hooks* in the base class for adding functional behavior during the deployment steps described above. Such behavior is added by overriding one or more of the following methods:

- **`void doInit(IAttributeTable args)`** -- functionality needed during Component initialization
- **`void doStartup(IAttributeTable args)`** -- functionality needed during Component startup
- **`void doShutdown()`** -- functionality needed during Component shutdown
- **`void doUninit()`** -- functionality needed during Component uninitialization
- **`void doRemove()`** -- functionality needed during Component removal

In **`doInit`** and **`doStartup`** the arguments passed in via **`args`** are determined by the role the Component plays in ATST. Often, **`args`** is **`null`** in both instances.

Developers may throw an **`atst.cs.ccm.component.LifeCycleChangeException`** from any of the above methods. This exception is caught and the exception message and stack trace are logged. In the cases of **`doInit`** and **`doStartup`**, throwing the exception also aborts the lifecycle change. It does not abort the change when thrown from **`doShutdown`**, **`doUninit`**, or **`doRemove`**.

Developers of Java-based Components must also implement the following method:

- **`void doCheckHealth()`** -- determine current health of the Component

that is called automatically by the HealthService when determining system health. The implementation must take into account the functional requirements of the Component (and subsystems). As an aid, Component developers can update the *health status* of their Components using the static method provided by the Health Service:

- **`Health.setHealth(String newHealth)`**

where **`newHealth`** may be one of **`Health.GOOD`**, **`Health.ILL`**, or **`Health.BAD`**. If calls to this method are imbedded throughout the Component's code, then a simple implementation (*not suitable for Components that manage subsystems!*) would be:

```
public void doCheckHealth() {
        Health.setHealth(Health.GOOD);
}
```

The two functional operations are implemented as:

- **`IAttributeTable get(IAttributeTable params)`** -- request values of Attributes from Component
- **`void            set(IAttributeTable params)`** -- set values of Attributes in Component

The first of these uses the Attribute names in **`params`** to identify Attributes within the Component that are included in the return value. Attributes with unknown names are ignored. The second sets the Component's Attributes to the respective values. Attributes are simply passed to the **`doSet`** method described below. Functionality, including the actual setting of values, must be added by overriding the **`doSet`** method.

Subclasses of Component should **not** override the above **`get`** and **`set`** operations. Instead, subclasses should override the respective methods:

- **`IAttributeTable doGet(IAttributeTable params)`**
- **`void            doSet(IAttributeTable params)`**

Some Component subclasses may want to do addition processing when the log *debug level* changes. Two convenience methods may be overridden to provide for additional functionality at that point:

- **`void doSetDebugLevel(int level)`** -- defaults to an empty call
- **`void doSetDebugLevel(String category, int level)`** -- defaults to an empty call

More details can be found in the sections on predefined subclasses of **`atst.cs.ccm.component.Component`** such as Controllers.

5.1.5.1 Simulated Java Components

A Java-based Components may be marked as simulated. This enables the enforcement of the functional restrictions imposed on simulated Components.

- **`void markAsSimulated()`** -- mark this Components as simulated.

This method *only* has effect when it is embedded in the Components's *default constructor*, as in:

```
public MySimComponents() {
    markAsSimulated();
    }
```

All simulated Components must be marked in this manner.

5.1.6 C++ based Components

The base Component class **`atst::cs::ccm::component::Component`** is an abstract class that must be subclassed. This base class has very little support for the ATST functional architecture. Component developers must add functionality to a subclass.

There are *hooks* in the base class for adding functional behavior during the deployment steps described above. Such behavior is added by overriding one or more of the following methods:

```
// Functionality needed during Component initialization
void doInit(tr1::shared_ptr<IAttributeTable> args);

// Functionality needed during Component startup
void doStartup(tr1::shared_ptr<IAttributeTable> args);

// Functionality needed during Component shutdown
void doShutdown();
```

In **doInit** and **doStartup** the arguments passed in via **args** are determined by the role the Component plays in ATST. Often, **args** is **zero** in both instances.

Additional functionality may be added by overriding the following methods which will be available in the Sacramento-2 release:

```
void doUninit(); // functionality needed during Component uninitialization
void doRemove(); //functionality needed during Component removal
```

The ***doCheckHealth()*** method will be required of Components in releases beyond Sacramento-1:

```
void doCheckHealth();  // determine current health of the Component
```

The ***doCheckHealth()*** method is called automatically by the HealthService when determining system health. The implementation must take into account the functional requirements of the Component (and subsystems). As an aid, Component developers can update the *health status* of their Components using the static method provided by the Health Service:

```
Health::setHealth(const string& newHealth);
```

where **newHealth** may be one of **Health::GOOD**, **Health::ILL**, or **Health::BAD**. If calls to this method are imbedded throughout the Component's code, then a simple implementation (*not suitable for Components that manage subsystems!*) would be:

```
void doCheckHealth()
{
        Health::setHealth(Health::GOOD);
}
```

The two functional operations are implemented as:

```
// Request values of Attributes from Component
tr1::shared_ptr<IAttributeTable> get(tr1::shared_ptr<IAttributeTable> params);

// Set values of Attributes in Component
void set(std::tr1::shared_ptr<IAttributeTable> params);
```

The first of these uses the Attribute names in **params** to identify Attributes within the Component that are included in the return value. Attributes with unknown names are ignored. The second sets the Component's Attributes to the respective values. Attributes are simply passed to the **doSet** method described below. Functionality, including the actual setting of values, must be added by overriding the **doSet** method.

