Keck Adaptive Optics Note 679

NGAO Control Software Architecture

Last Revision Date
8/10/09

Erik Johansson, Jimmy Johnson, Doug Morrison

Version 1.0, August 10, 2009

Revision Table

Version	Date	Authors	Comments
1.0	August 10, 2009	EJ, JJ, DM	Initial Release

Summary Table of Contents

Table of Contents

Table of Figures

1 Introduction

This document describes the design of the software architecture for the NGAO control system. It is not a description of the control system design, but rather the software architecture that will be used as the foundation for designing and implementing the control system. There are several target audiences for this document: people who are not software engineers who need a basic understanding of the software architecture, software developers who will be implementing the NGAO control system using the architecture, AO scientists who will be implementing AO applications using the NGAO control system, and finally, software developers who will be maintaining and extending the architecture itself.

The document is written using the concept of “views” organized around these target audiences. We begin with some background and an overview that are appropriate for all the target audiences. Next, we present the *application developer view*, which is appropriate for the software developers and AO scientists implementing the control system and AO applications. We conclude with the *architecture developer view*, which is an overview of the software architecture appropriate for software developers maintaining and extending the architecture.

2 Related Documents

a. KAON 671: Keck NGAO Software Architecture: Container Component Model, J. Johnson, D. Morrison, E; Johansson, Rev. 2, June 3, 2009.
b. KAON 672: Keck NGAO Software Architecture: Connection Service, D. Morrison, J. Johnson, E. Johansson, June 3, 2009.
c. KAON 673: Keck NGAO Software Architecture: Logging Service, D. Morrison, J. Johnson, E. Johansson, June 30, 2009.
d. KAON 674: Keck NGAO Software Architecture: Sequencer Architecture Design, D. Morrison, J. Johnson, E. Johansson, July 31, 2009.
e. KAON 675: Keck NGAO Software Architecture: Tasks, D. Morrison, J. Johnson, D. Morrison, July 30, 2009.
f. KAON 676: Keck NGAO Software Architecture: Configuration Service, D. Morrison, J. Johnson, E. Johansson, July 27, 2009.
g. KAON 677: Keck NGAO Software Architecture: Alarm System, J. Johnson, D. Morrison, E. Johansson, June 23, 2009.
h. KAON 678: Keck NGAO Software Architecture: EPICS Interfacing, J. Johnson, D. Morrison, E. Johansson, June 9, 2009.
i. “ATST Common Services User’s Manual”, Steve Wampler, ATST Software Group, Panguitch-1P7, Feb 12, 2007.
j. “ATST Common Services Reference Guide”, Steve Wampler, ATST Software Group, Panguitch-1P7, Feb 12, 2007.
k. “Evaluation of Software Frameworks for the ASKAP Monitoring and Control System”, Guzman, J. C., ASKAP-SW-0002, Version 0.2, 11/21/08.
l. “Evaluation of the Data Distribution Service for Real-Time Systems (DDS) in the Context of the E-ELT TCS Infrastructure”, Taylor, P., ESO Document number E-TRE-OSL-439-0012, Issue 1, 2/9/09.

3 Document Conventions

When using code fragments as examples, we will use the non-proportionally spaced Courier New font:

```
class foobar extends foo {
   int foo1;
   double foo2;
}
```

We also use this font for specific code related terms embedded in the text (class names, method names, etc.).

The architecture is designed using modern software methods, which have their own unique vocabulary. Hence, many of the terms in the document may be unfamiliar to people. Rather than defining all of them as they are used in the text, which can obscure the overall meaning of a passage, we have chosen to describe some as necessary and to include others in a glossary of common terms at the end of the document.

4 Background

We approached the NGAO software architecture design with two primary objectives in mind. First and foremost, the architecture has to meet its main functional requirement of supporting the NGAO controls infrastructure so that the system can be successfully integrated into the Keck Observatory environment and support operational science observing. Secondly, and nearly as important, the software architecture should be designed using modern software methods and tools. This means embracing modern object oriented design methods, design patterns and new development environments. This second objective is somewhat of a departure for the Observatory. With the exception of MAGIQ, and, perhaps, portions of the Interferometer, most of the software development and software architectures employed by the observatory are based on an aging, and in some cases, nearly unsupportable infrastructure. Given the strategic importance of the NGAO system to Keck Observatory, we believe it is critical that the NGAO software architecture be a foundation and an example for future projects and be sustainable for many years to come.

Our current software and controls infrastructure relies on the EPICS control system middleware, Sun workstations and VxWorks real-time CPUs in VME crates. In some areas, our versions of EPICS are so old they are no longer supported by the EPICS community. Most of our versions of Solaris are old as well and not widely used in the broader computing community. Finally, our version of VxWorks is quite old, and in some cases the host CPU hardware is no longer supportable. Moreover, while EPICS works quite well in an environment where the process variables are scalar, it is not well suited to process variables that are arrays or structures, which are used quite heavily in adaptive optics.

As a result, we began the architecture design process by looking at a number of communication middleware packages and software frameworks as alternatives to EPICS. Some items we examined directly, but for others we relied on analysis work done by other groups. One useful input was the “Evaluation of Software Frameworks for the ASKAP Monitoring and Control System” produced by Juan Carlos Guzman (ASKAP is the Australian Square Kilometre Array Pathfinder, a next generation radio telescope being built in Western Australia). Other useful input came from Gianluca Chiozzi of ESO who provided “Evaluation of the Data Distribution Service for Real-Time Systems (DDS) in the Context of the E-ELT TCS Infrastructure” and other documents. As a side note, this cooperation between observatories has proven to be quite fruitful. We have received quite a lot of supporting material and software from others and we have shared some of our results and progress as well.

During the course of our initial investigation we explored the frameworks RTC, TANGO, EPICS, TINE and ACS before deciding to invest more time on the Advanced Technology Solar Telescope project (ATST) Common Services Framework (CSF). Bret Goodrich and Steve Wampler, from NOAO, have been a source of tremendous support, providing code and documentation, as well as hosting a Keck visit to their facility. CSF is a control system framework which uses a distributed component-based development environment. It uses a container-component architecture which is designed to be independent of the communication middleware that is used. The ATST project evaluated several middleware solutions for the CSF: CORBA, ICE and NDDS. They decided on ICE, a CORBA-like product from ZeroC Inc, due to its good performance (over CORBA) and the fact that it has both commercial and GPL licensing options. We have some small experience with ICE from the MOSFIRE project.

In addition to being middleware neutral, another important aspect of CSF is its use of narrow interfaces (these can be seen in many modern solutions such as TANGO, and similar approaches are being discussed for ELT). There are a small number of standard objects defined: components are defined as controllers or devices, with their interfaces defined in a standard IDL file. All data transfers (apart from bulk data) are transmitted using a concept called attributes. This allows for very loose coupling and the ability to add or remove functionality without requiring major code rewrites and dealing with API interdependencies. CSF supports peer-to-peer communication in addition to publish-subscribe capabilities.

While our approach differs from some of the philosophies of CSF in terms of how data is internally represented and in some of the component and controller interfaces and interactions, the majority of CSF is directly applicable to us. With some minor changes to a few interfaces, some data structures, base components and controllers, we have been able to implement a proof of concept for our framework. We have also been able to use a number of the CSF common services, such as logging and its associated UI, that go beyond the core infrastructure. In addition to evaluating the ICE middleware we have also been successful in prototyping CSF over Data Distribution Service (DDS) which is a good testament to its architecture. We recently shared the code we have implemented for channel access (CA) service, and DDS based connection and event services with NOAO/ATST. We are currently iterating on some design ideas for the alarm system and will have a visit in October from Steve Wampler to further discuss the frameworks. As we move forward in our design, there will, no doubt, be other good opportunities for collaboration and both Keck and ATST will continue to benefit in kind.

5 Overview

The software architecture we are proposing for the NGAO system is a framework based on the ATST CSF described above. We have dubbed the architecture the "Keck Common Services Framework", or KCSF. The KCSF provides the base infrastructure needed to develop a distributed control system. In a distributed system, components will need to communicate with each other, perform certain actions, and work together in a pre-defined way to form a larger system. The framework enables this by providing common communications, common data types, common commands and common tools throughout. The solution uses a component based development model that supports methods, attributes and properties. The framework provides a kernel API which hides all the details of network access and provides object browsing, discovery, common services and more.

It is important to keep in mind while reading this document that there are two distinct architectural views: the technical architecture and the functional architecture. The *technical architecture* is concerned with the component execution environment, associated tools, services, user interfaces and technical facilities required to develop and run a distributed component based system. The *functional architecture* is concerned with the functional aspects of the system, which are the specification and implementation of a control system that satisfies the functional requirements.

The framework implements the technical architecture in full and allows the developers to focus on their functional areas. This separation of concerns means that that once the framework is delivered, developers can limit themselves to the design and implementation of the required devices and controllers, without regard to any of the technical infrastructure. It also means that functionality can be added or changed later on without adversely affecting the rest of the system.

The basic functional component supported by the framework is a "device". Devices can represent physical hardware or more abstract concepts. The framework also supports a special case of a device, called a controller, which can perform simultaneous operations and coordinated activities. All device configuration data is stored in a database. The devices can be on the same computer or distributed over a number of computers interconnected by a network.

The framework is flexible in its choice of network protocols and we are currently evaluating the Internet Communication Engine (ICE) and Data Distribution Service (DDS) protocols. Both peer-to-peer and publish-subscribe communication models are supported. Communication between components can be synchronous or asynchronous, and on-demand or event driven.

Some ready to use graphical applications allow users to graphically display data coming from devices without the need for any programming. Graphical layers above the kernel API reduce specific graphical client software development time. Logging, alarm and archiving capabilities are available to keep track of what is happening in the system.

5.1 Key features

This section describes the key features of the framework. A high-level overview is shown in Figure 1 and the features are described briefly in the sub-sections that follow.

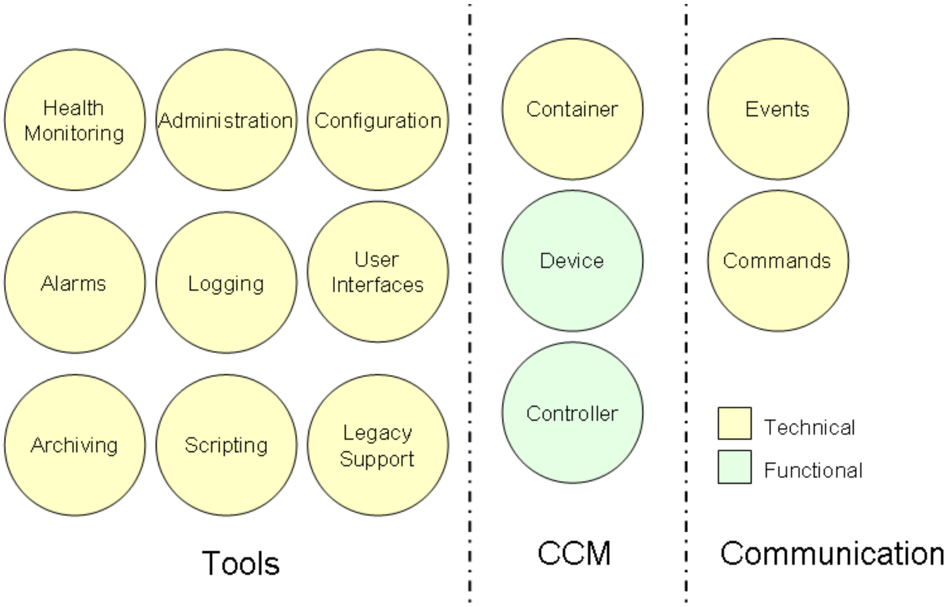


Figure 1: An overview of the Keck Component Framework.

As discussed above, the software infrastructure consists of a technical architecture and a functional architecture. The technical architecture consists of the following:

- A container component model (CCM) which includes:
    - An implementation of containers
    - An implementation of container managers
    - Lifecycle management of components
- A communication model and communication middleware
- A set of core tools, that includes
    - Command support
    - Events
    - Alarms
    - Logging
    - Configuration
    - Archiving
    - Legacy Support
    - User Interface
- Low level tools for generic database access and communications (middleware libraries, device drivers and protocol libraries).

The functional architecture consists of the following:

- A base implementation of devices and controllers
- Templates for common devices and controller

5.1.1 Distributed component based architecture

Software development and design have evolved significantly over the last decade. Distributed development is now quite common, where applications no longer exist in a single process space on a single node, but can be distributed over many machines throughout a network. Distributed development has been a very successful design method for large software projects; however, it still sufferers from many of the same weakness of previous models. Small modifications to code tend to impact many modules, and typically result in full or large-scale code recompilation and deployment. Additionally, distributed systems (as with the previous models) tend to strictly define object relationships directly through custom class interfaces and functionality. This typically makes upgrading a distributed application a difficult task since class dependencies are built into the code and may span the entire system.

With the desire to improve the maintainability and scalability of distributed systems, the Component-Based Development (CBD) model was created. This model focuses on removing dependencies between objects by designing components with strict and well defined interfaces. Components must conform to a known standard based on the problem each implementation is designed to solve. The goal is to be able to isolate the functional implementation of a task within a single plug-and-play object that can be modified or swapped with an alternative implementation without impacting the rest of the system. A user should be able to replace a component that satisfies a similar use-case without creating or breaking any dependencies in the system.

CBD provides a flexible and agile way to architect, design, implement and deploy scalable software systems. The goal of this model is to produce software that is interchangeable and easily replaceable by separating the functional and technical aspects of a system. In essence, the model drives developers to see their system as a breakdown of self-contained and swappable objects.

Component software consists of components, tools to manage them and a component runtime. Components are a physical replaceable part of a system that package the implementation of domain specific functionality in a modular way. The tools aid in managing the components and configuring them to the desired runtime environment. A component runtime defines a runtime environment for components.

Depending on the system needs, components can be deployed in different ways. They can be reassigned to balance resource usage and network load. They can be functionally partitioned to create independent islands that provide domain partitioning to support rapid fault recovery, easier management and monitoring and so on.

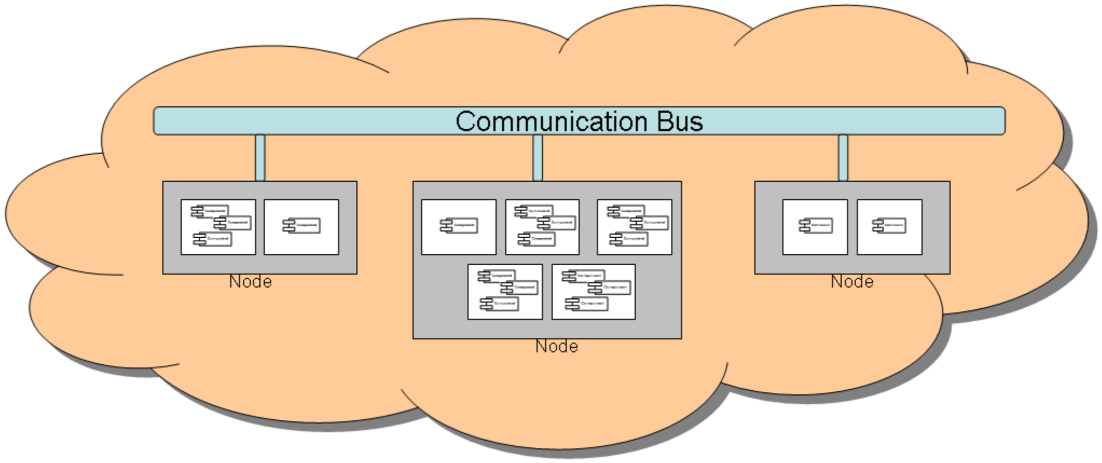


Figure 2: A distributed Container-Component Model view.

5.1.2 Commands

Commands are used to set and get device attributes and to initiate an action or sequence of actions. KCSF supports both synchronous and asynchronous commands. The basic commands are `get`, `set`, and `execute`. `get` and `set` are used to read and write attributes to a component, while `execute` is used to initiate a command requiring action. Commands follow a command-response pattern for synchronous operations and a command-action-response pattern for asynchronous operations. Components respond to commands in one of three ways:

1. No response is needed. Some systems and configurations operate in a stateless manner. For example, the logging system simply accepts messages and records them, no response is required. This is implemented as a synchronous command.
2. An immediate action is performed and acknowledged. If the action can be performed essentially instantaneously, the command response provides the result of performing that action. This is a synchronous command.
3. The command is acknowledged as valid and an action is initiated. When the action completes a separate mechanism (callback or event) announces the completion. This is an asynchronous command.

As actions are being performed by a component in response to a command, the component may receive and generate events, make requests of the persistent stores, generate alarms and record log messages.

5.1.3 Monitoring

In addition to the `get`, `set`, and `execute` methods discussed above, KCSF provides the ability to monitor the attributes of any component for changes. A monitor subscription is established using the component's `addMonitor` method, and the component executes a call back when any of the requested attributes have changed in value. The rate at which the monitor operates is specified by the user. There are several other utility functions for monitors which are described in detail in the User View and Developer View sections of this document.

5.1.4 Events

Events are used for asynchronous message delivery. The event system in KCSF is modeled after conventional notification services. Events are a publish/subscribe mechanism that is based on event names: Components may post events without regard to the recipients and subscribe to events without regard to their source. Subscriptions may be made using names containing wild-card characters to subscribe to multiple events sharing common name characteristics.

5.1.5 Alarms

Alarms are propagated using the event system. Components may generate alarms, clear alarms, respond to alarms, and recast alarms.

5.1.6 Log messages

Components may generate messages that are recorded into a persistent store.

5.1.7 Configuration

It is expected that most devices and controllers will have some set of configurable properties associated with each instance. When objects are instantiated, these properties and associated attributes may have hardcoded default values that will need to be overridden for the object before standard operations can begin. A convenient way to store and retrieve object properties is through a Configuration Database. This database will define all of the configuration information required by each class instance, as well as any additional metadata and run-time information needed by the system.

When a component is loaded, the framework will automatically retrieve the configuration for it and will update the associated component attributes automatically. The framework will also provide the capability to allow all components to retrieve their configuration on demand.

It is also possible to create device type configurations, which apply to all devices of a particular type, not just to a particular device instance (e.g., all LGS wavefront sensors, as opposed to LGS point-and-shoot WFS number 1). When a device is configured, the configuration for the device type will be loaded first, followed by the configuration for the particular device instance. This way, default parameters that apply to all devices of a particular type can be loaded during the device type configuration, with the capability to override them using the device instance configuration.

5.1.8 Archiving

Data archiving refers to the long-term storage of data. The framework allows component attributes to be archived at high rates to a relational database, where they can be later queried. Both a push and pull model are supported. Components can choose to archive data on demand, though this may not be a common case. In addition a data archiver can be configured to pull data from components at a dedicated periodic rate or on the occurrence of a particular event.

5.1.9 Legacy Support

In order to support backwards compatibility with existing instruments and systems, the framework will provide full channel access (CA) support such that it can act as both a CA server and a CA client.

5.1.10 User Interface

User interfaces are a key part of the KCSF framework. The design will define the overall look and feel, the preferred technical implementation (UI toolkit), and development guidelines: use of the Model/View/Controller (MVC) or Model/View/Presenter (MVP) model, helper functions and base widgets. The following user interfaces will be provided as part of the base framework:

- **Configuration UIs –** user interfaces to configure the system
- **System Administration UIs-** user interfaces to help deploy and monitor the running system
- **Alarm Handler –** user interface to monitor and acknowledge alarms from the system
- **Log Viewer –** examine contents of the log database with filtering capabilities, such as source, time ranges and message types etc.
- **Archive Viewer –** examine contents of the archive database with filtering capabilities, such as source and time ranges etc.

- **KCSF Panel** - a generic UI that can be used to display the values of attributes or to execute commands on any set of devices. Values can be displayed numerically or graphically.

5.1.11 Scripting

The framework provides a scripting engine to execute scripts that can interact with the framework. A number of different scripting languages can be supported. Scripts have full access to the framework services and tools including the connection service allowing scripts to communicate with any KCSF components. Sequences can be extended to use scripts and scripts can be submitted and executed directly through standalone KCSF applications.

5.2 Architecture Layers

The framework is a layered architecture. This is a very common design pattern used when developing systems that are composed of a large number of components across multiple levels of abstraction. Components are separated into layers. The components in each layer are cohesive and at roughly the same level of abstraction. Each layer is loosely coupled to the layers underneath.

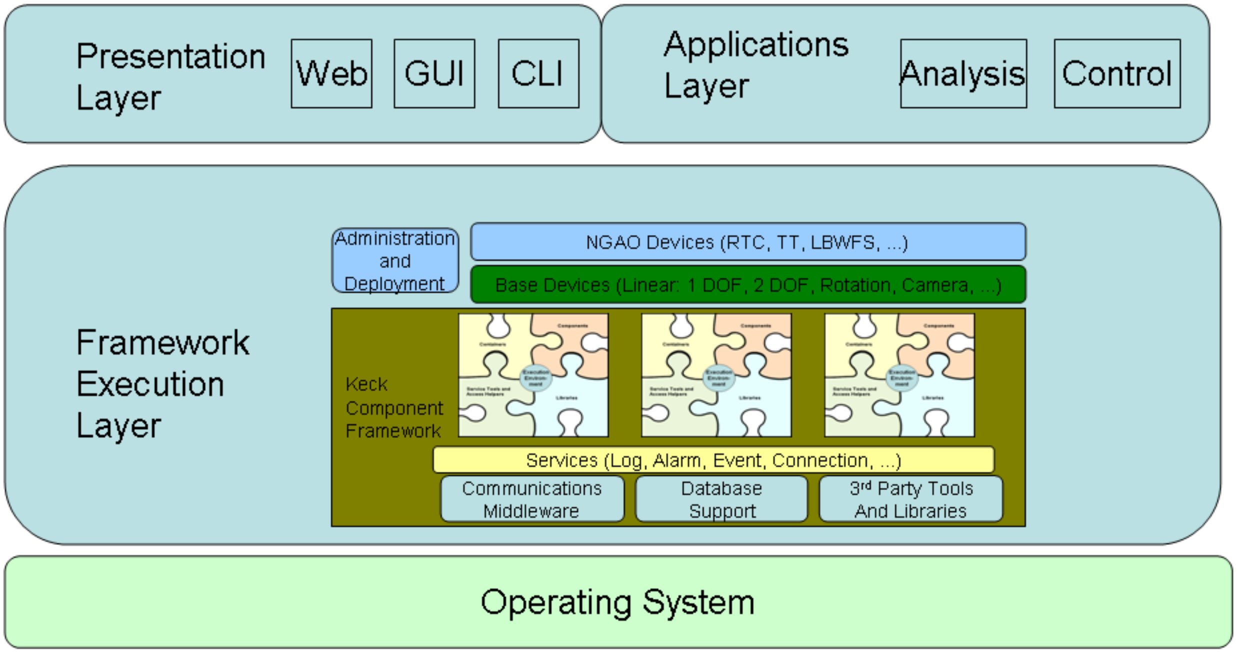

Presentation Layer
Web
GUI
CLI
Applications Layer
Analysis
Control
Framework Execution Layer
Administration and Deployment
NGAO Devices (RTC, TT, LBWFS, ...)
Base Devices (Linear: 1 DOF, 2 DOF, Rotation, Camera, ...)
Keck Component Framework
Containers
Components
Execution Environment
Service Tools and Access Helpers
Libraries
Services (Log, Alarm, Event, Connection, ...)
Communications Middleware
Database Support
3rd Party Tools And Libraries
Operating System

Figure 3: Architecture Layers Overview

The design pattern addresses the need to support operational requirements such as maintainability, reusability, scalability, and robustness. By using the layered application pattern the container-component based development benefits can be further enhanced so that

- Localized changes to one part of the solution minimize the impact on other parts, reduce the work involved in debugging and fixing bugs, ease application maintenance, and enhance overall application flexibility.
- There is an additional separation of concerns among components increasing flexibility, maintainability, and scalability.