Subclasses of Component should **not** override the above **get** and **set** operations. Instead, subclasses should override the respective methods:

```
tr1::shared_ptr<IAttributeTable>
doGet(std::tr1::shared_ptr<IAttributeTable> params);

void doSet(tr1::shared_ptr<IAttributeTable> params);
```

More details can be found in the sections on predefined subclasses of **atst::cs::ccm::component::Component** such as Controllers

5.1.7 Python-based Components

TBD

-- BretGoodrich - 11 Jan 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.30 | ≥ | r1.29 | ≥ | r1.28 | Total page history | Backlinks
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsManual > AtstCsManualComponents

to top

Skip to topic | Skip to bottom

Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
    - News
    - Design
    - Common Services
    - Application base
    - OCS
    - DHS
    - TCS
    - ICS
    - ICDs
    - Docs
    - Meeting Notes
    - Problem Tracking
    - Changes

- **Misc**
    - Users
    - Groups

- ♦ Offices
- ♦ Topic list
- ♦ Search

- **TWiki Webs**
    - ♦ Know
    - ♦ Main
    - ♦ Sandbox
    - ♦ TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsManualControllersr1.50 - 06 Feb 2007 - 16:19 - SteveWamplertopic end

Start of topic | Skip to actions

5.2 Controllers

A **Controller** is a subclass of a Component, used to manipulate configurations. The Controller class is only one possible way of manipulating components, others may be implemented during the course of ATST development. However, a controller is ideally suited for many situations, especially those that need to handle multiple, simultaneous configurations.

A component does nothing with configurations, it simply manages its own lifecycle and accepts low-level *get* and *set* operations on attribute tables. Since a configuration is more than a grouping of attribute-value pairs, there needs to be a class that controls configuration lifecycle issues. Hence, the **Controller** class. Some useful subclasses of **Controller** for real-time control include the **`atst.base.controllers.SequenceController`** and various subclasses of the **`atst.base.controllers.MotorController`**.

Controllers are part of the *application framework* layer of the ATST Common Services.

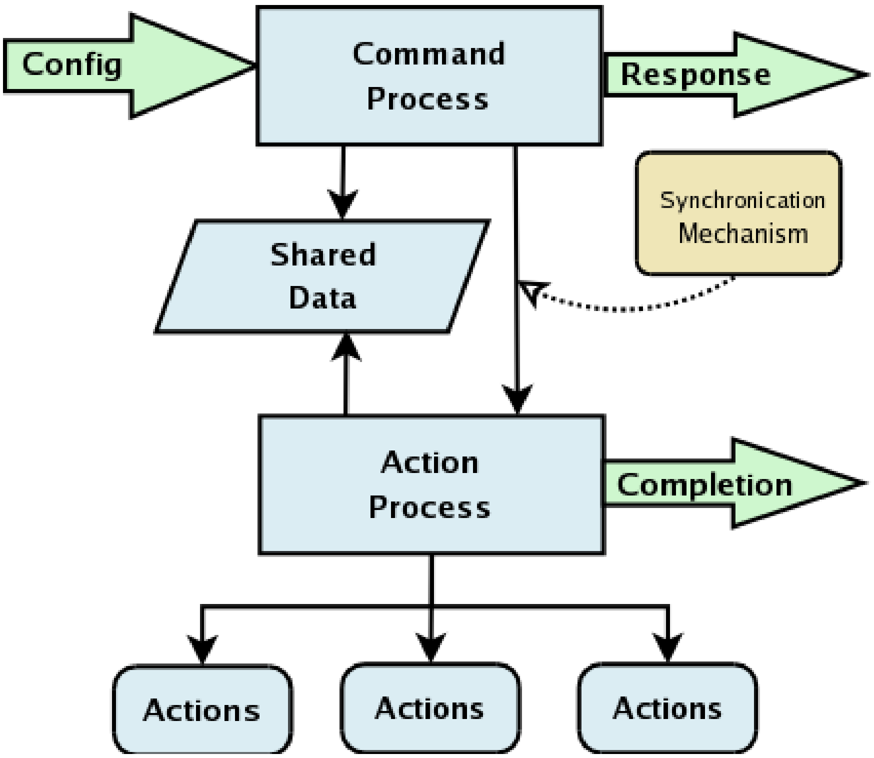
5.2.1 Functionality

5.2.1.1 Command-Action-Response

The controller implements a command-action-response model. In this model, commands are submitted to the controller where they are either accepted or rejected based upon the validity of the argument and the state of the controller. If a command is accepted by the controller it causes an independent action to begin. A response to the command is returned immediately. The action begins matching the current configuration to the new demand configuration. When the configurations match (i.e., the controller has performed the input operations) the action signals the successful end of the action. If the configurations cannot be matched (whether by hardware failure, external cancel command, timeout, or some other fault) the action signals the unsuccessful end of the action.

The important features of the command/action/response model are:

- Commands are never blocked. As soon as one command is submitted, another one can be issued. The behavior of the controller when two or more configurations are submitted can be configured on a per-controller basis.
- The actions are performed using one or more separate threads. They can be tuned for priority, number of simultaneous actions, critical resources, or any other parameters.
- Action completions produce events that tell the state of the current configuration. Actions push the lifecycle of the configuration through to completion.
- Responses may be monitored by any other Components/Controllers.



ATST Controllers use threads to implement both command and action processing.

5.2.1.2 Command Thread

The command thread receives configurations from the external interface and performs several basic sanity tests on them. In addition to checking for properly formed configurations, the command thread calls the Property Service to test individual attribute's ranges and types. The command thread also calls a *doSubmit* method provided by a subclass to test for other conditions that might preclude executing the configuration.

Once these tests are performed the command thread queus the configuration and signals the action manager.

The command thread also handles the *cancel* command. The configuration identified by the argument to *cancel* is either queued or active. If the former, it is removed from the queue and an *abort* event is sent for it. It is then destroyed. If the configuration is currently active, the action thread it is running under is issued a *cancel* signal, whereupon it propogates the cancel command to any subsystems involved in the processing, then aborts and destroys the configuration. The *doCancel* command is available for subclasses to implement their own behavior when the *cancel* command is received.