- Independent teams are able to work on parts of the solution with minimal dependencies on other teams.
- Various components of the solution can be independently deployed, maintained, and updated, on different time schedules.

5.3 Container Component Model

Facilitating component based development is the Container Component Model (CCM), a key architectural pattern that provides for explicit descriptions of provided and requested services and a separation of concerns between the functional (business logic) and the technical architecture. The pattern leads to easier development, deployment and reuse of software modules. Object Management Group (OMG) standards exist for a Common Object Requesting Broker Architecture (CORBA) Component Model and a newer solution called a Lightweight CORBA Component Model (LWCCM). There is also a proposed standard on using LWCCM with Data Distribution Service (DDS), a publish-subscribe middleware of interest to us. Our proposed use of CCM is based on the architectural pattern and not any of the specific CORBA based solutions mentioned above.

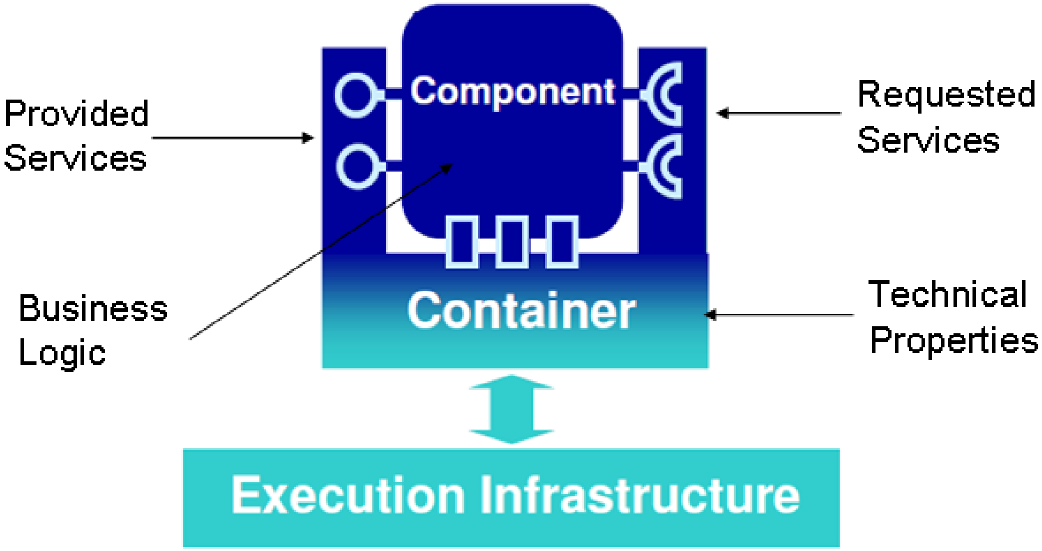


Figure 4: Container Component Model

The CCM defines a distinct separation between the technical and functional requirements of a task. The model is based on the utilization of two distinct software modules: *containers* and *components*. Containers provide a structural or logical organization to software objects (components), and are responsible for their lifecycle and management. This is illustrated in Figure 4.

A ***container*** will create components, start them running, shut them down, and remove them from the system. Containers provide a uniform method of system management and allow component developers to ignore the majority of the technical requirements of individual objects. Containers, component interfaces and abstract implementations are provided as part of the software infrastructure.

A ***component*** is a piece of software that implements the functional requirements of the system and conforms to a well defined interface or set of interfaces. The interface abstracts away the actual implementation of a component, and from the perspective of the system, there is no difference between one implementation and another. In this way, a developer can swap out multiple implementations of a component for a specific use case without affecting the system as a whole. This is most evident when you compare the maintenance and upgrade of a component-based and a non-component-based system.

In order to create a complete application solution, components must be able to communicate with each other, which is provided through the communications infrastructure.

5.4 Communications

The KCSF is a *distributable* system. As a result, the communications infrastructure must be an integral part of the KCSF. Much of the implementation of the communications infrastructure is based upon services and support features found in third-party communications middleware packages. However, the framework isolates the dependence on third-party middleware from the rest of the KCSF software system so that replacing the middleware is always a viable option. We have successfully prototyped three different communication middleware packages (ICE and two versions of DDS). In addition, different middleware can be transparently adapted for different services if they prove to be better in one particular area (i.e., middleware can be mixed and matched as needed). The connection service that encapsulates the chosen middleware essentially forms a software bus to which components can connect. This is illustrated in Figure 5, which shows many components attached to an abstract software bus. The components all use a simple abstract interface (the software bus) to communicate with each other and are oblivious to the underlying communications mechanism.

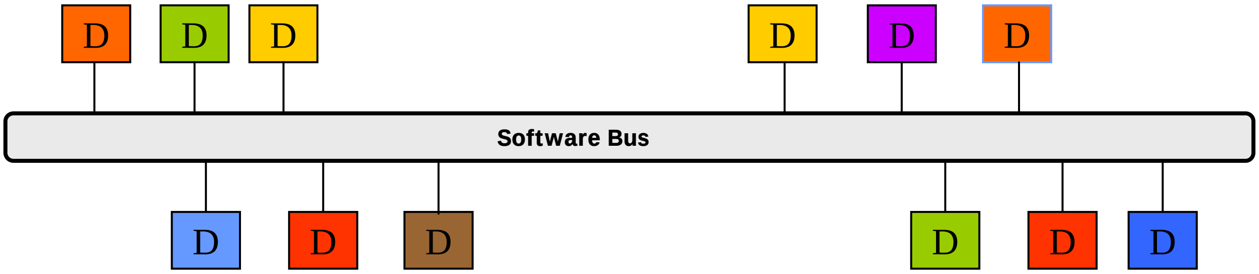


Figure 5: Multiple devices/controllers communicate with each other using a simple abstract interface.

The framework takes care of the following:

- Automatically registering all components when they are created so that they become known on the bus.
- Automatically un-registering all components when they are destroyed so that they are removed from the bus.
- Allows any component to be referenced by its fully qualified name, regardless of its location.

- Allows operations to be performed on a remote component as though it were local to the caller.
- Transparently handles network problems such as packet drops etc.

Moreover, the communications infrastructure must be capable of high throughput, varied data size and diverse message types, using different messaging paradigms. KCSF supports this by allowing different communication middleware to be used without affecting the application developer and by using a middleware that supports:

- Peer-to-peer *command* messaging, allowing arbitrarily complex messages to be sent directly from one component to another.
- Publish-subscribe *event* messaging, allowing the generation and reception of messages by components without regard to the intended message recipients or sources.
- 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 is not running).
- Heartbeat monitoring. Applications are watched and an alarm is raised if an application unexpectedly stops responding.
- Distributed systems. Distributed systems are easily supported, allowing seamless integration of third party modules.

5.5 Benefits

By combining the communication middleware with component-based development, services and a technical architecture, a number of benefits can be realized.

System details are hidden from the developer and user. This allows application developers to concentrate on applications, not the underlying infrastructure and allows system developers to make infrastructure changes with out affecting deployed applications.

By using a narrow interface, components can be updated as needed without having to worry about version mismatch on interfaces or methods. The KCSF API is very simple and is designed such that all third party communication middleware is completely isolated from the API. All major services are provided through standard interfaces promoting simplicity and consistency. Dependencies between objects are eliminated, resulting in improved ease of maintenance. As a result components may be swapped out with alternate implementations without impacting the rest of the system. Users can replace a component with another that satisfies a similar use-case without creating or breaking any dependencies in the system. The component model makes it very easy to use components from different development groups in an application.

The technical architecture takes care of deploying components. Components can easily be mixed and matched and even be added or removed during run time, as can services, effectively changing the system function on the fly. The implementation of the CCM provided by KCSF allows components to be installed, uninstalled, started, stopped or updated without bringing the entire system down. For deployment, the system simply needs to be configured by the engineer to tell it which components to deploy in which containers and on what machines.

Once a component is deployed, the API provides access to its internal state as well as how it is connected to other entities. Parts of the applications can be stopped to debug a particular problem, or diagnostic components can be brought in. Instead of staring at hundreds of lines of logging output and enduring long reboot times, applications can often be quickly debugged using a live command shell. Any entity can be tracked to see its current health and if it is registered.

5.6 Summary

The KCSF architecture provides a component-based development environment using a container-component model, coupled with tools and services in a robust framework with a clean separation between the technical and functional implementations. This allows the application developer to focus on the development of the control system, (the component devices, control system applications scripts and tasks, user interfaces, and how they should communicate with each other) rather than on the underlying framework. The architecture is scalable, distributable and maintainable. These concepts are shown below in Figure 6.

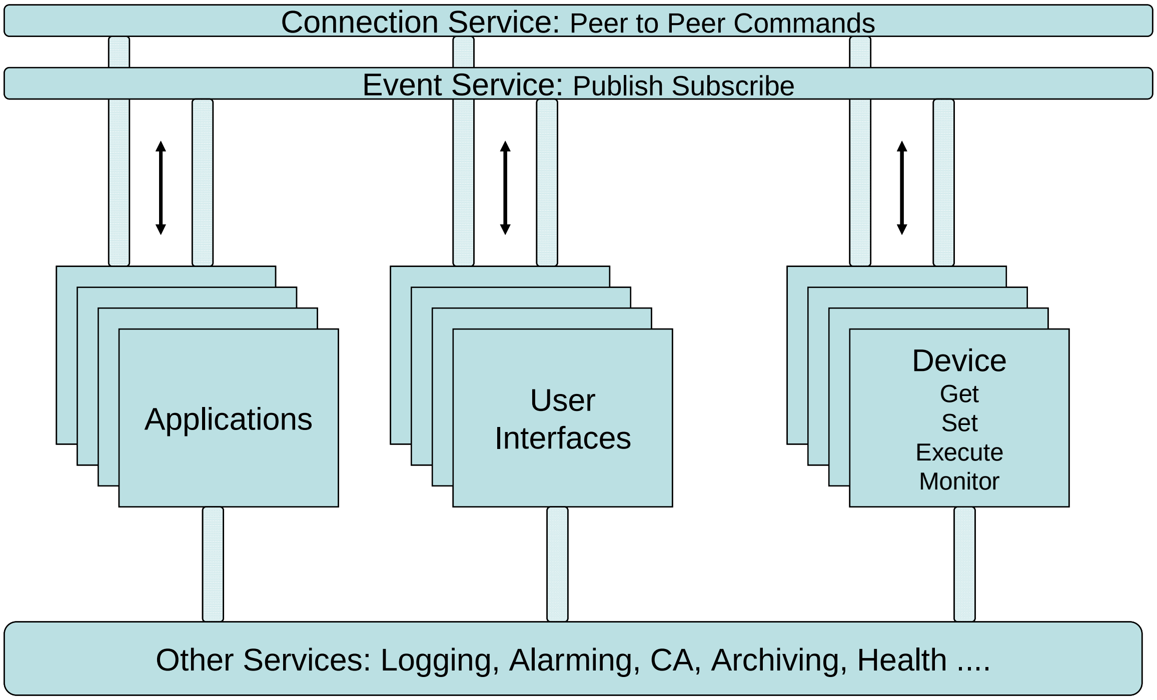

Connection Service: Peer to Peer Commands
Event Service: Publish Subscribe
Applications
User Interfaces
Device
Get
Set
Execute
Monitor
Other Services: Logging, Alarming, CA, Archiving, Health ....

Figure 6: A summary of the KCSF architecture.

6 Application Developer View

This section contains information needed by an application developer to write applications that use services and communicate with components, and to develop new components. While details of the technical architecture are not required, having a basic understanding of the overall architecture is useful.

6.1 Container Component Model

The following is part of the technical architecture and not something that an application developer needs to have any detailed knowledge about. However, an application developer should be aware of where the developed devices reside within the architecture, how to create a device, how device lifecycles are managed and what services are available to devices.

The CCM includes a number of key entities:

- Container Manager: A component responsible for managing containers. It is capable of deploying containers on different nodes, attaching to already running containers, stopping and removing containers. A container manager determines the appropriate set of policies and service bindings to be used by its containers, and then acts as a factory to create them.
- Container Server: A container server is an execution environment in which one or more containers are created during runtime.
- Container: Manages components and provides services and connections.
- Component: A standalone reusable functional entity.
- Services: Services are support functions typically shared by all components in a container. Examples include events, logging, alarming etc.

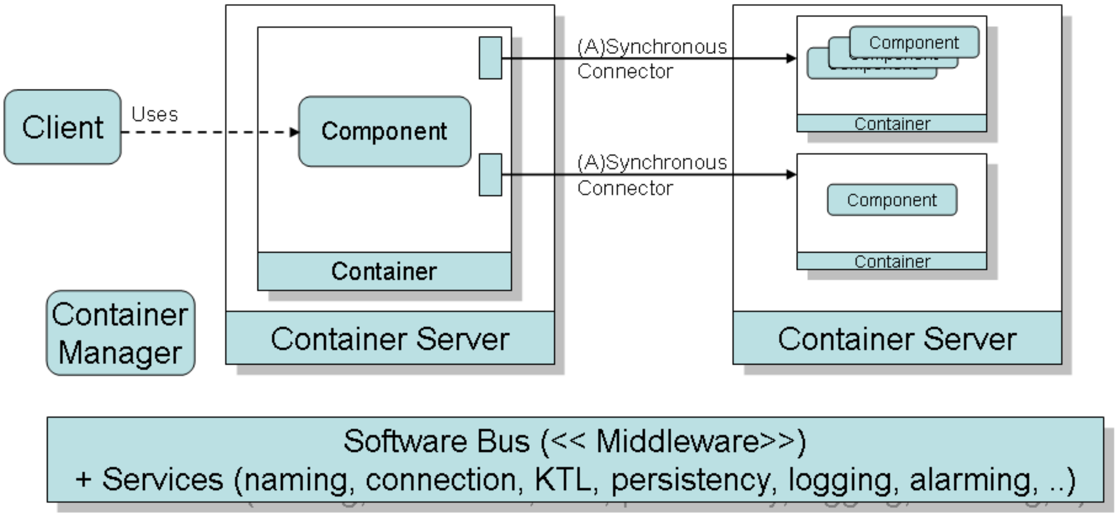


Figure 7: General model for containers.

Figure 7 shows a general model for containers. A container manager deploys containers on one or more container servers. The containers are then used to deploy one or more components each.

A client interacts with a component by sending requests to the component via the software bus (note: if components are on the same container server (compute node) the communication is via IPC not RPC). Just as containers provide the creation and life-cycle management of the system (the technical architecture), components are designed to implement the functional requirements of the system (the functional architecture). All components are derived from a base class that provides a consistent interface to the parent containers. In addition to any custom implementation of the base methods (creation, startup, shutdown, etc.), a component provides the specific functionality for the task it is designed to solve. From the perspective of a container, everything is a component – components can be swapped out without requiring explicit changes to the container's implementation. This ability to swap component implementations is what makes the CCM pattern such an effective design technique.

In KCSF, there are two basic types of components: devices and controllers. These are discussed in further detail below.

6.1.1 Component Lifecycle

The lifecycle of a component consists of the following basic states: creation, initialization, startup, operation, shutdown and removal. The technical aspects of component lifecycle management are handled by the container and the component base class methods. There are opportunities for the component designer to provide functional aspects of the lifecycle management as well (this is discussed in detail in Section 6.11.1). The lifecycle states are described briefly below:

- **Creation**. The container constructs the component and creates the required services. The services are not available to the component until the initialization state.
- **Initialization**. The component calls the configuration service to get its configuration data and applies it to the designated attributes. The component then performs local initialization by creating any required buffers and connecting to device drivers.
- **Startup**. The container registers its component with the connection service. The component performs any local startup tasks.
- **Operation**. The main functional state for a component. In this state the component performs all of its control functions, calls any required services as part of its operation, and periodically performs health monitoring checks on itself.
- **Shutdown**. The component releases all external resources used. The container unregisters the component from the connection service and it can no longer be seen by other components in the system.
- **Remove**. The component ceases all activities and releases its internal resources. The component no longer exists.

The lifecycle is illustrated in Figure 8 below. A component will remain running in the operation mode (shown in blue) until specifically requested to stop or re-initialize.

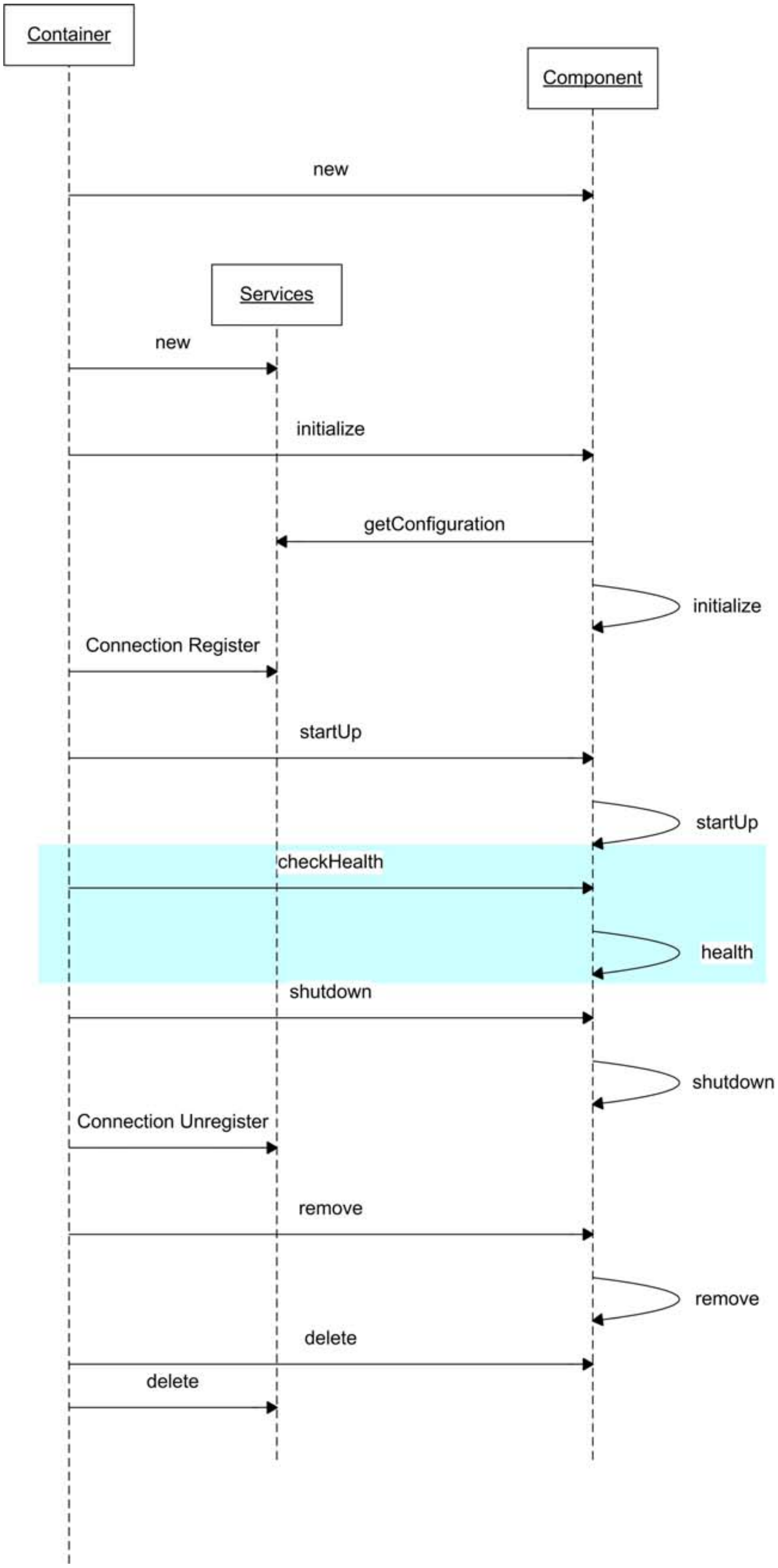


Figure 8: Component lifecycle sequence diagram

6.2 Data Transfer Objects

A data transfer object (DTO) is a design pattern used to transfer data between software application subsystems. A DTO does not have any behavior except for storage and retrieval of its own data. All functional commands in KCSF utilize the same DTO, which is called an `Attribute`. An attribute is essentially a (name, value) pair. A collection of attributes is called an `AttributeList`. The `AttributeList` is used for device method calls (these are described in further detail in Section 6.3). The interfaces are represented as `IAttribute` and `IAttributeList` and the class representation is through `Attribute` and `AttributeList`.