Finally, the command thread handles the *pause* and *resume* commands. The *pause* command either keeps the configuration from leaving the queue or forces an action thread to pause the configuration and all of the associated subsystems. The *resume* command causes either the queue or action thread to restart executing the configuration. Pausing an active configuration is the reponsibility of subclasses which may implement the *doPause* and *doResume* commands to do so.

5.2.1.3 Action Manager

The action manager is a thread responsible for managing the execution of the incoming configurations. The actual implementation of the action manager is beyond the scope of this document, but it is useful to understand its basic operation. Configurations are queue in *priority* order based upon their *starttime*. If a configuration requires immediate execution, the action manager finds an available action thread and assigns the execution details to that thread. If a configuration has a *starttime* attribute the action manager delays execution until the requested time.

The action manager may be configured on a per-controller basis for its behavior. Controllers that protect critical resources may have only one action thread. Queued configurations are held or aborted, depending upon the value of the *fullthreadaction* attribute. Other controllers may have a large pool of action threads.

When an action thread completes it signals the action manager with a done or aborted signal using an *action callback*. The action manager then regenerates that signal as an event and deletes the configuration. The action thread is returned to the available thread pool.

Commands to *cancel*, *pause*, or *resume* the execution of a configuration are passed through the action manager. If necessary, the action manager signals the appropriate action thread to perform the requested command.

5.2.1.4 Action Threads

Each controller has a pool of available action threads, the number may vary for each controller. Once taken from the pool by the action manager, an action thread is assigned a configuration. The heart of the action thread is the *action* command provided by a subclass of the **Controller** class. All action threads run the same *doAction* command, so it must be thread-safe. This command does not need to know about the lifecycle of the action thread, however, nor its interactions with the action manager. It only needs to perform work upon the input configuration. Upon completion of an action, the *doAction* command returns a flag indicating whether or not the action was successfully completed ("done") or not ("aborted").

5.2.1.5 Action callbacks

When a controller submits a configuration to another controller (for example, when the TCS submits a configuration to the MCS), a *callback* is attached to the submission to provide for action response synchronization. The callback provides two basic commands: *done* and *abort* for use by the action callback as signals. Developers may add functionality to these commands by implementing the *doDone* and *doAbort* commands. A third command *report* may be used to issue *progress reports* during the processing of an action.

5.2.2 Control of Configuration Lifecycle

A configuration's lifecycle is well-defined in its definition, it is initialized, is running, and is completed. Each of these stages is entered through some type of external transition. The type of transition determines the event posted about the configuration to any interested parties (such as the OCS). Configuration lifecycle transitions within a controller are handled entirely by the base controller class. However, the base controller implementation only posts configuration lifecycle transition events on *done* or *aborted* transistions unless the *traceconfigs* parameter has been set to *true*.

5.2.2.1 Scheduled

When a configuration arrives in the controller though the *submit* interface it is in the initialized state. The controller generates the "scheduled" event through the Component interface to signal that it has accepted a configuration and is scheduling it for execution. The implementation details of the scheduler are not important for this discussion, it is enough to say that the scheduled phase of the configuration lifecycle may be arbitrarily short or long.

5.2.2.2 Running

Once the configuration is assigned to an action process things begin to happen. First, the state of the configuration goes to "running". Next, the action process begins to match the current and demand configurations. During this time other events (like position) are generated.

5.2.2.3 Completed

A configuration that is no longer being matched by the Controller is "completed". An event is generated indicating whether the matching was successful ("done") or unsuccessful ("aborted") and the configuration is removed from the action process and destroyed.

A configuration may not be successfully completed for a number of reasons:

- It was rejected prior to scheduling.
- It could not be scheduled.
- An external *cancel* command was issued for the configuration_ID matching that of the configuration.
- The configurations's *timeout* value was exceeded.
- The controller's action process determined the configuration could not be met.
- A component owned by the controller and involved in matching the configuration aborted its own configuration.

5.2.3 Interface

The Controller class has a public and protected interface. The public interface has an associated interface definition and communications implementation. The protected interface allows up-calls from the action task or subclasses.

5.2.3.1 Public Interface

In addition to the methods defined in the Component public interface, the public interface for a Controller adds:

- **submit** — Schedule a configuration to be executed. Returns OK (`0`) if the configuration can be scheduled and a non-zero flag otherwise (see below for a list of the known error flags)
- **cancel** — Stop the execution of a scheduled configuration.
- **pause** — Pause the execution of a scheduled configuration.
- **resume** — Resume the execution of a paused configuration.

5.2.3.2 Protected Interface

The public interface methods are predefined by the base Controller class and cannot be overridden. However, each makes calls to some protected methods that can be overridden:

- **doSubmit** — Subclass checks during a *submit* command. Returns OK (`0`) only if the configuration action can be scheduled.

- **doCancel** — Subclass checks during a *cancel* command. Returns `true` if the configuration action was cancelled.
- **doPause** — Subclass checks during a *pause* command. Returns `true` if the configuration action was paused.
- **doResume** — Subclass checks during a *resume* command. Returns `true` if the configuration action was resumed.
- **doAction** — Subclass actions upon the configuration. Returns `true` if the configuration action was successfully completed.

More details about the above methods can be found in the language-specific sections that follow.

5.2.3.3 Attributes

Attributes of a controller can be read and modified through the *get* and *set* commands in the Component interface.