6.2.1 Attributes

The `Attribute` class represents the common data structure used for device method calls. Attributes can be grouped together into an attribute list to allow for batching or atomic operations on devices. There is no limit to the number of attributes in a list. There can be only one instance of an attribute with a given name within an attribute list. Inserting a new attribute with the same name replaces the old one. The attribute list allows for easy and fast retrieval of Attributes and provides wrapper calls to get and set attribute data.

An attribute is a (name, value) pair, where the value field is essentially a union. The following scalar data types can be supported:

- Boolean
- Byte
- Integer
- Long
- Float
- Double
- String

In addition one- or two-dimensional arrays are allowed for each supported data type.

The `Attribute` class provides convenience methods to allow polymorphic setting and retrieval of the underlying data. For example, the value might be set as a double but can be retrieved as a string, int or long etc. or visa versa.

All attributes used in inter-application communication must be uniquely named. The way this is supported is by use of the fully qualified name, which makes use of the hierarchical structure of the framework. For example: `ngao.ao.vibrationsensor.pos` is the name for the vibration sensor multi-axis positioning stage.

Any attribute can also have configuration meta-data associated with it. This might include information such as the attribute’s default value, min and max values, alarm limits, and is represented by a set of reserved keyword names.

The API for the using attributes is quite simple:

- `String getName();`
    - Return the attribute's name.

- **void** setName(String name);
    - Set the name of the attribute.
- **void** setValue(String[] value);
    - Set the value of the attribute.
- String[] getStringValue();
    - Return the value of the attribute as an array of strings.

In addition, the `IAttribute` interface defines a number of convenience methods:

- `<type> get<Type>();`
    - Return the first element of the attribute's value as a `<type>`.
- `<type>[] get<Type>Array();`
    - Return the attribute's value as an array of `<type>`.

where `<type>` can be `boolean`, `int`, `long`, `double`, `float` or `String`. Examples:

- **boolean** getBoolean();
    - Returns the first element of the attribute’s value as a Boolean.
- **long**[] getLongArray()
    - Returns the attribute’s value as an array of Longs.

6.2.1.1 Similarity to Keywords

In Keck nomenclature an attribute is very similar to a keyword in that is a named value that can be read from, written to or monitored. Unlike a keyword, which is a floating concept where the mapping of the keyword to a physical entity is done via CAKE or a similar mechanism, an attribute is strongly associated with a component. An attribute name can appear many times in the system and the same attribute name can be used within different components, but to be externally referenced outside of a component, the fully qualified name must be used. The use of the fully qualified attribute name is closer to how a keyword would operate.

An attribute list provides for a more object oriented approach allowing attributes to be treated as grouped entities. While it is up to the component as to how it handles its attributes, this grouping allows for atomic operations across attributes as well as the capability to provide transactional support and command-response patterns.

6.2.2 Attribute Lists

Attributes can be grouped into a set of attributes, called an `AttributeList`, which can be searched efficiently. As mentioned previously, at most one instance of an attribute with a given name may exist within an attribute list. Inserting a new attribute with the same name replaces the old one.

The following methods are provided:

- **boolean** contains(String attributeName);
    - Does the list include a specified attribute?
- **void** insert(IAttribute attribute);
    - Insert an attribute into the list.
- IAttribute remove(String attributeName);
    - Remove an attribute from the list and return it.

- `IAttribute get(String attributeName);`
    - Return an attribute from the list.

Additional methods are provided for convenience, including:

- `int size();`
    - Return the number of attributes in list.
- `String[] getNames();`
    - Return the names of all attributes in the list.
- `IAttributeList extractOnPrefix(String prefix);`
    - Return all attributes with names sharing a specified prefix.
- `IAttributeList extractOnSuffix(String suffix);`
    - Return all attributes with names sharing a specified suffix.
- `void merge(IAttributeList aList);`
    - Insert all attributes from another list into this list.

Moreover, the convenience methods defined for `IAttribute` are extended to work with an attribute list:

- `<type> get<Type>(String aName);`
    - Returns the value of the first element of the attribute `aName` from the attribute list as a `<type>`.
- `<type>[] get<Type>Array(String aName);`
    - Returns the value of the of the attribute `aName` from the attribute list as an array of `<type>`.

6.3 Devices and Controllers

Every user component in the framework is a device. A device represents an entity to be controlled. A device can be physical or logical, representing hardware or software. It can, for example, represent:

- A piece of equipment (e.g. a camera)
- A collection of equipment (e.g. a motor and encoder)
- An aggregate of devices (e.g. TTFA/TWFS)
- An application (e.g. Multi-command sequencer)

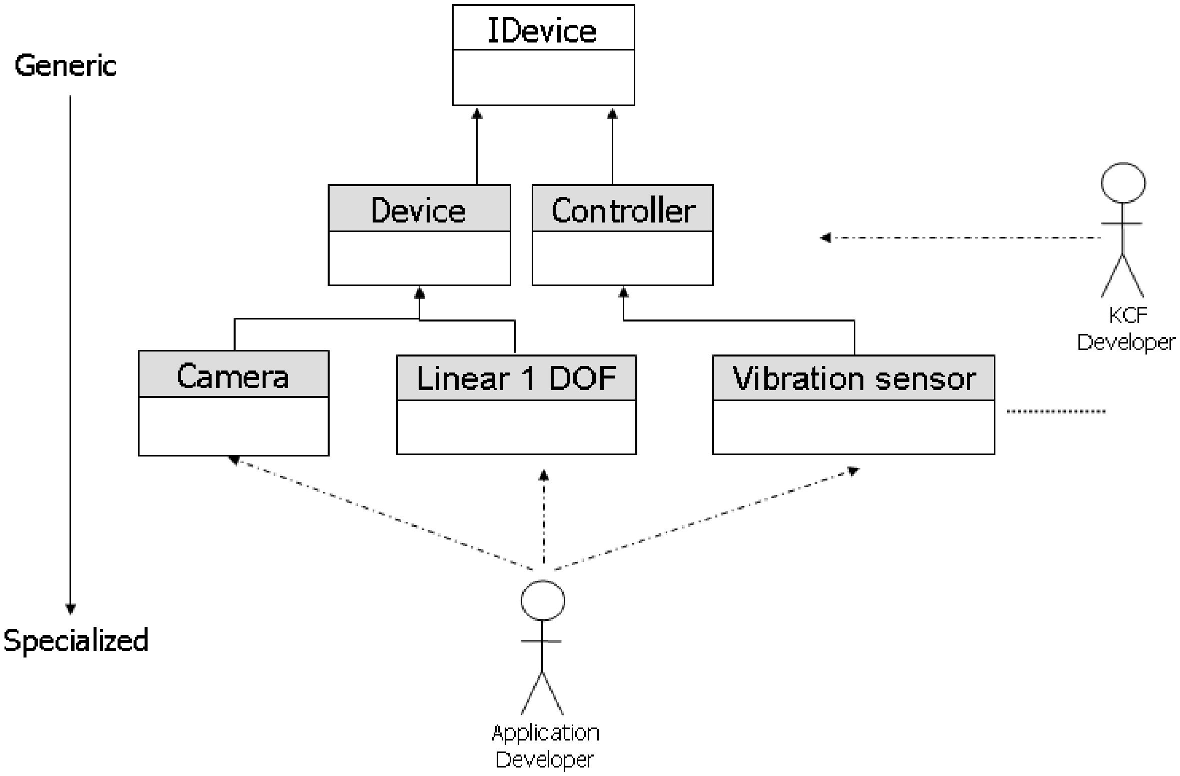


Figure 9: Devices and Controllers.

As shown in Figure 9, KCSF provides generic implementations of a device and controller. The generic implementations provide lifecycle support, helper support such as configuration and attribute validation, as well as delegation support to simplify the work of the application developer.

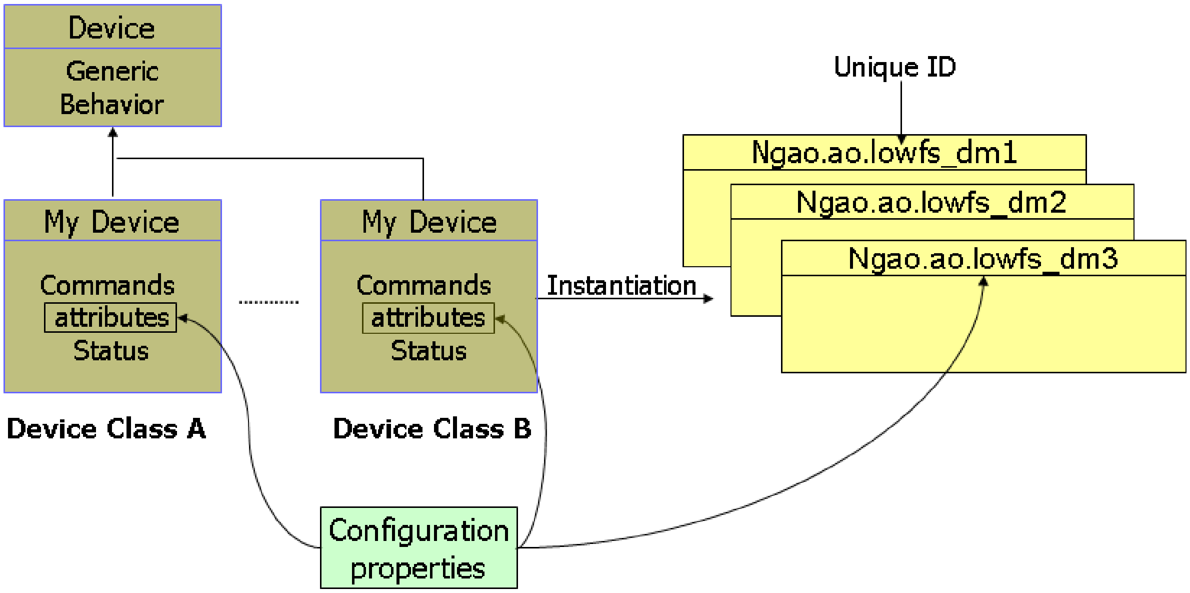


Figure 10: Device types and configuration

When devices are developed, common base classes can be used to encapsulate common characteristics to represent a family of devices. For example, the user can design a camera device which can be extended to represent several specific types of cameras (WFS cameras, diagnostic cameras, etc.). Furthermore, multiple devices may be combined using composition to create composite devices (e.g., a 3-axis motion controller can be composed from 3 single axis controllers). The extension of a base device class to develop a new device class is shown in Figure 10. Device classes A and B are derived from the base Device class. When actual devices are instantiated from these classes (shown in yellow), they are given unique IDs called fully qualified names. A fully qualified name is all that is needed to identify a device anywhere in the entire NGAO system. When devices are instantiated, they use the configuration service to load the proper default attribute values for the device. Some default attributes are retrieved based on the device type (e.g., default attributes for Device Class A), whereas others are retrieved based on their fully qualified name ("<system>.<subsystem>.<device name>.<attribute name>"). This is indicated by the arrows in Figure 10.

A device supports the `IDevice` interface and the `get`, `set`, `execute` and `xxxMonitor` methods. A controller is a specific type of device and has an identical interface. The difference between a device and a controller is in how the `execute` methods are processed. For a device, the `execute` methods are processed on a single server thread, while `execute` methods on a controller are on separate threads allowing multiple simultaneous commands and the ability to pause, resume or cancel ongoing commands. Hence, controllers may be used to execute multiple simultaneous commands. In the remainder of this section, both devices and controllers will be referred to simply as devices.

6.3.1 Functional Interfaces

The functional interface to a device is via the `get`, `set`, `execute` and `xxxMonitor` methods. The `get` and `set` methods are implemented in both synchronous and asynchronous forms. `execute` is inherently asynchronous. Synchronous calls will block until the operation has completed. Asynchronous calls will return as soon as the request has been sent on the wire and a boolean value will indicate whether or not the command was successfully sent to the remote component. Actual notification of command completion will occur sometime later via an optional callback. We have developed a standard callback interface for these methods: `ICommandCallback`. The APIs for these methods and the `ICommandCallback` interface are described in detail in the following sections.

6.3.1.1 Get

The `get` method provides a read interface for devices. Clients are able to request the current value of one or more device attributes through an attribute request list. Any attributes defined in the request list which are not supported by the component are ignored. The attributes read from the device are returned to the user in the form of an attribute list in one of two ways:

- The synchronous method returns the attribute list directly.
- The asynchronous method returns a boolean value indicating whether the request was successfully sent to the remote component. Assuming success, the attribute list is returned in the callback object.

A client is able to obtain a full list of all of the supported attributes by requesting the '*_SupportedAttributes*' attribute. The API for the `get` method is:

```
AttributeList get(AttributeList requestList);
boolean get(AttributeList requestList, ICommandCallback callBack);
```

There are two possibilities when reading from a device using the `get` method: returning a cached value from the device, or performing a read from the physical or virtual entity represented by the device. A read from the entity forces an update to the device cache. It is possible to select the type of read on an attribute by attribute basis by using the enumerations `CACHED` or `DEVICE` in the value field for a particular attribute. For most reads, the value field for each attribute will be null, as they have no meaning in the context of a read. The default behavior for a null is left up to the device designer.

The get method can also be used to retrieve the metadata values for a particular attribute. Metadata is read-only information that describes the attribute. To specify a metadata member use the '.' (dot) operator and the desired metadata reserved keyword:

- *_type*: An enumeration value that identifies the type of the attribute.
- *_defaultValue*: The default (startup) value of the attribute.
- *_minValue*: Minimum value the attribute can represent.
- *_maxValue*: The maximum value the attribute can represent.
- *_loloAlarm*: The LOLO alarm threshold.
- *_loAlarm*: The LOW alarm threshold.
- *_hiAlarm*: The HIGH alarm threshold.
- *_hihiAlaram*: The HIHI alarm threshold.
- *_permissions*: An enumeration value that identifies the read-write permissions of the attribute.

6.3.1.2 Set

The `set` method provides a write interface for devices. Clients are able to set the value of one or more device attributes through an attribute set list. Any attributes defined in the set list which are not supported by the component are ignored. The status of the command is returned to the user as an attribute list, in a similar fashion as described above for the synchronous and asynchronous `get` methods. The returned attribute list contains the overall status of the `set` operation, which uses the reserved keyword "*_OperationResult*" as the attribute name, and a list of the attributes that were set during the operation. The "*_OperationResult*" attribute can have one of the following values:

- *"Success"* – The set operation completed without problems.
- *"Partial"* – One or more attributes could not be set.
- *"Fail"* – The set operation was rejected in entirety.

The list of attributes that were set during the operation contains the actual values that were set in device, including any modifications made by the device (e.g., clipping values that are out of range, etc.). The API for the `set` method is:

```
AttributeList set(AttributeList setList);
boolean set(AttributeList setList, ICommandCallback callBack);
```

6.3.1.3 Execute

The `execute` method provides a means to initiate a task or operation on a device, similar to invoking a class method with arguments. A noticeable difference between execute and the other API methods is that `execute` accepts a `CommandSet` in place of an `AttributeList` to define parameters and values. The `CommandSet` implementation is a subclass of `AttributeList`, and provides additional capabilities related to invoking and executing a device task. Specifically, `CommandSet` adds the concept of an action. "*_Action*" is a reserved keyword that allows a device to identify the operation the client wishes to execute ("*_Action*" is the attribute name and its associated value in the attribute list represents the action to be performed). The associated attributes needed by the operation are added to the `CommandSet` the same as they would be for an `AttributeList`. `CommandSet` also adds a mechanism to identify and modify the execution of the task through a unique identifier. The task identifier is created automatically by the `CommandSet` and can be used to change the state of an issued operation through the methods `pause`, `resume`, and `cancel`. `pause` suspends the execution of a command that is in progress, `resume` resumes the execution of a previously paused command, and `cancel` cleanly stops the execution of a command that is in progress. A helper function to return the unique ID from a command set that has been executed is provided. The API for the execute method is:

```
boolean execute(CommandSet commandSet, ICommandCallback callBack);
boolean pause(long commandID);
boolean resume(long commandID);
boolean cancel(long commandID);
```

The command ID helper function is implemented as a method on the `CommandSet` class:

```
long getCommandID();
```

6.3.1.4 Monitors

In addition to the basic device methods described above, KCSF devices support a monitor capability. Monitors are used to request notification when a device attribute changes in value. The user creates a monitor using the `addMonitor` method, specifying the target component, the attributes to be monitored in an attribute list, a callback object to be used when a monitor is triggered and a rate at which the monitor will test for changes. There are several additional methods required to manage monitors: `addToMonitor` adds attributes to an existing monitor attribute list, `removeFromMonitor` removes attributes from an existing monitor's attribute list, `removeMonitor` cancels and removes an existing monitor, and `refreshMonitor` requests all the current values from an existing monitor's attribute list. The API for monitors is:

```
boolean addMonitor(String component, IAttributeList attributeList,
  ICommandCallback callback, int rate);
boolean addToMonitor(String component, IAttributeList attributeList);
boolean removeFromMonitor(String component, IAttributeList
  attributeList);
boolean removeMonitor(String component);
boolean refreshMonitor(String component);
```

6.3.1.5 Callbacks

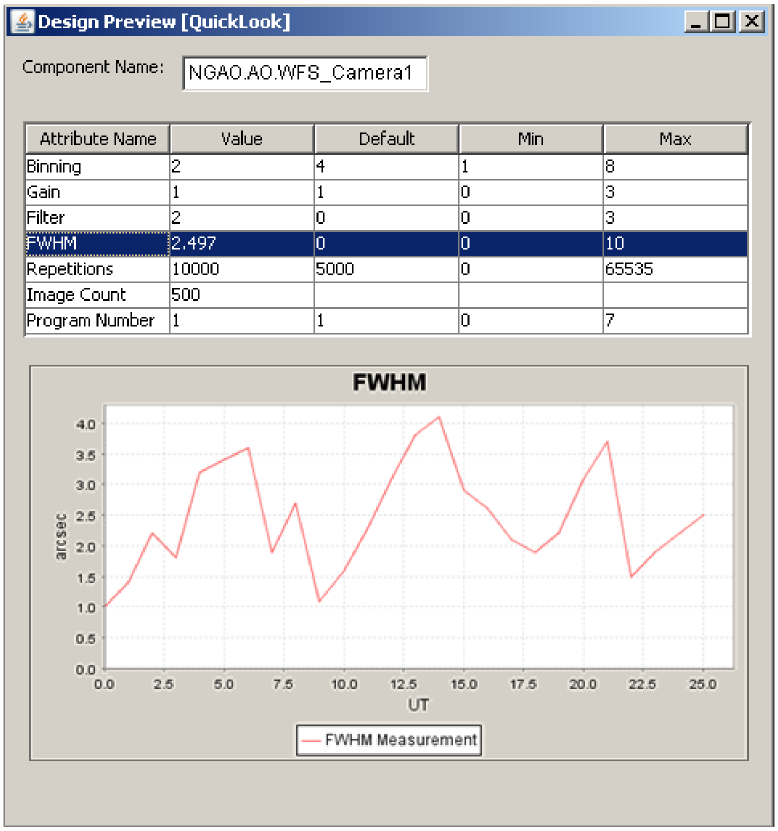
Several KCSF methods require the use of callbacks. We have defined a standard callback interface for commands, `ICommandCallback`, along with several callback methods.

`getComplete`, `setComplete`, and `executeComplete` are used to notify a requesting application that its associated `get`, `set`, or `execute` command request has completed. Similarly, there is `monitorTriggered` method to indicate that a requesting application's monitor has been triggered. Each method returns an attribute list containing the details of its respective command completion or monitor triggering. The callback API is:

```
AttributeList getComplete(IAttributeList attributeList);
AttributeList setComplete(IAttributeList attributeList);
AttributeList executeComplete(IAttributeList attributeList);
AttributeList monitorTriggered(IAttributeList attributeList);
```

6.3.1.6 Diagnostic Quick-Look

Using the concepts of supported attributes and metadata along with the get method and monitors presented in this section, a diagnostic "quick-look" widget can easily be created to display all the relevant information about a device. The idea is that the widget is launched and the user enters a device name. The widget then queries the device to get the list of supported attributes and metadata. Monitors are used to watch for attribute changes and to generate the display. An example is shown below in **Figure 11** for a CCD camera.



Component Name: NGAO.AO.WFS_Camera1

Attribute Name	Value	Default	Min	Max
Binning	2	4	1	8
Gain	1	1	0	3
Filter	2	0	0	3
FWHM	2.497	0	0	10
Repetitions	10000	5000	0	65535
Image Count	500			
Program Number	1	1	0	7

Figure 11: An example "Quick-Look" widget.

6.4 Services

So far, we have discussed components (devices and controllers), the container-component model, and the basic component API in some detail. However, in order to serve as a foundation for a distributed control system, the framework must provide more than just components and the ability to manage their lifecycles. Components must be able to do the following:

- Locate other components by name and communicate with them as required.
- Configure themselves at startup and upon request during run-time.
- Periodically check and report on their overall health and status.
- Send messages to logs.
- Generate and respond to alarms.
- Generate messages or take action in response to system events
- Archive data to files
- Communicate with legacy systems.

The KCSF provides all this functionality in the form of “services”, that are an integral part of the overall framework. Some of these services are required for normal operation (connection, configuration, alarming, etc.), while other services are optional and may be used by components as needed.

The KCSF provides the following services, which are discussed in further detail in the remainder of this section:

- Connection Service - provides the ability for devices to connect with each other.
- Alarm Service - provides the ability for components to set, clear and respond to alarms and for uniform handling of system error conditions.
- Event Service - provides the ability to generate event-based messages using the underlying publish-subscribe capability of the framework communications system.
- Log Service - provides a standard means of logging many types of messages.
- Health Service - provides for periodic monitoring and reporting of system and component health.
- Configuration Service - provides the ability to create and consume class-specific and instance-specific configuration data.
- Archive Service - provides the ability to archive system data for future use.
- CA client service – provides the ability to communicate with other observatory systems using the legacy EPICS Channel Access communications middleware.
- CA server support – provides the ability for other observatory systems using the legacy EPICS Channel Access to communicate with KCSF devices.

These services are not exclusive to just components – applications and scripts may access the services as well. All of the services with the exception of the configuration service and the CA server support may be used by scripts or applications. An application needs only to load the KCSF toolbox and then has access to the services.

6.4.1 Connection Service

The connection service provides for peer to peer communications between components by supporting the registration of components with the KCSF communications system, and by supporting communications between registered components. The following basic functions are provided by the connection service:

- Registration - the current component is registered by name with the KCSF 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 KCSF common services. Component developers do not need to perform these two actions in their code. Each component has a unique fully qualified name derived from its parent container, and this uniquely indentifies it in the system. The container automatically registers its components with the communication subsystem when they are first loaded and all that is needed by an application or remote component to make a connection is the fully qualified name. Once a connection has been made by a source component to a target component, the source component may issue commands to the target component. The connection service API for connecting and disconnecting to a component is as follows:

```
IRemote connect(String target);
void disconnect(String target);
```

`IRemote` is the client interface to a connected device (local or remote) and `target` is the fully qualified name of the target device. For example, to connect and send commands to an AO camera c1, the following simple commands are all that is required:

```
Device c1 = connection_service.connect(“AO.Camera.c1”);
c1.execute(.....);
```

Moreover, remote components appear to be the same as local components, so the user has no need to know where in the system a particular component is located. This is referred to as location transparency.

6.4.1.1 Example

The following example demonstrates how a developer might use the connection service. Using the component's name, one can obtain a reference to the component and invoke operations on it as if it is a local object. Note: a fully qualified name is required to make the connection to the component. However, once the connection to the component has been made, a fully qualified name is not required to identify its specific attributes. Either an attribute name or a fully qualified attribute name may be used. We show examples of both.

```
// Obtain a reference to the component and downcast appropriately
// A fully qualified name is required here:
String componentName = "system.subsystem.device"
   IDevice device = (IDevice)App.connect(componentName);

// Invoke operations
try {

   // Construct a list of status items to report on -- posn, rate, ...
   IAttributeList statusReq = new AttributeList();
   // Use a fully qualified attribute name here:
```

```
   statusReq.insert(new Attribute("system.subsystem.device.position", ""));
   // Use just the attribute name here:
   statusReq.insert(new Attribute("rate", ""));

   // Obtain the values from the device
   IAttributeList result = device.get(statusReq);
   result.show("device status is: ");

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

6.4.2 Alarm Service

The alarm service is just one part of the larger NGAO alarm system. The alarm system is discussed in more detail in Section 6.5. The alarm service provides the capability for components to set and clear alarms that are recognized by the larger alarm system. An alarm can be considered an event except that it has an associated abnormal condition, which requires special attention outside the control application (events are discussed in Section 6.4.3).

Alarms have several characteristics associated with them: area, source, condition, category, state and severity:

- Area: The area is the location or sub-system where the alarm originated (e.g., AO system, Laser system, LGS Wavefront sensors, etc.).
- Source: The source is the owner or originator of the alarm, which is typically a KCSF component. Any component can be an alarm source.
- Condition: The condition is the name for the abnormal state of the source. Examples are “Bad input device”, “High alarm limit”, or “Disk space is low”.
- Category: The alarm category is a grouping of alarm conditions which are functionally related. Examples of alarm categories are process inputs, process outputs, or system status.
- State: There are four states associated with alarms:
    - Inactive/Acknowledged (not included in active alarm list)
    - Active/Unacknowledged
    - Active/Acknowledged
    - Inactive/Unacknowledged
- Severity: The alarm severity is a number indicating the relative severity of the alarm. The alarm severity is programmed into the configuration database and is not directly programmable by the user. This is to ensure consistency of alarm severities throughout the NGAO system and to avoid hard coding severities in the applications.

The alarm service provides the capability for KCSF components to set or clear alarms using the following API:

```
SetAlarm(String Category, String Condition, Object AlarmValue);

ClearAlarm(String Category, String Condition);
```

When an alarm is set or cleared through the API, the source (component name) is automatically inserted into the alarm text by the service. In addition, extra information such as alarm severity, available through the alarm configuration, is added by the service. When setting an alarm it is possible to give an actual value. This field may be used to save the value of some key variable associated with the alarm or event. Any value set represents the value at the time of the alarm or event occurrence. A good example might be a temperature limit. For example,

```
      SetAlarm(“Process_input”, “HI”, Float(43.23));
```

might generate an alarm text like “AO.Cameras.Lbwfs Hi alarm limit exceeded, reading 43.23”

In addition to the component interface, the Alarm Service provides an interface to clients (e.g., the Alarm Summary Display) for the purpose of querying alarm settings (areas, categories, condition names, source conditions). The clients can use this information to produce filtered lists for user display. The APIs for these methods have not yet been fully defined, but the method names are:

```
      String[] QueryCategories();
      String[] QueryAreas();
      String[] QueryConditionNames(String Category);
      String[] QuerySourceConditions(String Source);
```

6.4.3 Event Service

Events are based upon the publish-subscribe communications system provided by KCSF. 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 themselves are simple name-value pairs. The event name is a single word (i.e. no spaces), typically written using camelCase notation. The event value is an attribute list, but the event service helper class provides convenience methods for automatically embedding other types within the attribute list.

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 normally 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.

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.

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

The API is as follows:

```
void subscribe(String eventName, IEventCallback callback);
void unsubscribe(String eventName);
void post(String eventName, IAttributeList aList);
void post(String eventName, long value);
void post(String eventName, double value);
void post(String eventName, String value);
void post(String eventName, IAttribute aValue);
```

6.4.3.1 Event Service

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 KCSF.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 any key to quit");
System.in.read();
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);
}
```

6.4.4 Logging Service

The logging service provides the ability to write log messages to one of three types of logs: console, file, and database. It is described in detail in KAON 673: NGAO Software Architecture: Logging Service. The logging service uses the standard Java logging service but wraps the Java classes to provide a simpler KCSF interface. Moreover, the specific logs used by a device are assigned during configuration by its parent container using information from the configuration service (described in a section below), so the user has a simple interface to generate log messages. A brief overview is given here along with the API.

6.4.4.1 Log levels

The following log levels are defined:

```
public enum LogLevel {
      ALL,          // Log all messages
      TRACE,        // Trace the path of execution
      DEBUG,        // Debug messages
      INFO,         // Information message, normal operation.
      WARN,         // Suspicious operation, possible problem.
      ERROR,        // Recoverable user error has occurred.
      CRITICAL,     // Critical system error, recoverable.
      EMERGENCY,    // Non-recoverable severe failure
      OFF           // Disabled logging
}
```

6.4.4.2 Log message format

The following format will be used as the standard for human readable log messages:

```
<Time> [<LEVEL>] <Source> - <Message>
```

The *Time* component will be displayed as:

```
<Month> <Day>, <Year> <24-HourTime>
```

The *LEVEL* component will be the level of the message – one of the Log Level enumeration values. *Source* is the name of the component that generated the log message. This will allow a user to identify the specific software object that produced the message. Source name injection is performed automatically by the logging service. The *Message* component will be the raw message passed to the log service.

A typical message will look something like:

May 7, 2009 13:14:53.378 [TRACE] ngao.ao.wfs.camera1 – Invalid arguments.

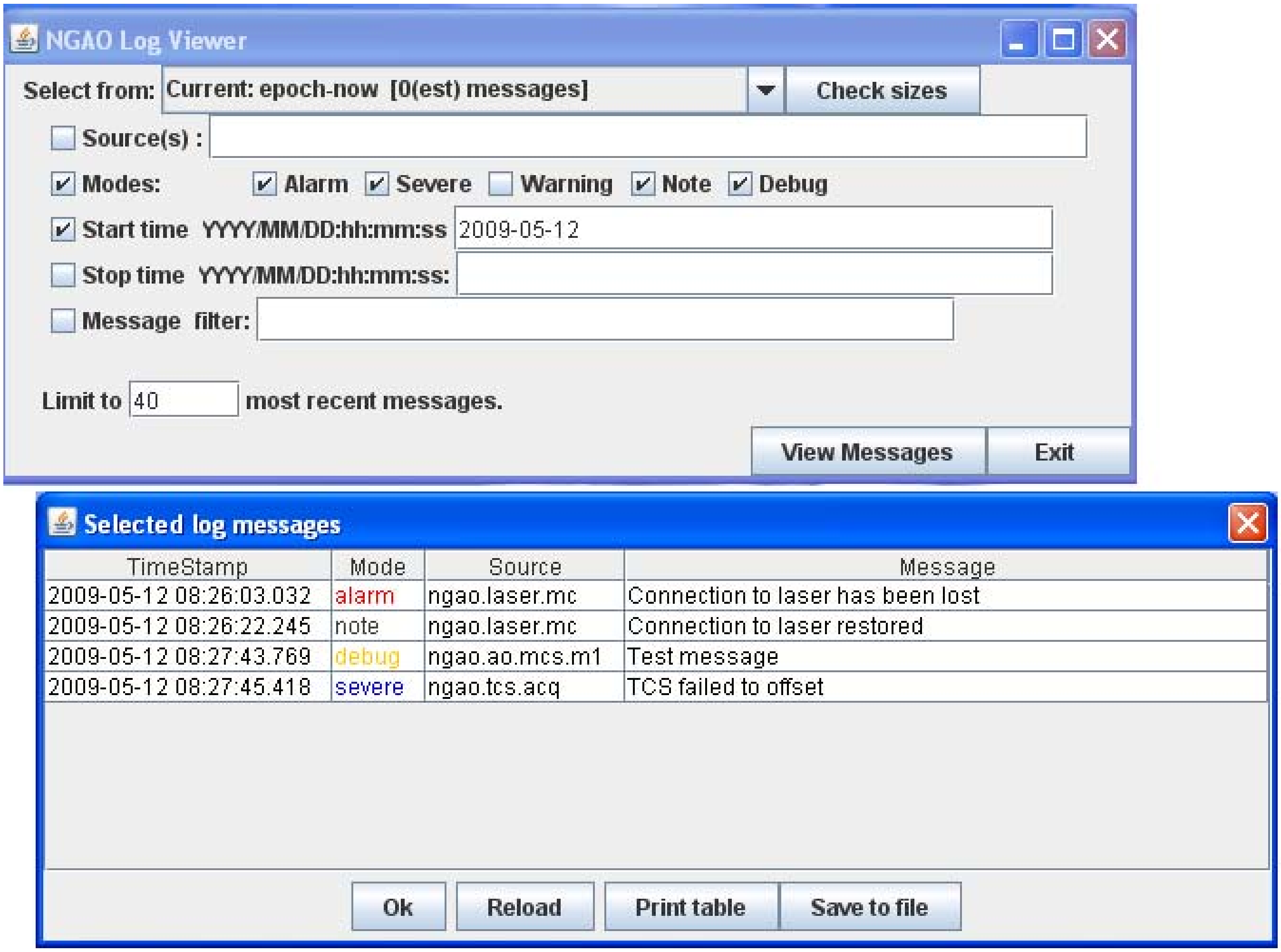
6.4.4.3 API

The user interface to the logging service defines a generic log method where the user specifies the log level and the message, along with separate methods for each log level which only require a message. These methods are defined as follows:

```
void log(LogLevel Level, String Message);
void trace(String Message);
void debug(String Message);
void info(String Message);
void warn(String Message);
void error(String Message);
void critical(String Message);
void emergency(String Message);
```

6.4.4.4 LogViewer

As part of the framework a User Interface would be provided to allow the log messages to be viewed and processed as needed. A rough prototype is shown below. Log messages could be filtered based on date, time, message type, sources or even on message content. The resulting messages could then be printed or saved to a text file. It is possible that a number of file formats could be supported.



Select from: Current: epoch-now [0(est) messages]

□ Source(s) :

☑ Modes: ☑ Alarm ☑ Severe □ Warning ☑ Note ☑ Debug

☑ Start time YYYY/MM/DD:hh:mm:ss 2009-05-12

□ Stop time YYYY/MM/DD:hh:mm:ss:

□ Message filter:

Limit to 40 most recent messages.

TimeStamp	Mode	Source	Message
2009-05-12 08:26:03.032	alarm	ngao.laser.mc	Connection to laser has been lost
2009-05-12 08:26:22.245	note	ngao.laser.mc	Connection to laser restored
2009-05-12 08:27:43.769	debug	ngao.ao.mcs.m1	Test message
2009-05-12 08:27:45.418	severe	ngao.tcs.acq	TCS failed to offset

Figure 12: Log Viewer UI Prototype

6.4.5 Health Service

The ability to quickly ascertain device health throughout the NGAO control system is very important. KCSF supports this by providing a health monitoring service. The health of each component is monitored by its parent container. The various health states for a component are: good, ill, bad, and unknown, and are defined as follows (in order of worsening health):

- Good: No problems have been detected by the component, it is fully operational.
- 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. The component is partially operational.
- 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 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 severity switches from warning to severe.

Component developers must implement the following API:

```
        String performCheckHealth();
```

The return values are:

```
        Health.GOOD
        Health.BAD
        Health.ILL
        Health.UNKNOWN
```

The container will call `checkHealth` on a periodic basis and report the results (checkHealth calls the delegate method `performCheckHealth`). This is illustrated below in Figure 13.

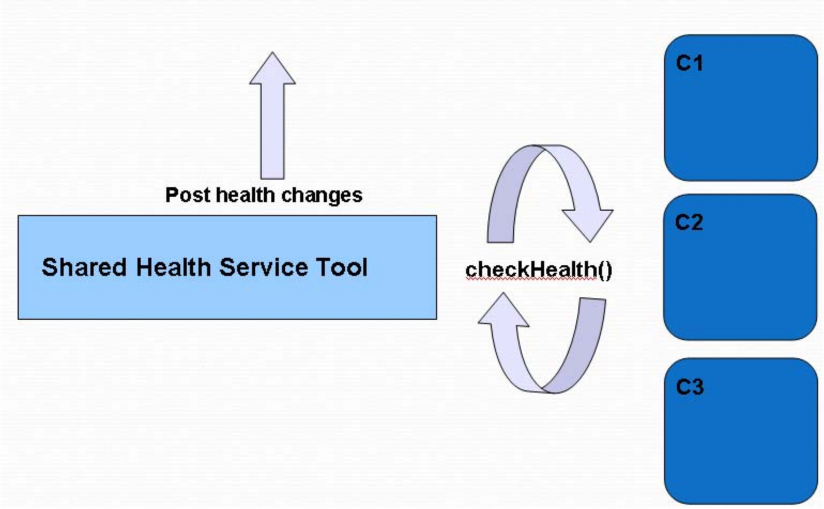
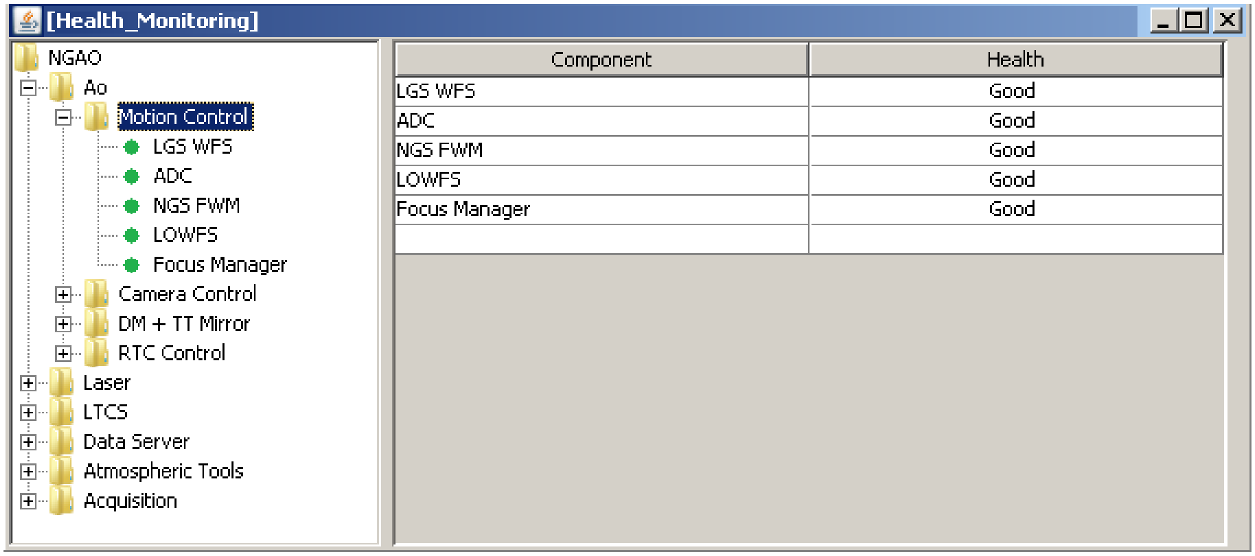


Figure 13: An illustration of the health service. The container periodically polls its components as to their health status and posts health changes to a health event.

It is quite simple to create a central health service monitor which subscribes to the various health service events throughout the system and posts the data to a centralized display. A mock-up of this concept is shown below in Figure 14.



Component	Health
LGS WFS	Good
ADC	Good
NGS FWM	Good
LOWFS	Good
Focus Manager	Good

Figure 14: The health monitoring summary display.

6.4.6 Configuration Service

The configuration service is just one part of the larger configuration system in KCSF. The configuration system combines the configuration service with a configuration database and associated tools to manage the configurable properties of each component in the NGAO system. The configuration service provides the capability for each component to request its own configuration information from the configuration database through the following API:

```
interface IConfigurationService extends IServiceTool {
  public IAttributeList getContainerManagerConfiguration(String ManagerName);
  public IAttributeList getContainerConfiguration(String ContainerName);
  public IAttributeList getComponentConfiguration(String ComponentName);
  public IAttributeList getMetaData(String AttributeName);
};
```

When components are created by their parent container, they are connected to the available services. Next, during initialization, each component calls the configuration service to access its specific configuration data. There is a specific method for Container Managers, Containers, and Components that accept a fully-qualified name used to lookup the information relevant to each of these types. The Configuration Service utilizes attribute lists to return data to the invoking object. The details of this process are handled automatically by the parent container with no action required on the part of the user.

The configuration database and its associated tools are discussed in more detail in Section 6.6.

6.4.7 Archive Service

The archive service is intended to store bursts of engineering data to a relational database. Once in the database the data can be viewed through the framework user interfaces, or processed at a later stage as needed. The archive service is intended for recording short data sets for troubleshooting, engineering tests, etc., and is not meant to be used for recording large data sets, such as telemetry, throughout an observing run. The archive service is not meant to be an implementation of or a replacement for the NGAO data server; however, the data server may serve as the data store for the archive service.

Data is stored in a relational database table that has the following fields: timestamp, source, name and value, where

- timestamp is the time the value was recorded in UT (YYYY/DDD:HH:MM:SS.SSS)
- source is the name of the component or application that owns the attribute
- name is the name of the attribute
- value is the value of the attribute

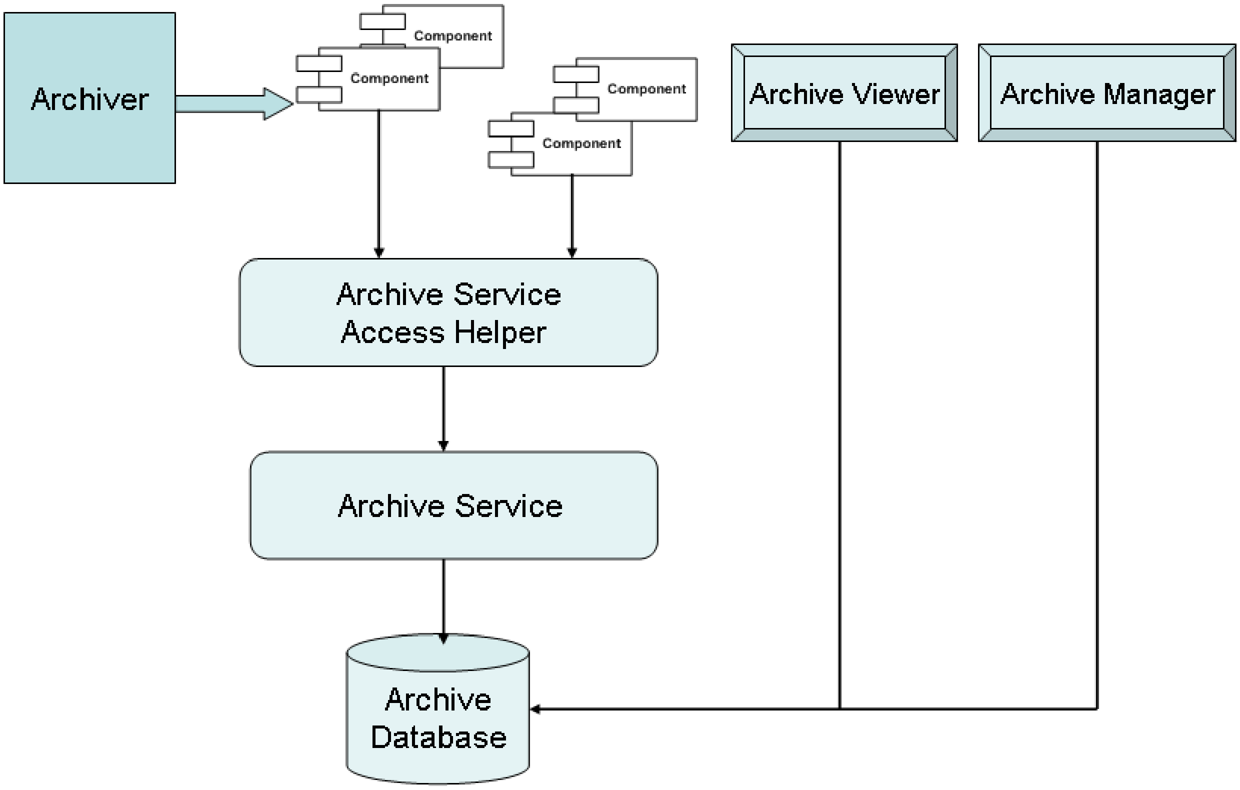


Figure 15: Archive Support Overview.

The framework provides two user interfaces: an Archive Viewer and an Archive Manager. Through the Archive Viewer, users can select the sources, attribute names, and start/stop ranges of timestamps for the arhive data to be viewed. Archive data is provided through a table where the UI supports sorting and filtering. The Archive Manager is a UI for managing Engineering data archives. Through the GUI, users can see the names of existing archives, and create new archives. This GUI is intended for system maintenance only.