- **threadmodel** — whether the pool of action threads is *fixed* or *growable*.
- **numthreads** — the current number of action threads.
- **maxthreads** — the upper bound on the number of action threads, when *growable*.
- **fullthreadaction** — queue or reject configurations when there are no available threads.
- **activethreads** — the number of active action threads.
- **schedlist** — a list of configuration IDs in the schedule.
- **timeout** — the default action timeout if not given in the configuration.
- **traceconfigs** — determines whether or not configuration lifecycle events are posted on non-terminal transitions (DONE or ABORTED transistions always have status events posted).
- **minactiondelay** — the minimum allowed delay (in milliseconds) when pausing an action.
- **schedulecheckrate** — the delay between checks of the Action queue for runnable actions.

There are attributes in a configuration that the controller uses to run the configuration.

- **timeout** — the time limit (in *milliseconds*) to be imposed on the action to match the configuration
- **starttime** — the earliest time at which the action should begin (`yyyy/mm/dd:hh:mm:ss.s tz` or `yyyy/DDD:hh:mm:ss.s tz`, where `DDD` is the Julian day)
- **stoptime** — the time the action should complete (`yyyy/mm/dd:hh:mm:ss.s tz` or `yyyy/DDD:hh:mm:ss.s tz`, where `DDD` is the Julian day)

Here **timeout** is the time limit imposed on the action *from the moment it actually starts execution*. It does not include time that the action is spent queued waiting to start. If omitted, the default timeout defined by the specific Controller is used. A value of `0` implies no limit. Conversely, **stoptime** is the point at which a queued action is not longer to be considered available for execution. If omitted, then the action will remain queued until executed or forcibly removed from the queue.

5.2.3.4 Events

The controller generates the following events:

- **configstate** — State change of a configuration. The event includes the Configuration ID and the state (**`"scheduled"`**, **`"running"`**, **`"paused"`**, **`"report"`**, **`"done"`**, or **`"aborted"`**).

5.2.4 Action Callback Interface

The action callback also has public and protected interfaces. The implementation of the public interface methods is complete and provided as part of the ATST Common Services. Controller developers can add additional functionality by implementing the methods given in the protected interface.

5.2.4.1 Public Interface

- **done** — reports successful completion of a submitted configuration's match
- **abort** — reports unsuccessful completion of a submitted configuration's match
- **report** — reports on the status of the action on a submitted configuration

5.2.4.2 Protected Interface

- **doDone** — action response to successful submitted action completion.
- **doAbort** — action response to unsuccessful submitted action completion.
- **doReport** — action status report

5.2.5 Controller Properties

The attributes described above all have *metadata* associated with them. This metadata describes the *property* of each attribute and resides in the Property Service *persistent store*. Properties are unique to every Component and Controller.

Some attributes have default values that are loaded from the property service and set when the Component is initialized. The proper way to change these default values is to modify the entries in the persistent store. The following steps illustrate one way to modify properties in the persistent store, assuming the Controller's name is **`atst.demoController`**:

- Extract the properties into a CSV (*comma-separated-values*) file (If this is a new Controller, you can use the *Controller Template* by replacing **`atst.demoController`** with **`ControllerTemplate`**, just remember to change the controller name in the next step):

```
PropertyWriter --name=atst.demoController > data.csv
```

- Edit `data.csv` and make any changes, including adding new properties (you cannot remove properties, however). You can also insert empty lines and comments (lines with '#' in column 0) if you want to keep the csv file for reference. Any commas or double quotes that are embedded within a field (for example, the commas separating various limit values) must be escaped with a backslash. Details on the meanings and acceptable values for each field can be found in the Users' Manual. The fields are, in order:
  - controller name (**`atst.demoController`** in this example), ***required***
  - attribute(property) name, ***required***
  - property type, ***required***
  - boolean flag on whether property is a vector of type or not
  - boolean flag on whether property is a readonly attribute
  - a short description of the property
  - any saved value for the property
  - any limits on legal values assigned to this attribute
  - any change deltas for monitoring this attribute

    - any default value for the property
- Insert the properties back into the persistent store:

```
PropertyReader <data.csv
```

If all you need is a clone of the properties from an existing Controller, you can use the **`PropertyClone`** command. For example:

```
PropertyClone --source=ControllerTemplate --target=MyNewController
```

All fully-qualified property names are correctly renamed by **`PropertyClone`**.

- Sample ControllerTemplate CSV file:

```
# CSV template for Controller attribute Properties
#    - defines properties for lifecycle attributes of all Controllers
#    - can be copied, edited, and extended to handle specific properties for
#        specific controller instances.
#
# Format is:
# Controller name, property name, type, vector?, readonly?, description, saved_values, limits, deltas, defaults
#
ControllerTemplate,threadmodel,string,f,f,"How action threadpool is managed",,"fixed,growable",,"fixed"
ControllerTemplate,numthreads,integer,f,f,"Number of thread in pool",,"1",,1
ControllerTemplate,maxthreads,integer,f,f,"Maximum number of thread allowed pool",,"0",,0
ControllerTemplate,fullthreadaction,string,f,f,"Behavior when no free action threads",,"reject,queue",,"queue"
ControllerTemplate,timeout,integer,f,f,"Default action timeout",,0,,0
ControllerTemplate,traceconfigs,boolean,f,f,"Trace configuration lifecycles",,,,f
#
# These next describe readonly properties
#
ControllerTemplate,activethreads,integer,f,t,"Number of active threads",,,,
ControllerTemplate,schedlist,string,t,t,"Scheduled configurations",,,,
```