KCSF supports both a push and a pull model for archiving. The push model allows component developers to decide what to archive and when by making an appropriate call to the service. The pull model has a collector that is running and it can be configured to archive data at a periodic rate or on an event. The Archiver will command the appropriate components as needed to archive the attributes of interest.

A component or application developer uses the archive service access helper API to store attributes on demand, or when requested by the archiver. The API supports the archiving of single attributes at a time. This could easily be expanded to support AttributeLists if that became a requirement. When an attribute is archived the service automatically adds a timestamp and the source name of the attribute to the database entry. The API is as follows:

- `void archive(Attribute attribute);`
    - Save an attribute.
- `void archive(String name, AttributeValue value);`
    - Save a generic name-value pair.
- `void archive(String name, <Type> value)`
    - Save a name-value pair where the value is a particular type.

Various flavors of the service can be supported such as a buffered archive service which would buffer archive messages internally, writing them out on some interval to achive better performance at the expense of some latency.

6.4.8 Channel Access Client Service

The Channel Access (CA) client service allows KCSF components and applications to read, write and monitor EPICS channels external to the NGAO system using the channel access protocol. The implementation of the client service is based on Channel Access Java library from CosyLab. The API is as follows:

```
boolean put(String channelName, <type> value);
boolean put(String channelName, <type>[] value);
boolean addMonitor(String channelName, IMonitorCallback cb);
boolean removeMonitor(String channelName);
<type> get<Type>(String channelName);
<type>[] get<Type>Array(final String channelName);
```

Where type can be byte, short, int, float, double or string. The put methods perform writes to an external channel, while the get methods perform reads. Calls to gets, puts and monitors access the appropriate CA process variable referred to by the channel name, for example, `dcs1.axe.axestat`. Get and put are all synchronous but asynchronous support can be easily added. All monitoring is asynchronous.

Monitor callbacks occur through the `IMonitorCallback` interface, which has the following API:

```
Public interface IMonitorCallback {
    public void onDataChanged(String name, Object value);
    public void errorEncountered(String name, String msg);
}
```

Data changes are reported through the `onDataChanged` method and errors are reported through the `errorEncountered` method. Currently the monitor is designed to only report on data changes, but if needed these can easily be expanded to include alarm condition monitoring also.

The following shows a simple example:

```
public void getCAServiceTest() {
   Ca.addMonitor("dcs1.axe.axestat", new MonitorCallback());
   System.out.println(Ca.getFloat("dcs1.axe.az.Bcode"));
   System.out.println(Ca.getFloat("dcs1.axe.el.Bcode"));
   System.out.println(Ca.getString("dcs1.axe.errstr"));
   Ca.put("dcs1.axe.pnt.wrapctrl", "shortest");
}

public class MonitorCallback implements IMonitorCallback {
   public void onDataChanged(String name, Object value) {
      System.out.println(“Received data for ” + name);
   }
   public void errorEncountered(String name, String msg) {
      System.out.println(“Monitor for ” + name + “ has err: ” + msg);
   }
}
```

6.4.9 CA Server Support

The KCSF will provide a component that is a CA Server. Server support allows for outside systems to act as a master pushing and pulling data to and from KCSF based on their timing needs. Like any component it can be loaded, unloaded, initialized, started and stopped. When initialized the component will instantiate an internal cache of all exported process variables. The cache will include their corresponding fully qualified KCSF attribute name and possible KCSF type. Once the component is started the CA Server context will be set running and will continue to do so until the component is stopped at which time the CA Server will be shutdown.

The server needs only to be configured so that fully qualified attribute names can be mapped to channel names. Once running, any time a CA client broadcasts a search for a particular process variable (PV), identified by a name, the server will check to see if it supports it. It if does then a positive answer is returned through the CA protocol. Once a CA client knows a process variable exists, it will most likely issue a request for channel creation. A channel is a connection between the server and client through which a single process variable is accessed. As per the CA protocol, the client never talks directly to a process variable, only through the channel. The CA Server will create a new channel as needed and will map all put and get calls on that channel to the appropriate component and attribute.

For KCSF the process variable names requested by the CA clients need to be translated to KCSF addresses. In the implementation the KCSF CA Server will create a process variable that is mapped to a component attribute. PV writes will be issued through the component *set* command and reads will be serviced using the `get` command. Monitors are also supported.

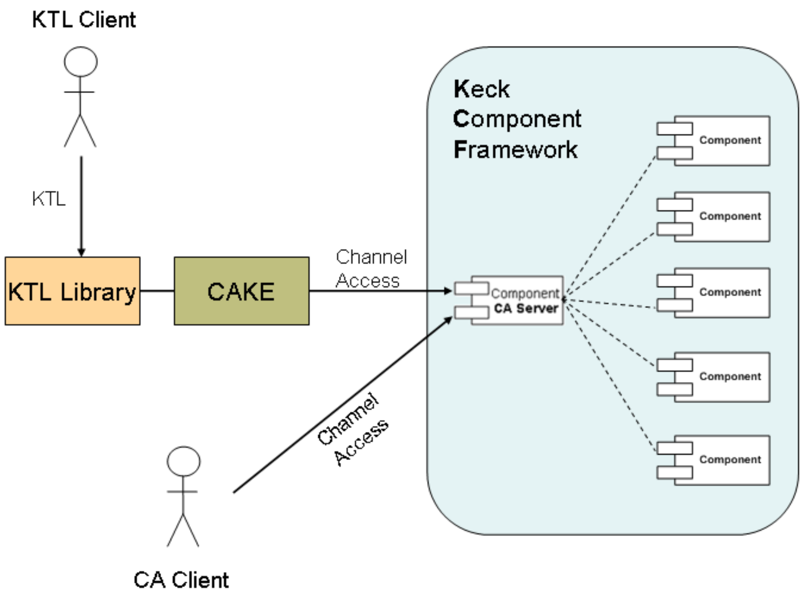


Figure 16: An illustration of the KCSF CA Server and external CA/KTL access.

It is expected that by utilizing the current Channel Access Keyword (CAKE) layer, which is a thin generic software layer providing KTL keyword access to EPICS systems (or more correctly systems that honor the CA protocol), the KCSF can be seamlessly exposed to existing KTL clients. This is shown above in Figure 16.

6.5 Alarm System

Alarms are reports of component failure (software or hardware) or other abnormal behavior within the system. Alarms occur asynchronously in a random fashion. Some alarm conditions may clear themselves and others may require operator intervention. The KCSF Alarm System provides a capability for components to set and clear alarms, and a managing system to detect the occurrence of alarms and to initiate the appropriate system response. The Alarm System is described in detail in KAON 677: NGAO Software Architecture: Alarm System; only a brief overview and the main APIs are discussed here.

The main features of the Alarm System are:

- Alarms are defined based on condition rules for device properties.
- Alarms are reported to a central alarm manager that logs and tracks the state of all alarms.
- Automated acknowledgement and recovery from alarms when system conditions permit, and manual operator acknowledgement and recovery from other alarms, as required.
- Alarms may be organized hierarchically, especially for display to an operator.
- It is possible to define actions that can be automatically started on the occurrence of specific alarms.
- Different annunciation methods can be applied to different alarms.
- Alarms can be associated with displays, help pages, diagnostic pages, etc.
- Applications may request to be notified of alarms.

The alarm system consists of three main sub-systems: the Alarm Service, the Alarm Manager and the Alarm Summary Display. This is illustrated in Figure 17, below. The Alarm Service is the means by which all NGAO components report their alarm statuses and is available to every container. Components use the service to set and clear alarms. The Alarm Service for each container synchronizes alarm state information with the system-wide Alarm Manager on initialization and sends all alarm state changes to the Alarm Manager during normal run-time operation. The Alarm Manager is the central coordinating function for managing all the alarms in the NGAO system. It maintains a list of all active and unacknowledged alarms in the system and makes this list available for user interface controls and displays. The Alarm Manager also logs all alarm state changes to a historical database. The Alarm Summary Display provides the user with a summary view of all active and unacknowledged alarms from the alarm area of interest. The user can view detailed information about each alarm from this display. This display may also be used to acknowledged alarms.

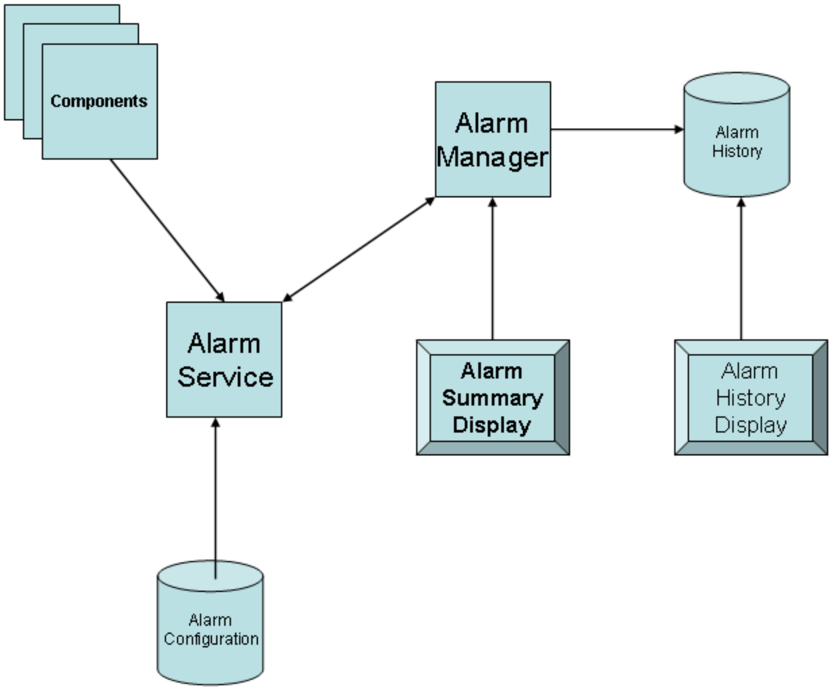


Figure 17: The KCSF Alarm System Architecture.

6.5.1 Alarm Configuration

As discussed in the description of the Alarm Service in Section 0, alarms have several characteristics associated with them: area, source, condition, category, state and severity. To help ensure consistency of usage in the alarm system and to avoid hard coding alarm values in the individual component applications, the data for these fields will be managed using the configuration database. There will be an Alarm Setup Display tool to aid in the configuration and management of all the alarms in the system. We anticipate at least two alarm tables: a simple table maintaining the list of all sources and another maintaining the list of alarm categories. The source table is expected to be similar to the following:

Field Name	Description
Name	Source Name
Alarm category	Associated Alarm category
Area	Area may be used to associate an alarm with a particular section of the observatory. The area associated with each alarm can be viewed at the Alarm Summary Display. Alarms can also be sorted and filtered by area at the Alarm Summary Display. A blank area name can be selected if no area is to be associated with this alarm record (default value).
Description	
Alarm help	Instance specific help.
Severity	The severity field is used to define the severity or importance of each alarm. Severity may range from 0 to 1,000 with 0 being the lowest severity and 1,000 being the highest. Alarm severity determines the alarm text color on the Alarm Summary Display. Alarms may be sorted or filtered by severity on the Alarm Summary Display. For compatibility reasons, severity values of 0 through 3 are mapped by the system to higher severity values as follows: 0 = 100, 1 = 300, 2 = 600, and 3 = 950.
Ack required	This field determines whether or not operator acknowledgement of this alarm is required. If AckRequired is set to "Yes" then this alarm must be acknowledged by the operator (and cleared by the application) before it will be removed from the Alarm Summary Display. If AckRequird is set to "No" then this alarm is considered an informational alarm, which does not require operator acknowledgement before it is removed from the Alarm Summary Display (this alarm will be removed from the Alarm Summary Display when cleared by the application).
Disabled	The Disabled field is used to disabled individual alarms. Alarms which have the Disabled field set to "Yes" cannot be set by the application, will not appear in the Alarm Summary Display and will not appear in the alarm history log.
Logging disabled	If the LogDisabled field is set to "yes" then changes to the state of this alarm (alarm set, clear or acknowledge) are not logged to the alarm history file. When this field is "No" (which is the default) alarm state changes are logged to the alarm history file.
Associated Display	One associated display can be defined for each alarm record. When an associated display is defined for an alarm then the operator can navigate directly to the associated display from the Alarm Summary Display whenever the alarm is active or unacknowledged.

Each alarm record is associated with one alarm category. The alarm category determines the possible alarms for a source. For example, alarm records with a category of "Process_Inputs" might include the following alarms: Bad input device, High alarm limit or Low alarm limit. Alarm records with a category of "System_Status" might include the following alarms: Disk space is low, Virtual memory usage is too high, CPU usage is high or Fan has failed. The alarm category table is expected to look similar to the following:

Field Name	Description
Name	Category Name
Conditions	Text describing each condition that can exist for this category

6.5.2 Alarm Service

The Alarm Service is the application that permits components to set and clear alarms. It is discussed in detail in Section 0.

6.5.3 Alarm Manager

The Alarm Manager is the managing server for the alarm system. It maintains the information it receives from the Alarm Service via the `SetAlarm` and `ClearAlarm` APIs and acts as a server for alarm clients, most notably, the Alarm Summary Display. It implements the methods used by clients to connect to the manager to start receiving alarms, enable or disable alarms, acknowledge alarm conditions and refresh to get the current alarm statuses. The Alarm Manager also accepts the `OnAlarm` callback object so that clients can be informed when a particular alarm occurs. The APIs for these methods have not yet been fully defined, but the method names are:

- `EnableConditionByArea()`
- `EnableConditionBySource()`
- `DisableConditionByArea()`
- `DisableConditionBySource()`
- `CreateSubscription()`
- `AckCondition()`
- `Refresh()`
- `CancelRefresh()`
- `OnAlarm()`

6.5.4 Alarm Logging

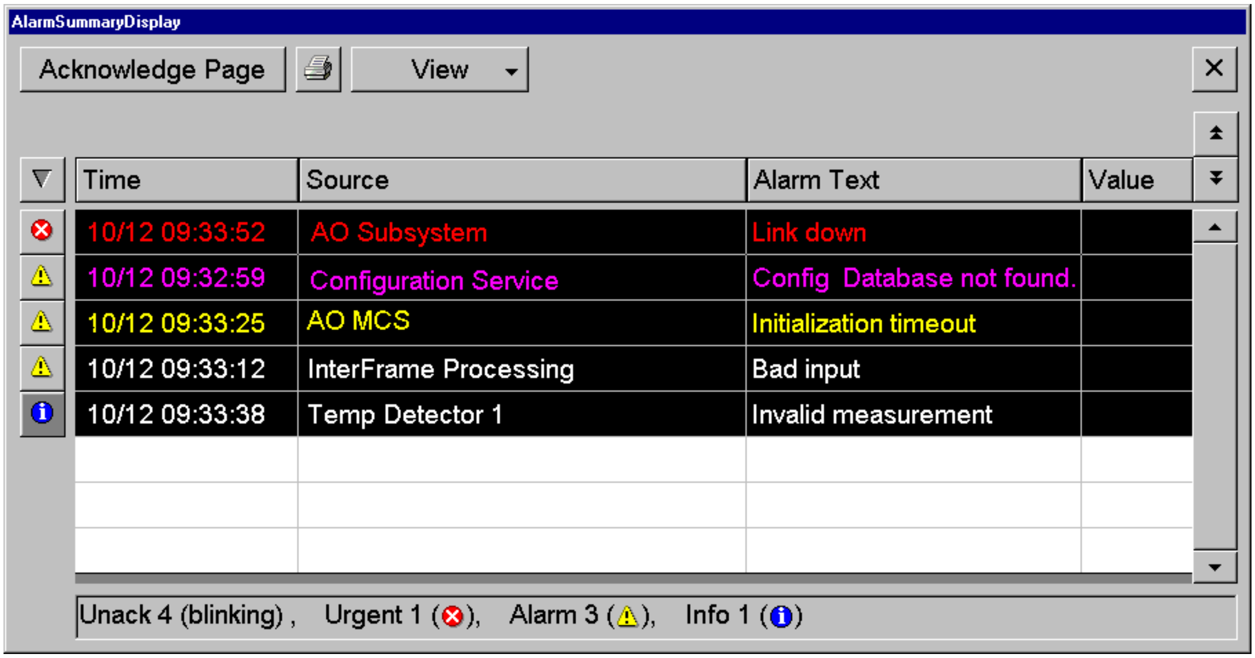
All alarms can be logged to a database. It is expected that there will be an Alarm Logging Display that can be used to view all recent alarm state changes and events. This display may be used to query the alarm history files in many ways. The alarms may be logged by the Alarm Manager or there may be a separate system wide component that is responsible for monitoring and logging alarms. The following shows a possible schema for the alarm table:

- **TimeStamp**: UT time when the alarm state change or event occurred.
- **Category**: Alarm category for this alarm. Alarm categories are defined by the alarm system configuration and are meant to provide logical alarm groupings.
- **Source**: Owner of the alarm or event.
- **Alarm Text**: Description of the alarm or event.
- **Action**: A classification of the alarm state change or event. Typical actions are Set Alarm, Clear Alarm, or Acknowledge Alarm.
- **Value**: This field may be used to save the value of some key variable associated with the alarm or event. Any value shown represents the value at the time of the alarm or event occurrence.

- **Operator**: May be used to record the operator performing the event or action.
- **Severity**: The severity (or priority) of the alarm or event. This may range from 1 to 1,000. Low values indicate a low urgency and high value represent a higher urgency.

6.5.5 Alarm Summary Display

The following shows what the Alarm Summary Display may look like and gives an indication of the functionality that could be made available. The snapshot is reduced in height for brevity. There can be many instances of the Alarm Summary Display running.



▽	Time	Source	Alarm Text	Value
⊗	10/12 09:33:52	AO Subsystem	Link down	
⚠	10/12 09:32:59	Configuration Service	Config Database not found.	
⚠	10/12 09:33:25	AO MCS	Initialization timeout	
⚠	10/12 09:33:12	InterFrame Processing	Bad input	
ℹ	10/12 09:33:38	Temp Detector 1	Invalid measurement	

Unack 4 (blinking) ,   Urgent 1 (⊗),   Alarm 3 (⚠),   Info 1 (ℹ)

Figure 18: The alarm summary display.

Alarms are listed on the Alarm Summary Display, which provides a one-line description of each alarm. The alarm summary is a ring buffer that can hold up to TBD alarms. The list can be scrolled using the vertical scroll bar on the right side of the display. What is shown on the Alarm Summary Display is configurable. For example, you can filter the Alarm Summary to show alarms of a particular priority only, or you can filter the Alarm Summary to show alarms for a particular area only. Additional fields such as alarm area and priority can also be shown. The display allows alarms to be acknowledged, if required. Alarm icons and color coding will be used to different high priority and urgent alarms, alarms that have not been acknowledged and so on.

6.6 Configuration system

The Configuration System provides the ability to manage the configurable properties of each component in the NGAO system. It consists of a configuration database, the associated tools to manage the database, and the configuration service, which provides the capability for each component to request its own configuration information from the configuration database. The configuration service was discussed earlier in Section 6.4.6. In this section we discuss the user view of the configuration system.

6.6.1 Configuration Model

The configuration system for the KCSF infrastructure is designed as a multilayered distributed model, as shown below in Figure 19.

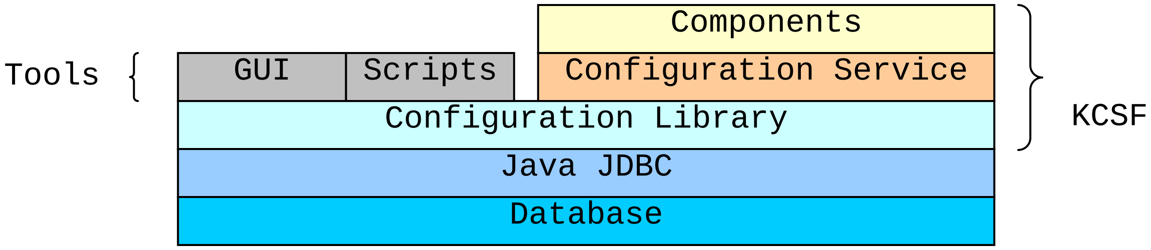


Figure 19: Configuration Layers.

Each layer of the model is responsible for implementing a black-box interface with the layer below, abstracting away the technical and functional details of database connectivity, I/O, and management. At the lowest level is the database, which will maintain the configuration data and preserve version information. Managing and communication with a database is simplified through the Java Database Connectivity API (JDBC). This native Java module provides an interface for connecting to a database and performing all of the standard database operations. JDBC is available for a number of database implementations including JDB, Sybase, SQL Server, and Oracle.

KCSF clients and components that wish to connect to the database will go through the Configuration Library. This API acts as the bridge between the client layers and database layers. Common technical tasks such as opening and closing connections to a database can be preformed in a single method call. The library also implements a number of functional tasks (such as retrieving and writing configuration values) as method calls, automatically converting the request into the appropriate SQL statement. Configuration data is also formatted into KCSF compatible attribute lists before being returned to clients. The configuration library is discussed in further detail below.

Above the Configuration Library sit the client tools and the KCSF Configuration Service. The tools provide users with full administrative control of the database and its contents. Clients are able to retrieve, modify and remove configuration data, as well as define new classes and instances for deployment. The Configuration Service provides read-only access to configuration data for devices and controllers. Managers, containers, and components can use this simple interface to retrieve fully formatted instance specific information by name.

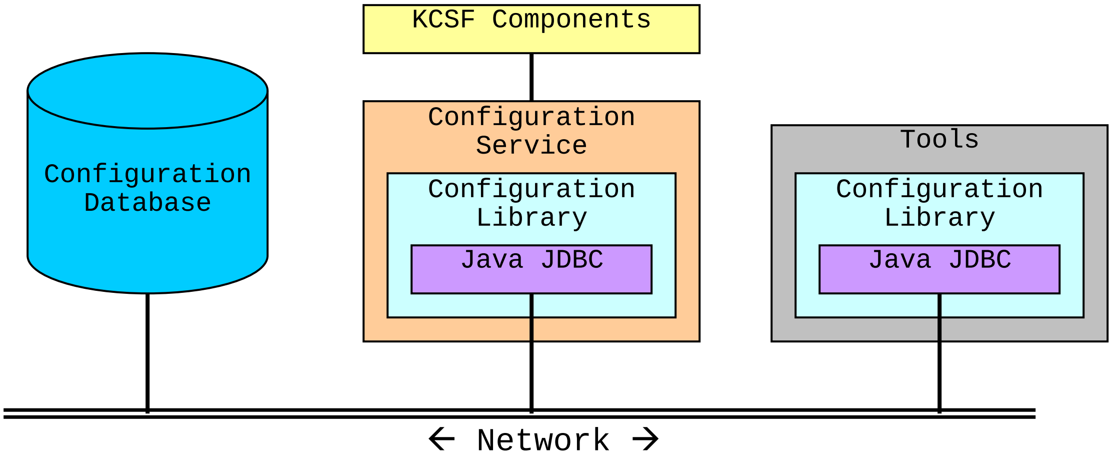


Figure 20: Distributed Database Access.