- Sample PropertyAdminServer CSV file:

```
# CSV template for PropertyAdminServer attribute Properties
#
# Format is:
# Controller name, property name, type, vector?, readonly?, description, saved_values, limits, deltas, defaults
#
PropertyAdminServer,threadmodel,string,f,f,"How action threadpool is managed",,"fixed,growable",,"growable"
PropertyAdminServer,numthreads,integer,f,f,"Number of threads in pool",,"1",,3
PropertyAdminServer,fullthreadaction,string,f,f,"Behavior when no free action threads",,"reject,queue",,"queue"
PropertyAdminServer,timeout,integer,f,f,"Default action timeout",,0,,0
PropertyAdminServer,traceconfigs,boolean,f,f,"Trace configuration lifecycles",,,,f
#
# These next describe readonly properties
#
PropertyAdminServer,activethreads,integer,f,t,"Number of active threads",,,,
PropertyAdminServer,schedlist,string,t,t,"Scheduled configurations",,,,
```

5.2.6 Simulated Controllers

ATST `Controllers` may be *simulations*. A simulated Controller is not permited to submit configurations to other Controllers and has all outgoing events tagged as coming from a simulated Controller. A single instance of a Controller is not permitted to switch between simulated and non-simulated. Switching is performed by unloading one instance from its `Container` and loading another another.

Once a Controller has been marked as a simulated Controller, all outgoing events contain an attributed named **`simulated`** with value **`true`**. This attribute is added automatically to the event by the Common Services. Similarly, Common Services enforces the prohibition on having simulated Controllers submit Configurations to other Controllers.

5.2.7 Java-based Controllers

5.2.7.1 The public interface

ATST `Controllers` all extend the base **`atst.cs.controller.Controller`** class and implement the **`atst.cs.interfaces.IController interface`**. The following methods are defined by that interface in addition to those methods inherited from the **`atst.cs.interfaces.IComponent`** interface:

- **`int submit(atst.cs.interfaces.IConfiguration config)`** -- match the supplied `Configuration`
- **`void cancel(String configID)`** -- abort processing of the named `Configuration`
- **`void pause(String configID)`** -- pause processing of the named `Configuration`
- **`void resume(String configID)`** -- resume processing of the named `Configuration`

In addition, the following *convenience methods* are available as part of the public interface:

- **`int submit(atst.cs.interfaces.IConfiguration config, long pause)`** -- match the supplied `Configuration`, but automatically resubmit after `pause` milliseconds if the Controller rejects the configuration because it is busy (see `Controller.BUSY`, below).
- **`int submit(atst.cs.interfaces.IConfiguration config, atst.interface.IActionCallback callback)`** -- match the supplied configuration, invoking the supplied `callback` in the *caller* on action completion.
- **`int submit(atst.cs.interfaces.IConfiguration config, long pause, atst.interface.IActionCallback callback)`** -- the combination of the above.

The **`submit`** methods return an integer representing the result of the submission. The permissible values are:

- **`Controller.OK`** (**`0`**): The configuration has been accepted for action.
- **`Controller.BUSY`** (**`-1`**): The configuration has been rejected because the Controller cannot not perform any additional simultaneous actions and cannot queue the submitted configuration.
- **`Controller.BAD_PARAM`** (**`-2`**): The configuration has an invalid parameter value.
- **`Controller.EXCEPTION`** (**`-3`**): There is a runtime error in the submit code for the target Controller.
- **`Controller.NOT_RUNNING`** (**`-4`**): The target controller is not at its *running* stage.
- **`Controller.DUPLICATE`** (**`-5`**): The configuration ID matches one already being acted on.
- **`Controller.NO_CONFIG`** (**`-6`**): There was no configuration submitted.
- **`Controller.SIMULATED`** (**`-7`**): The submit came from a component running in *simulation* mode.

All of these methods are fully implemented by the base `Controller` class supplied as part of the ATST Common Services. Controller developers may add functionality at these Controller lifecycle stages by implementing the appropriate methods defined in the *protected interface* below.

5.2.7.2 The protected interface

The lifecycle protected methods **`doInit`**, **`doStartup`**, **`doShutdown`**, **`doUninit`**, and **`doRemove`** found in the Component class are also provided in the **`Controller`** class. These methods may also throw the **`atst.cs.ccm.component.LifeCycleChangeException`** described in the Component section.

Additional functionality can be added by Controller developers by overriding the following `Controller` methods:

- **`int doSubmit(atst.cs.interfaces.IConfiguration config)`** -- match the supplied `Configuration`

- **`boolean doCancel(String configID)`** -- abort processing of the named `Configuration`
- **`boolean doPause(String configID)`** -- pause processing of the named `Configuration`
- **`boolean doResume(String configID)`** -- resume processing of the named `Configuration`
- **`String doAction(atst.cs.interfaces.IConfiguration config)`** -- additional action functionality

The **`doSubmit`** method returns the same values as **`submit`**, shown above. The methods **`doCancel`**, **`doPause`**, and **`doResume`** are only called if the action on the identified `Configuration` is currently active. If the action is scheduled, but not yet running, it is handled internally. Each should return **`true`** if the action was successfully cancelled, paused, or resumed, respectively. Unless overridden, each of the above do nothing, so the default behavior is that executing actions may not be paused, resumed, or cancelled.

The result of evaluating **`doAction`** should be **`Controller.ACTION_OK`** only if the action is successfully completed. Otherwise, the result should be a short description of the reason the action failed. The *empty string* (**`""`**) is a valid reason for action failure. However, the *null string* (**`null`**) is **not** - it is synonymous with **`ACTION_OK`**.

5.2.7.3 Simulated Java Controllers

A Java-based Controller may be marked as simulated. This enables the enforcement of the functional restrictions imposed on simulated Controllers.

- **`void markAsSimulated()`** -- mark this Controller as simulated.

This method *only* has effect when it is embedded in the Controller's *default constructor*, as in:

```
public MySimController() {
    markAsSimulated();
    }
```

All simulated Controllers must be marked in this manner.

5.2.7.4 Controller `Attributes`

Some `Attributes` are predefined as part of the base Controller class. The names for all these attributes are prefixed with the name of the specific Controller instance. For example, the `threadmodel` Attribute for the OCS alarm handler is named **`atst.ocs.alarmtree.threadmodel`**. The predefined Controller `Attributes` are:

- **`threadmodel`** -- is the action thread pool *fixed-size* or *growable*? Acceptable values are **`Controller.FIXED`** and **`Controller.GROWABLE`**.
- **`numthreads`** -- the current number of (total) action threads
- **`maxthreads`** -- the bound on the (total) number of action threads allowed when *growable*
- **`fullthreadaction`** -- queue or reject configurations when there are no available threads. Acceptable values are **`Controller.QUEUE`** and **`Controller.REJECT`**. The default is **`Controller.QUEUE`**.
- **`activethreads`** -- the current number of active action threads
- **`schedlist`** -- a list of configuration IDs in the schedule
- **`timeout`** -- the *default* action timeout if a Configuration does not specify a timeout. the value of **`0`** indicates no default. The value is in *milliseconds*.
- **`traceconfigs`** -- set to **`true`** if non-terminal Configuration lifecycle transitions are to have status events posted. The default is **`false`**.
- **`minactiondelay`** -- the minimum allowed delay when pausing or delaying an action. The value is in *milliseconds* and must be positive. The default is 10ms.

- **schedulecheckrate** -- the delay between checks of the Action queue for runnable actions. The value is in *milliseconds* and must be positive. The default is 100ms.

5.2.7.5 Configuration Attributes

The Controller is also interested in some of the Attributes that may be present in a Configuration. These are:

- **timeout** -- the timeout to be imposed on the action of matching this Configuration. If omitted, then the Controller's default action timeout is assumed. This value is a **long** representing the duration in milliseconds.
- **starttime** -- the time at which the action of matching this Configuration should begin, The value is a **String** representing an **atst.cs.util.AtstDate** object.
- **stoptime** -- the time at which the action of matching this Configuration should end, The value is a **String** representing an **atst.cs.util.AtstDate** object.

5.2.7.6 Generated events

A Controller generates the following events. All generated event names are prefixed with the name of the specific Controller instance (see example above). The following events are used:

- **configstate** -- the status of the action on a Configuration by this Controller. The event includes the *configuration ID*, possibly a reason for the reported status (in the case of an *aborted* status), and the new *state* of the configuration, one of:
  - **atst.cs.controller.Action.SCHEDULED**
  - **atst.cs.controller.Action.RUNNING**
  - **atst.cs.controller.Action.PAUSED**
  - **atst.cs.controller.Action.ABORTED**
  - **atst.cs.controller.Action.DONE**

In practice, only the **ABORTED** and **DONE** states are normally reported. Note, however, that Controller developers do not generate these events, they are generated automatically.

5.2.7.7 Action Callbacks

All action callbacks extend **atst.cs.controller.ActionCallback** and implement the **atst.cs.interfaces.IActionCallback** interface. The following methods are defined by this interface:

- **void done(atst.cs.interfaces.IConfiguration config)** -- report successful matching of the submitted Configuration
- **void abort(atst.cs.interfaces.IConfiguration config)** -- report unsuccessful matching of the submitted Configuration
- **void report(atst.cs.interfaces.IConfiguration config)** -- report on the current status of the submitted Configuration (does not terminate the action).

All of these methods are implemented by the **ActionCallback** base class and cannot be overridden. Controller subclasses can add additional operations to be performed by on successful completion of a submitted configuration by overriding the following methods from the **ActionCallback** class:

- **void doDone(atst.cs.interfaces.IConfiguration config)** -- handle successful matching of the Configuration
- **void doAbort(atst.cs.interfaces.IConfiguration config)** -- handle unsuccessful matching of the Configuration

- **`void doReport(atst.cs.interfaces.IConfiguration config)`** -- handle a status report on the `Configuration` action

5.2.7.8 Writing Java-based Controllers

This section describes the basic steps involved in writing a custom, Java-based Controller. It introduces some of the support that is available to developers when they subclass **`atst.cs.controller.Controller`**, and offers some suggestions on how to handle common situations.

All ATST controllers extend the **`atst.cs.controller.Controller`** class. Controllers that want to submit configurations to *other* Controllers must also override **`atst.cs.controller.ActionCallback`**. The next few sections describe the key methods that should be overridden by subclasses of **`Controller`** and **`ActionCallback`**. Naturally, additional support methods may also be added as needed.

5.2.7.8.1 Controller properties

Every controller includes a set of attributes that define its behavior. Some of these attributes are *technical* and found in all controllers, while some may be *functional* and specific to each controller. These attributes may have a property associated with them that describes any *metadata*, such as value limits and default values. A full description of properties can be found in the Users' Manual.

Controller properties are organized by controller *name*, not by controller *class names*. This means that different controllers implemented using the same Java class may have different property metadata. It also means that these properties must be maintained in the Property Service persistent store by controller name.

See the section on Controller Properties above for details on how to maintain a controller's properties.

During controller initialization, any default values for attributes should be set. The technical attributes default values are set automatically by the **`init`** method. If you add additional functional attributes, you must handle the setting of their default values in the **`doInit`** method. The source code for **`init`** in **`atst.cs.controller.Controller`** can serve as a guide.

5.2.7.8.2 The ControllerAdapter class and source file

The class **`atst.cs.controller.ControllerAdapter`** subclasses **`atst.cs.controller.Controller`** and includes rudimentary implementations of the methods in the protected interface where you can attach application specific functionality to a controller subclass. While **`ControllerAdapter`** itself may be subclassed, a more practical use is to use its *source code* as a template for constructing your own **`Controller`** subclass:

1. Copy **`$ATST/src/java/atst/cs/controller/ControllerAdapter.java`** into the source directory for your new controller subclass, renaming it to your new controller name.
2. In that new source file, change:
   - the package name to your package
   - the class name to your controller subclass name
3. Now edit that source file to introduction the functionality required for your application.