Components and tools will be distributed throughout the network. Typically there will be only one configuration database running for a system, although there is nothing that prevents multiple instance from running concurrently (redundancy, for example). A Configuration Service, Configuration Library, and JDBC instance will exist for every deployed container process. Tools, scripts, and GUIs will interface directly to the Configuration Library, and each instance will require its own JDBC. Tools and scripts will be started and stopped throughout the night, so the actual number of open database connections will vary. This is illustrated in Figure 20 above.

6.6.2 Configuration Library

The Configuration Library provides the methods, functionality, and administrative controls to the Configuration Database. Clients use the library to connect to the database through the KCSF Configuration Service and standalone tools. The configuration library is organized into two distinct interfaces: administrative and client. The client interface provides basic read-only access to configuration data: this is the standard interface used by the Configuration Service. The administrative interface adds to the client interface by providing functionality to modify the contents and structure of the database.

6.6.2.1 Client Interface

The client interface exposes read only capabilities to users. Within the Keck Component Framework, the primary user of the client interface is the Configuration Service. This service is the main access point for all KCSF components, and therefore, in the majority of systems, will constitute the highest number of connections to the database.

The following defines the client interface:

```
public class ClientInterface {
   public IAttributeList read(String name);
   public IAttributeList query(String statement);
   public IAttributeList diff(String versionID, String versionID = null);
};
```

The client interface has the following methods:

- *read* – Returns the data associated with the component specified by the fully qualified name. The following information is returned for each class of component:
    - *Container Manager* – The manager's properties and a list of all container names.
    - *Container* – The container's properties and a list of all component names.
    - *Component* – The component's properties and a list of all its attributes.
    - *Attribute* – The attribute's metadata.
- *query* – Returns the full qualified names of the components or attributes selected by the query.

6.6.2.2 Administrator Interface

The administrator interface extends the client interface by providing users with the ability to modify the state of the database. The Configuration GUI and other KCSF scripts will utilize this interface to interact with the database. Other scripts and tools can be developed by users to modify the database outside of the Framework. The administrator interface will be inaccessible from within the KCSF container / component framework.

The following defines the administrator interface:

```
public class AdministratorInterface extends ClientInterface {

   public boolean Modify (String name, String value);
   public boolean AddInstance(String name, String class);
   public boolean RemoveInstance(String name);
   public boolean LoadDatabase(String path);
   public boolean SaveDatabase(String path);
   public boolean CreateVersion(String ID);
   public boolean LoadVersion(String ID);
   public boolean RemoveVersion(String ID);
   public boolean DefineClass(String name, String inherits = null);
   public boolean DefineAttribute(String class, String name, String Type,
         IAttributeList metadata = null);
   public boolean RemoveClass(String name);
   public boolean RemoveAttribute(String name);

}
```

The administrator interface has the following methods:

- *Modify* – Modifies the value of the property identified by the fully qualified name. This must reference an attribute or metadata value (including manager and container properties).
- *AddInstance* – Add a new instance to the database of the specified type based on the fully qualified name. The hierarchy defined by the name must exist for an instance to be created.
- *RemoveInstance* – Removes an existing instance from the database. In the case of managers and containers, all children must be removed before subsequent removal.
- *LoadDatabase* – Loads a database from disk.
- *SaveDatabase* – Saves the current database to disk.
- *CreateVersion* – Saves a version of the active workspace within the database associated with the specified ID.

- *LoadVersion* – Replace the active workspace with the version specified by the ID. Modification to the active workspace will not modify the saved version, unless overwritten with a call to *CreateVersion*.
- *RemoveVersion* – Remove a version from the database.
- *DefineClass* – Create a new class definition placeholder in the database. The placeholder will be empty of attributes unless an inheritance is explicitly specified.
- *DefineAttribute* – Add an attribute to the specified class: all instances of the parent class will reflect the addition of this attribute. The attribute's type must be specified, and an optional list of meta-data value can be provided. Instances of this attribute will automatically be populated with the provided meta-data values. If the attribute already exists in the system the class definition will be updated. If the type has changed all instances of the parent class will be updated. If only the metadata has changed instance will not be affected.
- *RemoveClass* – Removes a class definition from the database. All instances of the class must be removed before subsequent removal.
- *RemoveAttribute* – Removes an attribute from a class definition: all instances of the parent class will be updated to reflect the removal of the attribute.

6.6.3 Administration Tools

Users and operators will be provided with a set of tools to configure and maintain the Configuration Database. Simple tasks such as loading a specific version, saving a new version, or making or restoring a database backup will be provided in the form of simple command line scripts. More complex tasks such as setting values and adding or modifying objects will be performed through an operator GUI. All script and GUI tasks are performed through the Configuration Library API.

6.6.3.1 Scripts

Scripts provide a simple and fast way to access the configuration database without having to bring up and navigate a GUI or tool. Since scripts are only command line tools they are not intended for complex operations. As such scripts only provide functionality for a subset of the available configuration capabilities. The following scripts will be available:

- Start and shutdown a Configuration Server / database.
- Save and restore a database backup.
- Save and load version(s).
- Text dump of the database.

6.6.3.1.1 Database Builder

An additional utility that can aid in the development process is a script that will convert code into a corresponding class configuration definition. This script will accept a path to one or more class files annotated with special tags that define the configurable attributes. When the script parses the file(s) it will create a class definition and add it to the database. This utility will allow developers to easily create the framework of a project without having to manually define classes twice – once in code the other in configuration.

The following example shows what would be generated by the script from a simple Java class.

Java Class:

```
/*
 * @ConfigAttribute double ExposureRate
 * @ConfigAttribute int Binning
 * @ConfigAttribute string Vendor
 */
class Camera extends Controller {
   public Camera();
   …

   protected double ExposureRate;
   protected int Binning;
   protected String Vendor;
}
```

Database:

ClassInheritance	
ClassName	Inherits
Component	
Controller	Component
Camera	Controller
…	

ClassDefinition						
ClassName	**AttributeName**	Type	DefaultValue	MinValue	MaxValue	…
…						
Camera	ExposureRate	double	0.0			
Camera	Binning	integer	0			
Camera	Vendor	string	“”			
…						

Figure 21: Class Definition Example.

6.6.3.2 Configuration GUI

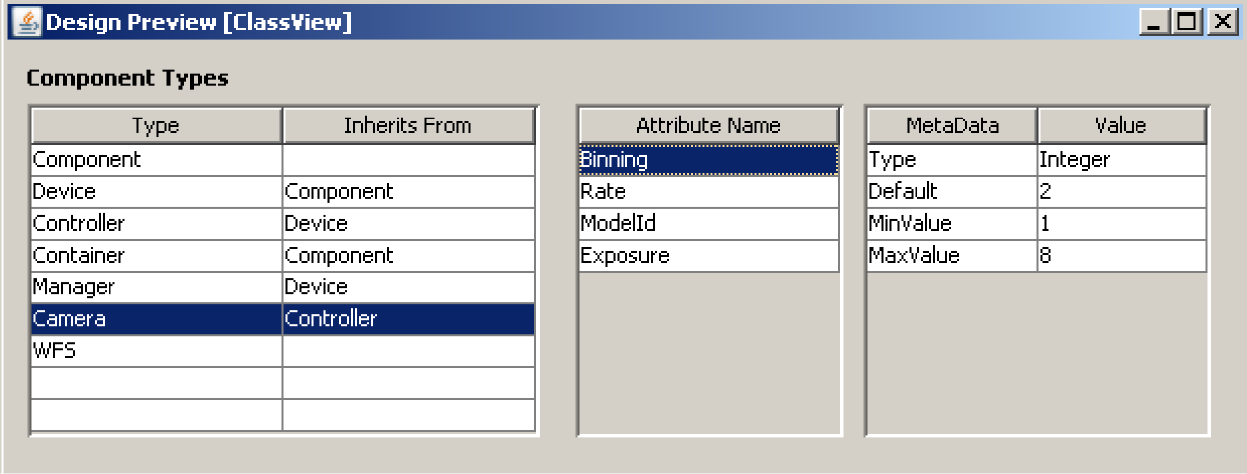
The Configuration GUI represents the primary means for full database control and manipulation. The GUI will provide all of the functionality available in the scripts, as well as utilize the full extent of the Configuration Library administrator interface:

- Add, modify and remove objects, attributes, and metadata.
- Perform complex queries and custom windowing.
- Data validation and version comparison.
- Display version history and details.
- Define base types and default values.

The GUI will be divided into two views: class definitions and instances. The class view allows users to define the structure of primitive and complex types in the database. Every KCSF class will have a corresponding definition in the database. Additional project specific types can also be defined. When it comes time to create a configuration for an application, the user will create an instance of a class definition. Instances are shown in the instances view, and define the configuration values for a specific object in the system. A user can modify the values of an instance, but can not modify its structure from this view.

6.6.3.2.1 Class Definitions

The Class Definition panel shows all of the database definitions of KCSF class types.



Type	Inherits From
Component	
Device	Component
Controller	Device
Container	Component
Manager	Device
Camera	Controller
WFS	

Attribute Name
Binning
Rate
ModelId
Exposure

MetaData	Value
Type	Integer
Default	2
MinValue	1
MaxValue	8

Figure 22: Example Class View

In this panel a user can create, remove, or modify the definition of a class. A class is defined by a name, optional inheritance, and a set of named attributes and their type. The type of the attributes must be one of the KCSF enumerated types (primitives and sequences). For each attribute, meta-data and a default value can be specified. Modifications to an existing class definition will automatically be applied to all instances of the class. Modifications to existing attribute will only be applied to existing instances if the type of the attribute has been changed. An example of the Class Definition Panel is shown in Figure 22.

6.6.3.2.2 Instances

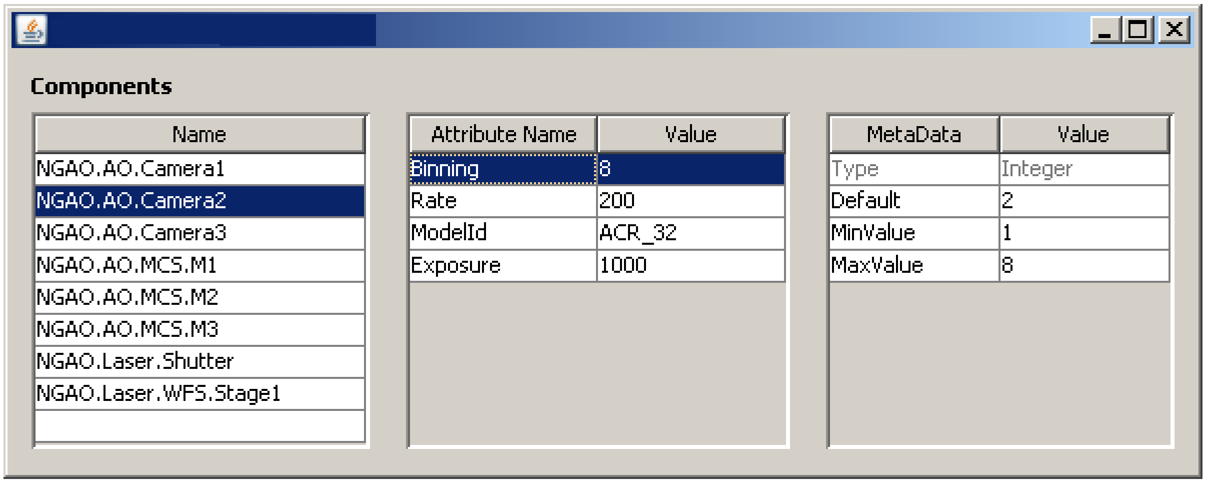
The Instances panel shows all of the object configurations for a project. From this panel a user can add and remove instances from a project, and modify their configuration values. When an instance is added it will automatically be populated with the default values specified in the class definition. Navigating to the object will allow the user to change these defaults and customize the configuration for the application object it models.

6.6.3.2.3 Navigation

There are two modes of navigation separated into distinct views: component view and system view. Component view presents a simple detailed listing of all the objects in the system in an alphabetical organized order. You can search by name for a component and open it up to modify its configuration. System view displays objects as hierarchical tree, showing their relationships to other objects in the project. The tree is built dynamically by inspecting the mappings in the instance tables. (*Note*: System view is only available in the Instances Panel.)

6.6.3.2.3.1 Component View

The Component View Panel presents a simple listing of all the instances defined for a project.



Components

Name
NGAO.AO.Camera1
NGAO.AO.Camera2
NGAO.AO.Camera3
NGAO.AO.MCS.M1
NGAO.AO.MCS.M2
NGAO.AO.MCS.M3
NGAO.Laser.Shutter
NGAO.Laser.WFS.Stage1

Attribute Name	Value
Binning	8
Rate	200
ModelId	ACR_32
Exposure	1000

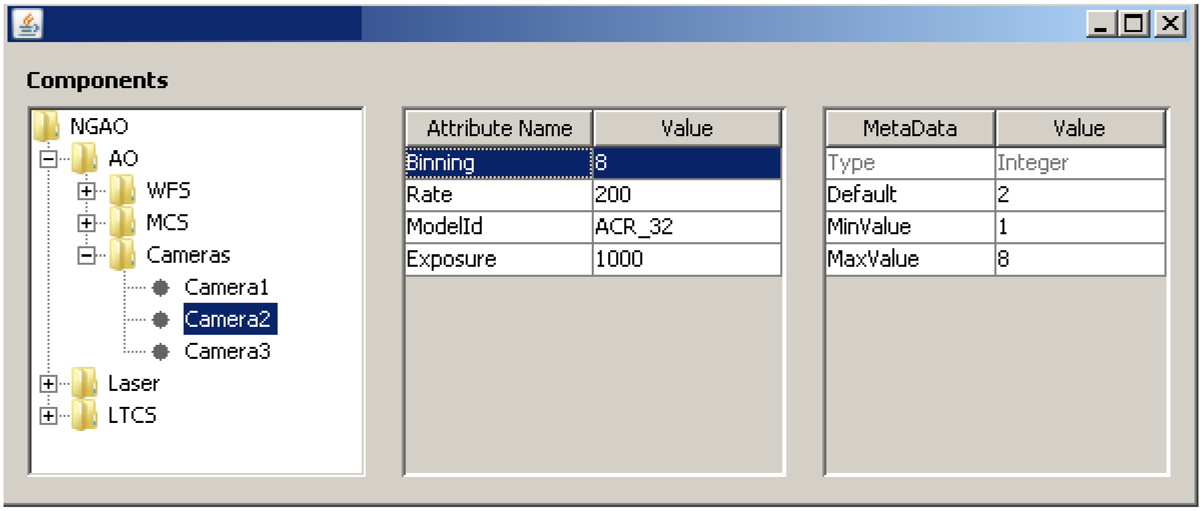
MetaData	Value
Type	Integer
Default	2
MinValue	1
MaxValue	8

Figure 23: Example Component View Panel.

The list can be sorted by type and alphabetized -- opening an item will show its configuration information in a side panel. The advantage of this view is that it can provide immediate access to an object's configuration information. If you know the name of the object you simply scroll to it, and bring up its configuration. An example of the Component View Panel is shown in Figure 23.

6.6.3.2.3.2 System View

The system view allows users to browse objects in a hierarchical fashion, as they would be deployed at run-time.



Attribute Name	Value
Binning	8
Rate	200
ModelId	ACR_32
Exposure	1000

MetaData	Value
Type	Integer
Default	2
MinValue	1
MaxValue	8

Figure 24: Example System View.

This view presents users with an easy to navigate hierarchy of object names. On the left of the screen is a list of all the top-level items (Container Managers). These items can be expanded to show a tree of all their immediate sub-items (Containers). In turn each of these items can be expanded to show their sub-items (Components). Clicking on any of the objects in the tree will present the associated configuration information in a side window. You can also add or remove objects from the hierarchy by clicking on an item and selecting the desired operation. This view is ideal for visualizing and modeling the system lay out, and makes it easier for traversing and finding relationships between components. An example of the System View Panel is shown in Figure 24.

6.7 Scripting

The framework provides support for executing scripts. At the current time, two languages are supported for scripts: Java and Python. Scripts can interact directly with devices and controllers by calling their methods (`get`, `set`, `execute`, `addMonitor`). Additionally, scripts have full access to the framework services and tools. A user does not have to do anything special to use scripts, other than prepare and submit them. Scripts can access component methods, can access and extend KCSF classes, can use generic Java/Python classes and have full access to the Java runtime environments. This includes full access to the KCSF services. The most common usage of scripts will be through the main subsystem sequencers. A standalone tool will also be provided for running scripts. At a minimum, a console based application will be provided that will allow pre-written scripts to be executed within the KCSF environment. Additionally more UI intensive applications can be made available allowing scripts to be edited and executed in place.

An example of a simple script is shown here:

```
TCS = “k1.tcs”;
Tcs = App.connect(TCS);
tcsInput = App.connect(“k1.tcs.axe”);

curRa = tcsInput.get(“ra”);
curDec = tcsInput.get(“dec”);

newLoc = new AttributeList();
newLoc.insert(new Attribute(“action”, “offset”);
newLoc.insert(new Attribute(“ra”, curRa + 0.5;);
newLoc.insert(new Attribute(“dec”, curDec + 0.5;);
tcs.set(newLoc);
App.disconnect(“k1.tcs.axe”);
App.disconnect(TCS);
```

Additionally, more traditional scripting can be supported through the shell and the use of show and modify when used with the KCSF CA Server.

6.8 Tasks

At the lowest level of the Keck Common Services Framework are controllers and devices. These distributed software objects interface directly with hardware and other systems. A number of these devices and systems will need to work in concert to configure and prepare the system for observing, and during observing. Tasks provide application developers with the means to control and coordinate devices at varying levels of granularity through a common interface. Tasks offer developers the flexibility to build layers of increasingly complex functionality from simpler self contained operations. Tasks of all levels present a simple uniform interface to the application developer, effectively abstracting away the underlying functionally and device complexity. Tasks are designed to provide a scalable solution to controls development through loose coupling, reusability and consistent use of the command design pattern.

6.8.1 Approach

All tasks implement the command pattern, where a command object encapsulates an action and its parameters. Objects provide a convenient temporary storage for procedural parameters and can allow a user to assemble a command some time before it is actually needed. A cornerstone of the command design pattern is that all tasks must implement a common control interface. This not only makes tasks appear generic from the point of view of the execution environment, but allows developers to build a hierarchy of nested tasks. The task library framework provides a base task class that abstracts services for processing status and other common infrastructure activities.

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

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

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

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

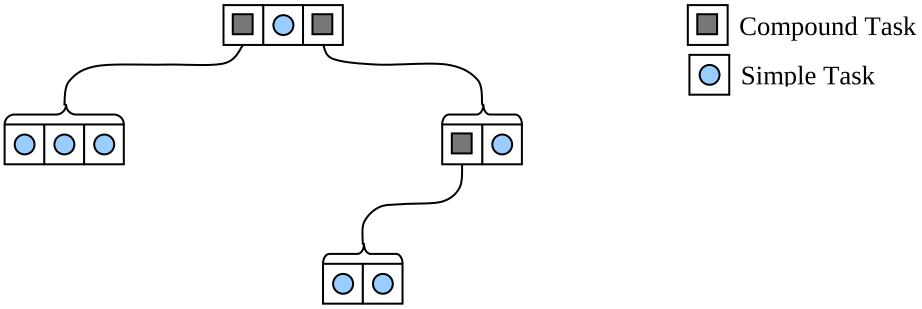


Figure 25: Nested Tasks.

6.8.2 Task Interface

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

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

   protected IAttributeList execute();
}

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

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

The following methods are defined for the Task base class:

- `Initialize(parent)`: this method is used to initialize a task from the parent task (if any), immediately prior to execution. Because a task can be instantiated arbitrarily long before it is actually started, this method performs any dynamic initialization needed just prior to execution, based on current conditions. It is also used to reinitialize a task object if it is reused. Normally this is an inherited method that performs some initialization in the task infrastructure and does not need to be implemented or overridden by the subclass.
- `start`: a method of no parameters that starts the task executing that must return quickly (the intention being that it does not perform the task, but merely initiates it). The mechanics of how this works is dependent on the implementation of the execution environment.

- `wait(timeout=None)`: a method with an optional timeout parameter. This waits for the executing task to finish and returns that task's result. If a timeout is passed a parameter (a float) then the caller will wait at most timeout seconds for the task to finish. If the task does not finish by that time a `TimeoutError` exception is raised. If no timeout is passed, then the caller will wait indefinitely for the task to finish. If an exception is raised by the child task, it will be re-raised in the parent on a wait.
- `run`: this is essentially a convenience function and as a combination of `start` and `wait`.
- `stop`: halt and cancel a task. The implementation of this method is optional and may not be appropriate for all tasks.
- `pause`: temporarily interrupt an executing task. The implementation of this method is optional and may not be appropriate for all tasks.
- `resume`: continue a paused task. Any task that implements the *pause* method must also implement *resume*.
- `registerCallback(callback)`: allows the user to define a callback object that will receive status information when the task completes. (See `ITaskCallback` interface below for more information on the callback signature.)
- `execute`: implements the task logic. This method is executed by the thread pool after the task has been started. The returned `AttributeList` should indicate the success or failure of the operation, and will automatically be forwarded to the `done` method for processing.
- `done(result)`: when a task is ready to terminate normally, it must call this method internally with its result value (defined in an `AttributeList`). A task normally calls this as its final act. The method itself is usually inherited from the parent class.