5.2.7.8.3 The ActionCallbackAdapter class and source file

The class **`atst.cs.controller.ActionCallbackAdapter`** subclasses **`atst.cs.controller.ActionCallback`** and includes rudimentary implementations of the **`doDone`** and

doAbort methods in the protected interface where you can attach application functionality. It serves the same roles to the **ActionCallback** class as **ControllerAdapter** serves to the **Controller** class.

5.2.7.8.4 Controller methods

This section covers the roles of the protected methods that can be overridden in Controller subclasses to provide any required functionality specific to the application.

5.2.7.8.4.1 doSet

The **doSet** method is part of the **IComponent** interface implemented by all controllers. It is called from the final method **set**. The **set** method is responsible for setting parameters for a component/controller that are outside the command/action/response model. For a Controller, this includes such parameters as the number of doAction threads_ and the behavior to take on a doSubmit command_ when all action threads are busy, for example, as well as any functional parameters defined by the controller developer. The parameters are passed into **set** as an IAttributeTable object.

The **set** method automatically handles all parameters that are part of the technical design of all controllers, setting the appropriate internal variables if the corresponding attributes from the argument are valid. Once these technical parameters are set, the IAttributeTable object is then passed to **doSet** to allow custom parameters to be set.

The **doSet** method can make use of the **atst.cs.services.Property** class to help perform its task. Here is a simple implementation of **doSet** that examines the attributes for two parameters: *sample.mode* and *sample.wavelength*:

```
protected void doSet(atst.cs.interfaces.IAttributeTable table) {

        IAttribute modeA = table.get("sample.mode");
        if (null != modeA) {
                String mValue = (modeA.getValue())[0];
                if (Property.inRange("sample.mode",mValue)) {
                        setModeParam(mValue);
                }
        }

        IAttribute waveA = table.get("sample.wavelength");
        if (null != waveA) = {
                String wValue = (waveA.getValue())[0];
                if (Property.inRange("sample.wavelength", wValue)) {
                        setWaveLength(wValue);
                }
        }

}
```

(Presumably, the **setMode** and **setWaveLength** methods have also been added to this Controller subclass.)

5.2.7.8.4.2 doGet

The **doGet** method is the inverse of **doSet**. It is passed an **IAttributeTable** and fills in the values of any attributes in that that describe parameters specific to this Component subclass. It is called from the **get** method after **get** has filled in the attribute values for any parameters from the technical architecture. A sample implementation is:

```
protected atst.cs.interfaces.IAttributeTable doGet(atst.cs.interfaces.IAttributeTable table) {
        String pNames = table.getNames();
```

```
        for (int i = 0; i < pNames.length; ++i) {
                String pName = pNames[i];
                if (isFunctionalParam(pName)) {
                        table.setValue(pName, getFunctionalValue(pName));
                }
        }
        return table;
}
```

(Again, the **isFunctionalParam** and **getFunctionalValue** methods have presumably been added to this Controller subclass.)

5.2.7.8.4.3 doSubmit

The **doSubmit** method is called from within the **submit** method and provides Controller subclass developers a chance to perform detailed validity checks on a Configuration's attributes. Prior to calling **doSubmit**, the **submit** method has already performed *simple* validity checks and all known attributes (attributes for which property metadata exists) are known to be writable and within the outer range of any bounds have been performed. Often, this is sufficient, but sometimes attributes must have more complex checks performed. For example, it may be that the range of legal *wavelengths* may vary with depending upon the *grating order*, so two attributes, **wavelength** and **order** must be checked in concert. As another example, some attributes may have two sets of limits (e.g. *lowlow* and *highhigh* as well as *low* and *high*. While **submit** has checked the value against the *lowlow* and *highhigh* limits, checks against *low* and *high* may also be needed. These types of tests may be added by the developer by overriding the **doSubmit** method.

If **doSubmit** determines that a configuration is invalid, it should return a non-zero value corresponding to the reason. The most common reason is **Controller.BAD_PARAM** (**-2**). If the configuration if valid, **doSubmit** should return **Controller.OK** (**0**). Only if **doSubmit** returns **Controller.OK** does **submit** schedule an action for matching the configuration. If **doSubmit** rejects a configuration, it should also log a *warning message* giving more details on the reason for the rejection.

As an aid in writing **doSubmit**, the Property service helper class provides direct access to any limit values for a property, both as a **String** array and as an array of objects matching the Java form for the corresponding ATST type. It is the developer's responsibility to understand the meaning of each array element in a limit array. The first two elements are always the extreme range of limits (those that the **submit** method has already checked against).

5.2.7.8.4.4 doCancel

The **doCancel** method is called by the **cancel** method to cancel configurations that are active. (Configurations that are queued pending the availability of an action thread are directly handled by **cancel**.) If called, **doCancel** should return **true** if the action thread may be safely aborted and **false** otherwise. The default implementation of **doCancel** always returns false, so *unless* **doCancel** *is overridden*, actions on active configurations can *not* be aborted.

Note that, in Java, stopping a running thread with an external call is *inherently unsafe*. Consequently, care must be taken when writing both **doCancel** and **doAction** (below).

A typical implementation of **doCancel** in a situation where actions may be interrupted is:

```
protected boolean doCancel(String configId) {
        Action action = getAction(configId);
        if (null != action) {   // May be gone!
                // Code to decide if the action can be cancelled
        }
```

```
        return true;    // Normal response
}
```

Note that this **doCancel** returns true if the action is interruptable. Its response *does not* indicate that the action has actually been interrupted, as it may take **doAction** an indeterminite amount of time before terminating. (In fact, if **doAction** is incorrectly written, it may never check to see if it has been asked to terminate!)