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

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

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

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

   public void step();
}
```

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

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

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

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

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

6.8.3 Executors

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

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

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

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

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

6.8.4 Using Tasks

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

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

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

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

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

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

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

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

In addition to simple tasks, nested tasks can be created using the task library's implementation of the compound tasks Sequential and Concurrent.

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

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

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

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

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

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

Since all task implementations share the generic interface and follow the command pattern we can compose compound tasks out of sequential and concurrent tasks. In this example a concurrent and sequential task are created, and appended to a list. A new sequential task is instantiated using said list as the task sequence. When ‘s2’ is executed it will perform the standard initialize-start-wait command sequence on each of the compound tasks, recursively executing the underlying simple tasks contained within each. The final return value is the overall status of all tasks within the hierarchy of commands.

6.9 Sequencers

Sufficiently complex systems built with the KCSF, such as the Next Generation Adaptive Optics System (NGAO), will comprise dozens or hundreds of devices and systems. During normal operation these components will need to be coordinated and managed throughout the distributed control system to perform their required tasks. Users and developers of KCSF systems will need a means of efficiently building and coordinating the command logic between these devices in an organized and repeatable way. The solution will need to manage many different types of concurrently executing commands, provide common mechanisms for control and synchronization, and be able to handle many disparate low-level interfaces. The KCSF Sequencer design has been developed to satisfy these requirements.

6.9.1 Approach

Sequencers are implemented as a state-driven KCSF Controller with added functionality for command execution and management. As controllers, sequencers are capable of receiving commands, sending responses, pushing events, and triggering alarms. As with other types of KCSF components, a sequencer instance is defined by a unique name within the system. Sequencers will implement the standard get, set, execute controller interface, which will be used to command the sequencer and issue state transitions.

An additional benefit to developing sequencers as controllers is that tasks can be used to allow one sequencer to command another. In this way, a hierarchy of sequencers can be created allowing the developer to organize sequencers within domains, all of which can be commanded by a single high-level multi-system command sequencer.

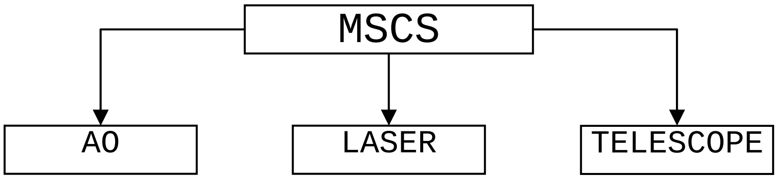


Figure 26: Multi-System Command Sequencer.

States are used to organize a set of related tasks to perform a single coordinated control sequence (for example, acquiring a target). Operators issue transitions to sequencer instances through the

KCSF middleware as they would execute actions on standard device controllers. The developer is responsible for defining the states and transitions for a sequencer, and assigning the tasks that will be executed within each state.

The Sequencer's main system control functionality comes from task implementations defined in the task library. Sequencers are responsible for providing the execution environment for the tasks they will use. Typically a sequencer will utilize an existing lifecycle management object to perform task scheduling and execution. Alternatively a custom executor can be created to provide unique task management if required by the sequencer design. As most sequencers rely on immediate execution of tasks (as opposed to scheduled), many of which contain parallel command processing, the Task Library Command Thread-pool Executor is an ideal management mechanism for sequencers. This executor implements a queuing thread-pool, which allows the sequencer to issue multiple commands simultaneously, as well as utilize concurrent tasks defined in the Library.

Although a task's functional requirements tend to be static after development, it is not unusual for a sequencer's requirements and control flow to evolve as the system matures, and operators' understanding of it improves. As such, the Sequencer design offers developers and users with the ability to modify the execution of a sequencer without making direct modifications to the sequencer code (this can even be done dynamically at run time). The ability to modify a sequencer's task control is provided through the KCSF Script Engine. This utility enables the loading and execution of external scripts which can be written in a number of different programming languages. Developer's can substitute state task(s) with scripting, allowing users to modify the execution logic as needed.

The following diagram details the basic Sequencer design concept and relationships.

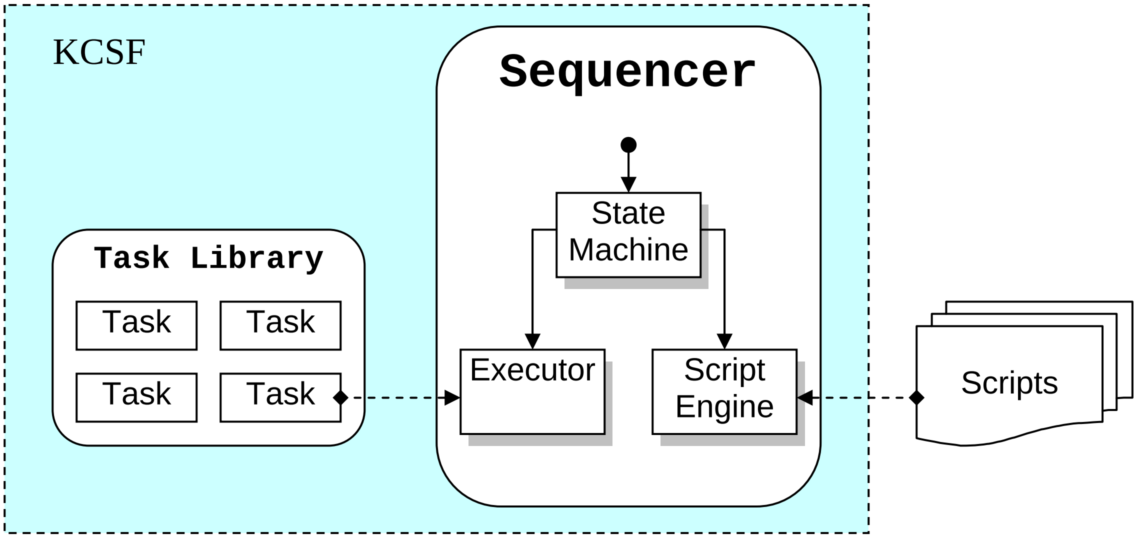


Figure 27: Sequencer Design.

6.9.2 State Machine

Typically, sequencers will implement a well defined state machine to control the execution of steps in an observing sequence. The state machine design pattern does this through the use of *states* and *transitions*. States represent the current configuration of the sequencer and transitions represent the valid paths that can be taken to a new configuration. In a state diagram, states are usually depicted as circles and transitions as arrows, both of which have an associated name.

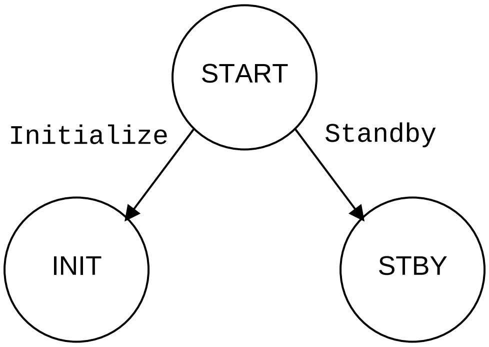


Figure 28: State Machine Diagram

The various states and transitions a sequencer will implement must be determined during the design phase of the sequencer, and are based on the intended functionality of the sequencer. It is within the transition process that the business logic of the sequencer is executed and tasks are performed.

6.9.3 Handling Transitions

The business logic of a Sequencer is executed during state transitions. A developer can implement the functional requirements of a transition in a number of ways. The Framework provided two convenient infrastructure options that can make Sequencer development easier: Tasks and Scripts.

6.9.3.1 Using Tasks

Tasks are an ideal solution for implementing sequencer control logic because they provide a simple, scalable, and composable method to build complex command sequences, and are publicly available to all users of the KCSF. The Task module will contain a number of implementations from simple tasks that control a single device, to complex nested tasks that can command an entire system. Usually a task will be designed to perform a single logical function (e.g., positioning a stage or acquiring a target with the AO). Compound tasks are formed by the combination of simple tasks with sequential and concurrent meta-tasks. The sequencer developer will use these components to build a command sequence to be executed during a state transition.

6.9.3.2 Using Scripts

As an alternative to directly defining tasks, the framework provides sequencer users with the flexibility to modify an observing sequence without impacting the sequencer code itself. This is done through the KCSF Script Engine. This utility allows developers to load and process external scripts used to control the execution of state tasks dynamically (even at runtime).

The business logic of a sequencer exists in the implementation of the state machine transition callbacks. Up till now we have discussed executing tasks during these callbacks to perform the system control. However an alternate solution (or to be used in conjunction with tasks) is to have the callback load and execute script(s) defined by the developer, and specified in configuration.

```
public void initializeTask() {
   script = ScriptEngine.load(this.pathToInitScript);
   script.execute();
}
```

The script is responsible for providing the functionality for the transitions. Since scripts executed through the script engine are able to use the KCSF services and can be given access to class members, a script developer can utilize any of the tasks and functionality available to the sequencer itself. Operators can fine tune script properties as the sequencer is running, or can modify the entire sequence if a better solution is discovered.

6.9.4 Issuing Transitions

Clients can connect to a sequencer instance by obtaining a proxy reference from the connection service. It is the responsibility of the sequencer's parent container to perform the initialization of the sequencer to prepare it for commanding. Once the client has a valid proxy they can issue transitions to the sequencer. Transitions are issued through the sequencer's `execute` method by configuring a `CommandSet` with the required transition information. Each transition may have its own unique parameters so clients must be aware of what each transition requires.

```
CommandSet command = new CommandSet(“Acquire”);
command.set(“TargetName”, “HD13089”);

if(!TelescopeSequencer.execute(command, null)) {
   // Failed to issue command. Reconnect to Sequencer?
}
```

In this example a transition request is created to acquire a target with the telescope sequencer. For this transition only one argument is required, *TargetName*. This argument is used to lookup target information (such as coordinates) in a star catalog. The command is then sent to the sequencer through the execute method.

The advantage of using the controller `execute` method to dispatch transitions is that it uses the controller thread pool to allow the sequencer to operate asynchronously. This permits a user to interrupt or alter a sequence by issuing a *halt*, *standby*, etc.

6.10 User Interface

User interfaces are separated from control operations in the architecture. Components provide a common control surface that is used by Graphical User Interfaces, scripts, and programmatic access. These control surfaces are available through the software bus. This allows UI applications to be distributed independently of the location of the underlying control components. Further, since all user interfaces (GUI, script, programmatic) operate through the same control surface, any form of user interface can be used to access any control surface. This means that simple UIs may be produced quickly to aid in development and safely replaced with more sophisticated UIs later. Status information is also provided through common interfaces across the software bus allowing similar flexibility in UI display of status information.

User interfaces will be composable. User interfaces are implemented as classes (or the equivalent) and can be combined to form UI applications. This allows sophisticated UIs to be built quickly from simpler base UI components. Engineering interfaces, while the responsibility of system developers, are implemented using the APIs and libraries provided by framework.

Since each component is self describing it will be possible to produce a set of data driven UI widgets that can be used to easily monitor a component or graph a value in an ad-hoc manner

- UI starts and is given a component name
- Connects to component and asks for a list of all its attributes
- Displays attribute names in a table view and starts a monitor connection to them
- Displays data as it becomes available.

We are considering several different UI tools for use with KCSF:

- The ATST CSF is bundled with a UI builder.
- GTK+, an open source GUI toolkit:
    - Cross platform support: Linux, Unix, Windows, Mac OS X
    - Used for the GNOME desktop and GIMP image processing program
    - Licensed under the GNU LGLP 2.1
- Qt, a popular UI framework:
    - Cross platform support: Windows, Mac OS X, Linux/X11 (includes Solaris).
    - Available with commercial and LPGL licenses
    - Wide commercial and academic use
- Java Swing

The user interfaces shown in this document are mock-ups and are simple proof of concepts. The actual look and feel should not be interpreted as being the final or even proposed solution. The screenshots are presented to show potential functionality only.

6.11 Developing components

This section describes what a developer must do to design and implement new device and controller components. Most of the bookkeeping work is handled by the base class methods, allowing the developer to concentrate on the application specific behavior of the component. The clean separation of the technical and functional aspects of component management is achieved using delegate methods. The technical tasks are implemented in the base class methods, which then call the delegate methods to complete the functional tasks. Method stubs which do nothing are already provided, so the developer need only override the delegate methods required to implement the desired functionality. The delegate methods have the same name as their base class method counterparts, with the prefix “perform” attached and the entire method name conforming to camelCase notation (e.g., `performGet`, `performSet`, etc.).

The component design process begins with the developer extending the `DeviceCore` or `ControllerCore` base class to implement a new device or controller type (the `DeviceCore` and

`ControllerCore` classes are discussed in more detail in the section on the architecture developer view). Next, the lifecycle delegate methods are overridden as required. Recall that the component lifecycle consists of the creation, initialization, startup, operation, shutdown, uninitialization and removal phases. The lifecycle of a component is managed using state: lifecycle methods all check to see if the current component state is appropriate before continuing. If the current state is not appropriate, the base class lifecycle method will exit without performing any of its tasks and without calling the delegate method. The component lifecycle state transition diagram is shown in Figure 29. We discuss each lifecycle phase below, summarizing the actions implemented by the base class methods and the actions that are left to be implemented in the delegate methods.

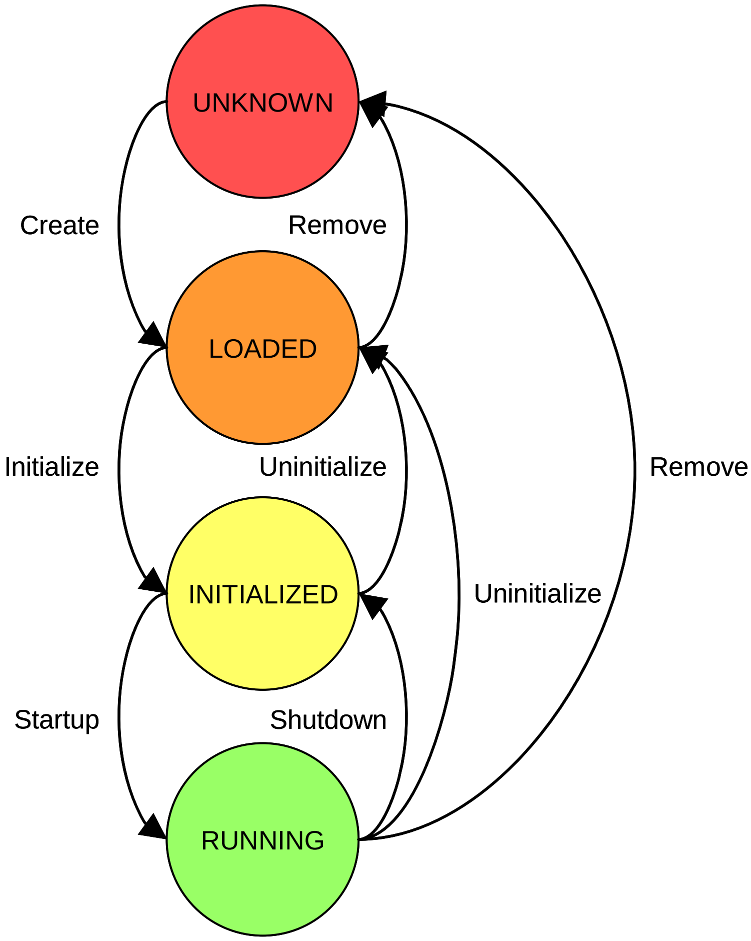


Figure 29: The component lifecycle state transition diagram.

6.11.1 Lifecycle methods

6.11.1.1 Creation

Prior to creation, the component's state is `UNKNOWN`. During creation, the component is created by its parent container and connected to its available services. There are no delegate methods to be written by the application developer. At the end of creation, the component state is set to `LOADED`.

6.11.1.2 Initialization

The component must be in the `LOADED` state in order to be initialized. During initialization, the base class method creates any local objects required to manage the component and then calls the configuration service. The configuration service returns an attribute list which is then applied to the component using its own set method. Note that the `performSet` method, which is called by `set` (which will be discussed in more detail below), is aware of the lifecycle state of the component and can respond accordingly. This means that parameters specific to the initialization can easily be passed to the component, placed in local storage, and then acted upon later when the delegate initialization method is called. The application developer should implement a `performInitialize` delegate method that uses the configuration information as needed, creates any desired local buffers, connects to any required device drivers, and performs any initialization steps required by the physical or virtual device represented by the component. If the delegate method completes successfully, the component state is set to `INITIALIZED`.

6.11.1.3 Startup

The component must be in the `INITIALIZED` state in order to be started. The base class then calls the `performStartup` delegate method to perform any local startup tasks. The developer should implement a `performStartup` method to perform any startup tasks required by the physical or virtual device represented by the component. When finished, `performStartup` should leave its device in its operational state. Once the delegate method has completed successfully, the component state is set to `RUNNING`.

6.11.1.4 Operation

This is the main functional state of the component (`RUNNING`). It responds to user requests using the main functional interface described earlier (`get`, `set`, `execute`, and `xxxMonitor`). There are no delegate methods for this lifecycle state.

6.11.1.5 Shutdown

The component must be in the `RUNNING` state in order to be shutdown. During shutdown, the component releases all external resources it is using. The delegate `performShutdown` method should stop all device activity and leave the physical or virtual device in a safe state upon completion. Once the delegate method has completed successfully, the component state is set to `INITIALIZED`.

6.11.1.6 Uninitialization

The component must be in the `INITIALIZED` or `RUNNING` states in order to be uninitialized. If the component is in the `RUNNING` state, the `shutdown` method will be called before the body of the `uninitialization` method is executed. The `performUninitialize` delegate method should perform any required steps to uninitialize the device, disconnect from any device drivers, and release all local resources. Once the delegate method has completed successfully, the component state is set to `LOADED`.

6.11.1.7 Remove

The component must be in the `LOADED`, `INITIALIZED`, or `RUNNING` state to in order to be removed. If the component is in the `RUNNING` state, the `shutdown` and `uninitialize` methods will be called before the body of the remove method is executed. If the component is in the `INITIALIZED` state, the `uninitialize` method will be called before the body of the `remove` method is executed. During removal, the component ceases all activity and releases all its internal resources. The `performRemove` delegate method should free any remaining local resources and perform any final cleanup required before the component is deleted. The component ceases to exist upon completion of the `removal` method and the state is set to `UNKNOWN`.

6.11.2 Functional methods

Once the lifecycle delegate methods have been completed, the application developer must implement the main functional methods of the component. These are the methods that determine the component's actual behavior in the system. As was done for the lifecycle methods, stubs of the delegate methods which do nothing are provided and must be overridden by the developer. The delegate methods to be implemented are: `performGet`, `performSet`, `performExecute`, `performPause`, `performResume`, and `performCancel`.

6.11.2.1 Get

The base class `get` method performs validation on the input attribute list and passes a validated attribute list to the `performGet` method. The application developer should do the following in the `performGet` method:

- Create a copy of the attribute list with nulls in the value fields to be used as a starting point for the list to be returned.
- Iterate through the list of attributes and take actions as required based on the attribute name:
    - Read the value corresponding to the attribute from the physical or virtual device.
    - If an error occurs, write to the appropriate error log.
    - Add the value to the appropriate attribute in the return attribute list. A null value should be used to indicate a failure.
- Call the `getResult` method to indicate to the base class the overall status of the get operation ("*Success*", "*Partial*", "*Fail*").
- Return the completed attribute list. The base class will automatically append the "*_OperationResult*" attribute as a result of the `getResult` call made above.

6.11.2.2 Set

The base class `set` method performs validation on the input attribute list and passes a validated attribute list to the `performSet` method. The application developer should do the following in the `performSet` method:

- Create a copy of the attribute list with nulls in the value fields to be used as a starting point for the list to be returned.
- Iterate through the list of attributes and take actions as required based on the attribute name:
    - Write the value of the attribute to the corresponding physical or virtual device parameter.
    - If an error occurs, write to the appropriate error log.

    - Determine the actual value of the attribute accepted by the device (e.g., account for any clipping due to the parameter being out of range).
    - Add the actual value to the return attribute list. A null value should be used to indicate a failure.
- Call the `setResult` method to indicate to the base class the overall status of the get operation ("*Success*", "*Partial*", "*Fail*").
- Return the completed attribute list. The base class will automatically append the "*_OperationResult*" attribute as a result of the `setResult` call made above.

6.11.2.3 Execute

The base class `execute` method performs validation on the input command set and passes a validated command set to the `performExecute` method. The application developer should do the following in the `performExecute` method:

- Determine the action required by finding the "*_Action*" attribute keyword. The base class execute method has already verified that the "*_Action*" keyword exists; the delegate method just needs to find its value. Note: The `get` helper method on the attribute list in the command set is used to return the "*_Action*" attribute, and the `get<Type>` helper method is used on the returned attribute to extract the keyword value.
- Execute the appropriate code based on the value of the action keyword. This code may be fairly complicated, involving a number of steps and may require extracting additional values from the input attribute list as parameters.
- Call the `executeResult` method to indicate to the base class the overall status of the execute operation ("*Success*", "*Partial*", "*Fail*").
- Return an updated command list. The base class will automatically append the "*_OperationResult*" attribute as a result of the `setResult` call made above.
- During the processing described above, the `execute` method must check to see if a `pause` has been issued for this command. If so, the `execute` method must gracefully halt its activities and sleep until awaken by a `resume`.
- Similarly, the `execute` method must check to see if a `cancel` has been issued for this command. If so, the `execute` method must gracefully cancel all its activities and return. The specific clean-up behavior is left up to the developer.

6.11.2.4 Pause

The `pause` command is used to pause an `execute` command that is currently in progress. The developer should use the `performPause` method to implement any necessary actions to place the physical or virtual device in a safe state until a `resume` command has been received.

6.11.2.5 Resume

The `resume` command is used to awaken an `execute` command that has been previously paused. The developer should use the performResume method to implement any specific recovery actions that should be taken and then resume the command that was previously paused.

6.11.2.6 Cancel

The `cancel` command is used to cancel an `execute` command that is currently in progress. The developer should use the `performCancel` method to implement any necessary actions to place the physical or virtual device in a known safe state and then cease all activities related to the original `execute` command.

6.11.3 Examples

This section demonstrates how to extend the base `DeviceCore` or `ControllerCore` classes to implement a new device or controller as described above.

6.11.3.1 Camera Device

This example shows how to extend the `DeviceCore` class to implement a camera device. It shows the overriding of the `performGet`, `performSet` and `performExecute` methods.