5.2.7.8.4.5 doPause

The **doPause** method is similar to **doCancel** except that it is called from the **pause** method if the configuration is currently active. As with thread termination, the pausing of a thread in Java is also inherently unsafe, so the same care must be taken when overriding **doPause**.

A typical implementation of **doPause** in a situation where actions may be paused is:

```
protected boolean doPause(String configId) {
        Action action = getAction(configId);
        if (null != action) {   // May be gone!
                // Code to decide if the action can be paused
        }
        return true;
}
```

Note that **doPause** returns true if the action is pausable. Its response *does not* indicate that the action has actually been paused, as it may take **doAction** an indeterminite amount of time before pausing. (In fact, if **doAction** is incorrectly written, it may never check to see if it has been asked to paused!)

5.2.7.8.4.6 doResume

The **doResume** method may be overridden to allow resuming of paused configuration actions. In fact, it *must* be overridden if **doPause** has been overridden. A typical implementation of **doPause** in a situation where actions may be paused is:

```
protected boolean doResume(String configId) {
        Action action = getAction(configId);
        if (null != action) {   // May be gone!
                // Code to decide if the action can be resumed
        }
        return true;
}
```

Note that **doResume** returns true if the action is pausible. Its response *does not* indicate that the action has actually been resumed, as it may take **doAction** an indeterminite amount of time before resuming. (In fact, if **doAction** is incorrectly written, it may never check to see if it has been asked to resume!)

5.2.7.8.4.7 doAction

It is the **doAction** method that implements the *functional* behavior of a Controller subclass. Since it is called to handle every configuration action, it must be thread safe.

Special care must be taken when implementing actions that may be cancelled or paused. (Typically, actions that do not contain a loop or a synchronization point are not cancelable or pausable.) In particular, such actions must check periodically to see if they have been asked to terminate or pause. *These checks must be done in places where it is safe to do so.*

The code to check for a pause request can usually be written by taking advantage of support routines provided by the controller:

```
Action action = getAction(config.getId());
if (action.paused()) {
        action.pause(delay);
        if (action.interrupted()) {
                // Handle a cancel that occured during the pause.
                // Do anything necessary to safely terminate at this point, then:
                return Controller.ACTION_CANCELLED;
        }
}
```

similarly, code to check for a termination request is typically:

```
Action action = getAction(config.getId());
if (action.interrupted()) {
        // Do anything necessary to safely terminate at this point, then:
        return Controller.ACTION_CANCELLED;
}
```

5.2.7.8.5 ActionCallback methods

This section covers the roles of the protected methods that can be overridden in ActionCallback subclasses to handle action responses that result from submitting configurations to other Controllers. Applications submit configurations to Controllers using one of the **`submit`** methods defined in the **`IController`** interface after connecting to the controller.

If the application *does not care* whether the resulting action completes successfully or not (e.g. the application is part of a processing pipeline), it may use the **`submit(IConfiguration config)`** method. Otherwise, one of the submit methods that allows attaching an doAction callback_ should be used.

All action callbacks must subclass **`atst.cs.controller.ActionCallback`**, which provides the core behavior required of action callbacks by the technical architecture. Two methods are available for overriding to add functional behavior.

5.2.7.8.5.1 doDone

The **`doDone`** method is called internally by the **`done`** method when the target controller reports successful completion of the action. While it possible that the functional behavior needed here does not need access to information in the surrounding Controller (or simple Component, potentially!) in most cases some type of access to the Controller will be needed (to set synchronization flags, for example). The methods **`getController()`** and **`getComponent()`**, described below, provide this access.

5.2.7.8.5.2 doAbort

The **`doAbort`** method is called internally by the **`abort`** method when the target controller reports unsuccessful completion of the action. While it possible that the functional behavior needed here does not need access to information in the surrounding Controller (or simple Component, potentially!) in most cases some type of access to the Controller will be needed (to set synchronization flags, for example). The method **`getController()`**, described below provides this acess.

5.2.7.8.5.3 doReport

The **`doReport`** method is called internally by the **`report`** method whenever a controller wants to issue a *progress report* on the processing of an action. Its use is not required.

5.2.7.8.5.4 getController()

The convenience method **`atst.cs.controller.Controller getController()`** provides access to the Controller that issued the submit for the action being reported by this ActionCallback. The result will likely need to be upcast to the appropriate Controller subclass to call getters and setters to access information that needs to be shared between that controller and this callback. For example, the **`doDone`** method could be written as:

```
protected void doDone(IConfiguration config) {
        ((MyController)getController().signalDone(config.getId());
}
```

to signal successful completion of this action to the controller. Here, the **`signalDone`** method has been added as a public method to the Controller subclass **`MyController`**.

5.2.7.8.5.5 getComponent()

The convenience method **`atst.cs.component.Component getComponent()`** provides access to the Component that issued the submit for the action being reported by this ActionCallback. The result will likely need to be upcast to the appropriate Controller subclass to call getters and setters to access information that needs to be shared between that component and this callback. For example, the **`doDone`** method could be written as:

```
protected void doDone(IConfiguration config) {
        ((MyComponent)getComponent().signalDone(config.getId());
}
```

to signal successful completion of this action to the component. Here, the **`signalDone`** method has been added as a public method to the Controller subclass **`MyComponent`**.

This method ***must not be used*** if the submitter is a Controller subclass. Use **`getController()`** (above) instead.

5.2.8 C++-based Controllers

TBD

5.2.9 Python-based Controllers

TBD

-- BretGoodrich - 09 Mar 2005

to top

End of topic
Skip to action links | Back to top

---

to top