```
package kcsf.cs.ccm.component.demos;

import kcsf.cs.ccm.component.DeviceCore;
import kcsf.cs.data.Attribute;
import kcsf.cs.data.AttributeList;
import kcsf.cs.interfaces.IAttributeList;
import kcsf.cs.services.Log;

/*
 * Simple camera device that has the following Attributes:
 * Binning, IntegrationTime
 *
 * It supports 2 commands: Start and Stop
 */
public class CameraDevice extends DeviceCore{

   private int _binning = 2;
   private double _intTime = 1000.0;

   @Override
   public synchronized IAttributeList performGet(IAttributeList values) {

      IAttributeList result = new AttributeList();
      if (values.contains("Binning"))
         result.insert(new Attribute("Binning", _binning));
      if (values.contains("IntegrationTime"))
         result.insert(new Attribute("IntegrationTime", _intTime));
      result.setResult(“Success”);
      return result;
   }

   @Override
   public synchronized IAttributeList performSet(IAttributeList values) {

      IAttributeList result = new AttributeList();

      if (values.contains("Binning")) {
         _binning = values.get("Binning").getInteger();
         result.insert(new Attribute("Binning", _binning));
      }
      if (values.contains("IntegrationTime")) {
         _intTime = values.get("IntegrationTime").getDouble();
         result.insert(new Attribute("IntegrationTime", _intTime));
      }
      result.setResult(“Success”);
      return result;
   }

   @Override
   public synchronized boolean performExecute(IAttributeList values) {

      String action = values.getString("_Action");
      if (action == null) {
         result.setResult(“Failure”);
         return false;
      }

      if (action.equals("Start"))
         return startExposure(values);
      else if (action.equals("Stop"))
         return stopExposure(values);

      Log.note("Unrecognised action for device");
      result.setResult(“Failure”);
      return false;
   }

   private boolean startExposure(IAttributeList values) {
      // check values to see if binning or other options set
      ....
      _cameraDevice.startExposure(_binning, _intTime);
      result.setResult(“Success”);
      return true;
   }

   private boolean stopExposure(IAttributeList values) {
      _cameraDevice.stopExposure();
      result.setResult(“Success”);
      return true;
   }
}
```

6.11.3.2 Camera Controller

This example shows how to extend the ControllerCore class to implement a controller that composes multiple devices. It shows the overriding of the lifecycle methods and the execute command to issue execute commands to multiple cameras on separate threads.

```
package kcsf.cs.ccm.component.demos;

import kcsf.cs.controller.ControllerCore;
import kcsf.cs.interfaces.IAttributeList;
import kcsf.cs.interfaces.ICallback;
import kcsf.cs.interfaces.IDevice;
import kcsf.cs.services.App;
import kcsf.cs.services.Log;

/*
 * Simple composite controller. Coordinates starting and
 * stopping a number of different cameras
 */
public class CameraController extends ControllerCore {

   private String[] _cameraIds = null;
   private IDevice[] _cameras = null;
   //
   // Expects a list of cameras to be coordinated
   //
   @Override
   public void performInitialize(IAttributeList values) {
      _cameraIds = values.getNames();
      _cameras = new IDevice[_cameraIds.length];
   }
   @Override
   public void performStartup(IAttributeList values) {
      for (int i = 0; i < _cameraIds.length; i++) {
         _cameras[i] = (IDevice) App.connect(_cameraIds[i]);
      }
   }

   @Override
   public void performUnInitialize(IAttributeList values) {
      for (int i = 0; i < _cameraIds.length; i++) {
         App.disconnect(_cameraIds[i]);
      }
   }
   @Override
   public boolean performExecute(IAttributeList values) {
      String action = values.getString("_Action");
      if (action == null) return false;

      if (action.equals("Start")) {
         for (int i = 0; i < _cameras.length; i++) {
            Log.note("Starting camera " + _cameraIds[i]);
            _cameras[i].execute(values, null);
         }
      }
      else  if (action.equals("Stop")) {
         for (int i = 0; i < _cameras.length; i++) {
            Log.note("Stopping camera " + _cameraIds[i]);
            _cameras[i].execute(values, null);
         }
      }
   }
}
```

7 Developer View

The Keck Common Services Framework (KCSF) is based on the ATST Common Services Framework (CSF). They both share the same component based development paradigm, the same container component model, tiered architecture, communication middleware neutrality and basic libraries. The framework provides a standardized environment to applications to aid in the development of control systems. The following provides a high level overview.

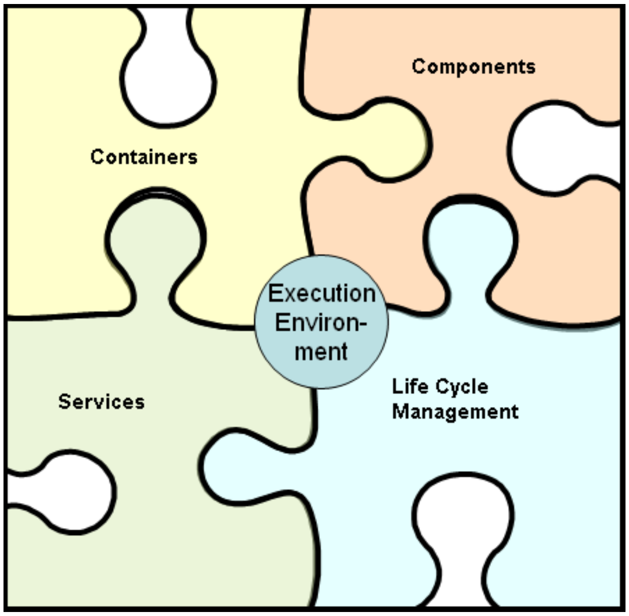


Figure 30: KCSF Overview.

The framework brings together containers, components, services and tools and provides life cycle management for these when they are deployed in an execution environment. The execution environment will be provided by the OS or in the case of Java by the Java virtual machine. The expected JVM will be the J2SE profile. This framework concept is illustrated in Figure 30.

Containers are responsible for housing components and for the class loading policies. Containers add modularization and provide the ability to dynamically load and unload components as needed as well as provide services to the components. Services are a set of basic functions that are available to all components, such as connecting to other components, logging parameter changes, posting events and configuration. Life cycle management is accomplished with a container manager. Components can be dynamically installed, uninstalled, started, stopped, configured and updated. Lifecycle management introduces dynamics that are normally not part of an application. Extensive dependency mechanisms are used to assure the correct operation of the environment.

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

Please see KAON 671: Keck Next Generation Adaptive Optics Container Component Model for a full discussion of the model.

7.1 Architecture Layers Details

The Common Software is a tiered architecture as described below and shown in Figure 31.

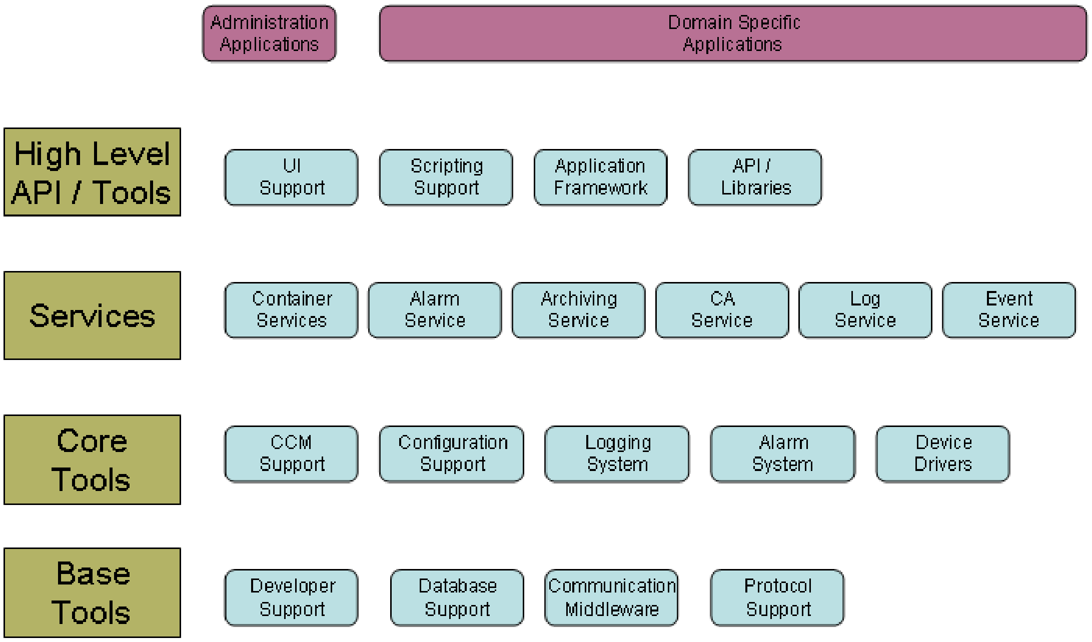


Figure 31: Detailed Architectural Layer.

- High-level APIs and Tools : This level of software directly supports the development of KCSF applications and is the level presenting the Common Software functionality to application developers.

- Services : The services layer consists of software that implements mid-level KCSF functionality based on the core tools.

- Core tools: The core tools build low-level KCSF required functionality directly from the base tools. Core tools often have performance constraints that mandate direct access to the foundation software.

- Base tools: The lowest level contains software that is independent from the actual KCSF functionality. It is support software on which KCSF aware software is based. Most of the software at the base level is COTS or open source and is not directly maintained by KCSF. It consists of:
    - Development tools include IDEs, languages and compilers, versioning systems, debuggers, profilers, and documentation generators.
    - Database support includes software on which one can implement persistent stores, engineering archives, and science header and data repositories.

- The communications middleware is the foundation for all inter-process communication in KCSF. It provides the communications bus, location utilities (name services), and a robust notification system.

7.2 Container Manager

The container manager is a component responsible for deploying and initializing containers and their components through dynamic process creation and dependency injection. Container managers have access to configuration information that defines the number and types of containers and their specific components and services. They are capable of deploying containers on different nodes, attaching to already running containers and managing the lifecycle of containers.

A container manager has the following API, which provides the capability to deploy containers, add a container that is already running to the manager, remove containers and get a list of all deployed containers for that manager.

```
public String deployContainer(String name, String host);
public String deployContainer(String name, String host, String type);
public String addContainer(String name);
public String delContainer(String name);
public String[] getAllContainers();
```

To deploy a container the container manager will execute a (possibly remote) script to call the main entry point for a container. Using the connection service the manager then connects to the container to verify it is operational. The connection service is used to send lifecycle commands to the container as needed.

7.3 Container

In KCSF, containers are part of the technical architecture and have the following responsibilities:

- Manage component lifecycles, including creating, starting, stopping, and destroying components. This means that there is a uniform mechanism for controlling component lifecycle operations. Multiple components may run under the same container
- Provide services to the components. Because services are shared amongst the components in a container, more efficient use of services is possible.

The container hides the details of the framework implementation from the component, and makes it possible for the component to be used in multiple modes of execution. Containers are multithreaded and can simultaneously manage multiple components, or can be limited to a single component as needed.

Containers are language specific: only a single language is supported by a container. Multiple containers are required to support more than one language. Although there are no technical prohibitions against doing so, KCSF generally does not run more than one type of language-specific container on a given host.

A number of classes are used to implement a container. A container uses a `ComponentInfo` class to record meta-data about the loaded components, a `ComponentLoader` to do the actual class and service loading, a `ToolBox` to hold and manage services and a `ComponentClassLoader` to perform the actual component instantiation. A `ContainerManager` is a special component that is used to deploy containers. These classes are discussed in further detail below, and their class hierarchy is shown in Figure 32.

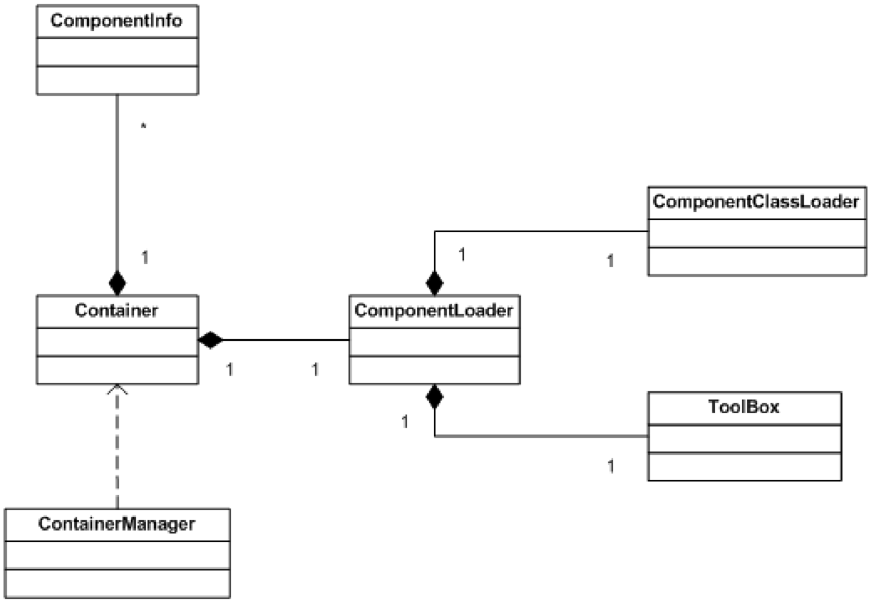


Figure 32: Container class decomposition

The container's lifecycle control module is responsible for managing the component lifecycles. It responds to external requests on the container to create, start, stop, and destroy specific components. The component loader handles these external requests for the container, calling on the component class loader to create each component. External requests generally come from container managers or sub-system administration applications.

The component loader works by establishing a new namespace for the component and then creating the component and a toolbox for that component within that new namespace. The component loader then instructs the toolbox loader (within the service access control module) to load that toolbox with service tools. Individual tools may be private (existing within the component's namespace) or shared (existing outside the component's namespace). It is the toolbox loader that determines which tools are private and which are shared; different toolbox loaders may make different determinations. This concept is illustrated in Figure 33.

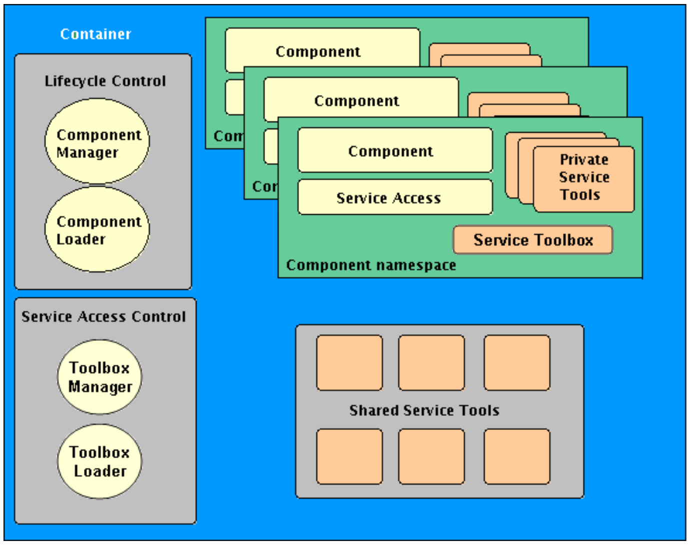


Figure 33: Typical Container

An important point is that the container, through the toolbox manager module, retains access to each component's toolbox - and hence to all the service tools. This allows a container to dynamically adjust service properties on a per-component basis.

Note: In the ATST CSF each component operates in its own namespace (with the exception of interface definitions which are always shared across all namespaces). This is accomplished by means of a private class loader that is utilized by the container. Since for KCSF we have decided we do not require the ability to dynamically replace classes on the fly, our current design is to use the standard class loader. Should the need for a private class loader arise then we can follow the CSF model.

7.3.1 Container Interface

A container has the following API :

```
public void addComponent(String name, String className);
public void delComponent(String name);
public String[] getAllComponents();
public boolean contains(String name);
public IComponent getComponent(String name);
public IComponentAdmin getComponentAdmin(String name);
public void setLifecycleCondition(String cName, String newStatus);
public void component(String cName, String action, IAttributeList
  args);
```

7.3.1.1 addComponent

Given a component name and the full class path this method will create an instance of the component and associate it with the requested name. It first checks to see if the instance is already loaded. The container itself does not directly load the class but uses its `componentLoader` member class. The `componentLoader` can be changed as needed to support separate namespaces, class load order and so forth. After this call, if successful, a component will be loaded but not initialized.

7.3.1.2 delComponent

If the requested component is loaded it will be unloaded and removed from the container. In the current design, if the component is running, it is shutdown first and then removed. A log message is generated if the component is shutdown. Through its lifecycle methods, the component is given the opportunity to free any resources and close any connections before removal. Components are removed on a separate thread.

7.3.1.3 component

This call is used to control a component's lifecycle. For parameters it accepts a lifecycle action and any additional parameters to be applied to that action. The following actions are supported:

- Initialize
- Startup
- Shutdown
- Uninitialize
- Remove
- CheckHealth

7.3.2 ToolBox

A toolbox provides an application with access to all essential KCSF services. When a component is created by a container, the container attaches a toolbox filled with service helpers to the component. This is a singleton class so (a) all component objects in a container see the same instance and (b) it implements the `IToolBox` interface so the `ContainerManager` can still manipulate it.

The toolbox also holds all component-specific knowledge that is needed by the common services. Service helpers can access this information as needed to perform various tasks. Component subclass access to this information is through the convenience classes mentioned below. The toolbox manages the following services: Alarm, Connection, Archive, Event, Health, CA, Logging, and Configuration. Any other service tools are passed on to chained toolboxes (if any) for handling. Service tools are loaded into the toolbox using a toolbox loader. Normally this class is not referenced directly; instead, the convenience classes KCSF.cs.service.[ServiceName] are used. They simply wrap this class, but provide a more mnemonic interface.

7.3.3 Component Info

This class provides simple storage for information about a component as seen by a container. Items such as the component name, class name and other information are retained.

7.3.4 Component Loader

The `ComponentLoader` class offloads from the container the work of installing a component into the container. This separation allows for the ability to locate the class for the component and instantiate it into a separate namespace. A toolbox is then built and populated with service tools. The toolbox knows its owner component and is responsible for handling all service requests from that component. The class loader dynamically locates and creates classes, their resources and any dependencies.

The steps involved in loading a component are as follows:

- Create a toolbox in the component's namespace.
- Assign the toolbox to the container.
- Populate the toolbox with services.
- Start the services.
- Use the class loader to create the component (the default constructor is called).
- Give container hooks to component and toolbox and assign the toolbox to the component.
- Store information about that component into the `ComponentInfo` record.

7.4 Components: Devices and Controllers

Components are the basic building blocks of a control system and are broken down into two basic types, Devices and Controllers, which form the backbone of the system. All components must implement the `ITechnical` interface to conform to the container component model. These are the methods needed for lifecycle control of components, and represent the technical portion of the component interface. In addition, components must implement the `IRemote` interface to support the basic set and get methods, which are the functional portion of the component interface. These interfaces are shown below in Figure 34.

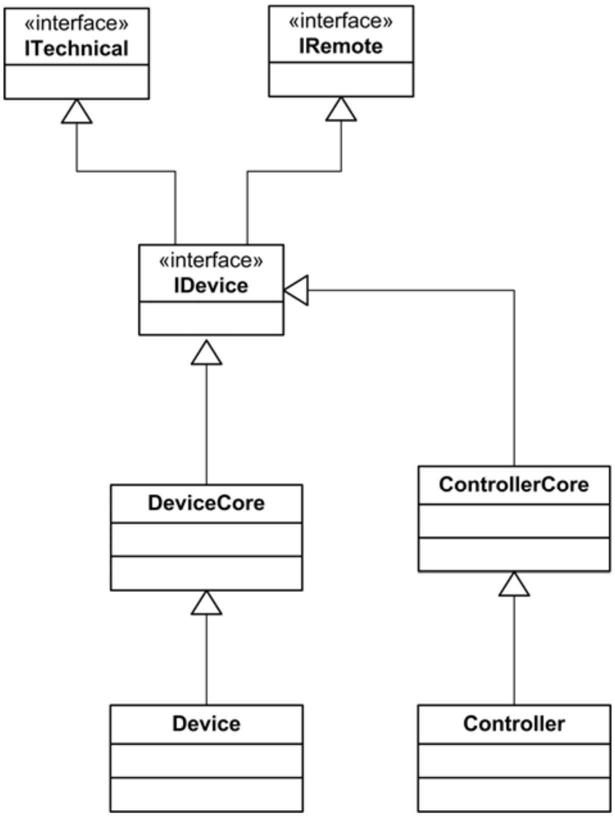


Figure 34: Components: Devices and Controllers.

KCSF provides further extensions of these interfaces through the `IDevice` interface to implement components called `Device` and `Controller`. These components provide additional capabilities such as the ability to execute commands and to monitor attributes. A Controller is a special case of a Device that is capable of supporting multiple simultaneous commands. Abstract classes called `DeviceCore` and `ControllerCore` implement the business logic for these components, providing the following:

- Management of component state
- Basic lifecycle methods
- Implementation of the monitor methods
- Single-threaded vs. multi-threaded implementation of the `execute` command.

The last point emphasizes the main distinction between devices and controllers: controllers are essentially multi-threaded devices.

The functional aspects of component behavior are left to the application developer to be implemented as delegate methods. All the lifecycle methods and the main functional interface methods have delegate functions (e.g., `performInitialize`, `performGet`, etc.). KCSF architecture developers are free to extend these abstract classes as required to best match their particular application. The `DeviceCore` and `ControllerCore` classes are shown here as separate classes, but in the future they may be consolidated into a single class.

The `IDevice` interface is logically comprised of actions and attributes. Actions are implemented through the `execute` method and attributes are accessed through the `get`, `set` and `xxxMonitor` methods. Actions are just that: actions to be performed on a device (e.g., starting a camera exposure), whereas attributes represent the parameter set of a particular device (e.g., the integration time of a sensor).

The `IDevice` interface continues the separation of technical and functional aspects of the component interface that were described earlier. The technical aspect refers to the details of the framework, whereas the functional aspect refers to application specifics. The technical side of the interface supports connections to containers, lifecycle support (initialization, startup, shutdown etc.) and services support. The functional side of the interface supports the domain specific capabilities (e.g. NGAO control). The technical and functional aspects of the device interface discussed in further detail in the following sections and are illustrated in Figure 35 below.

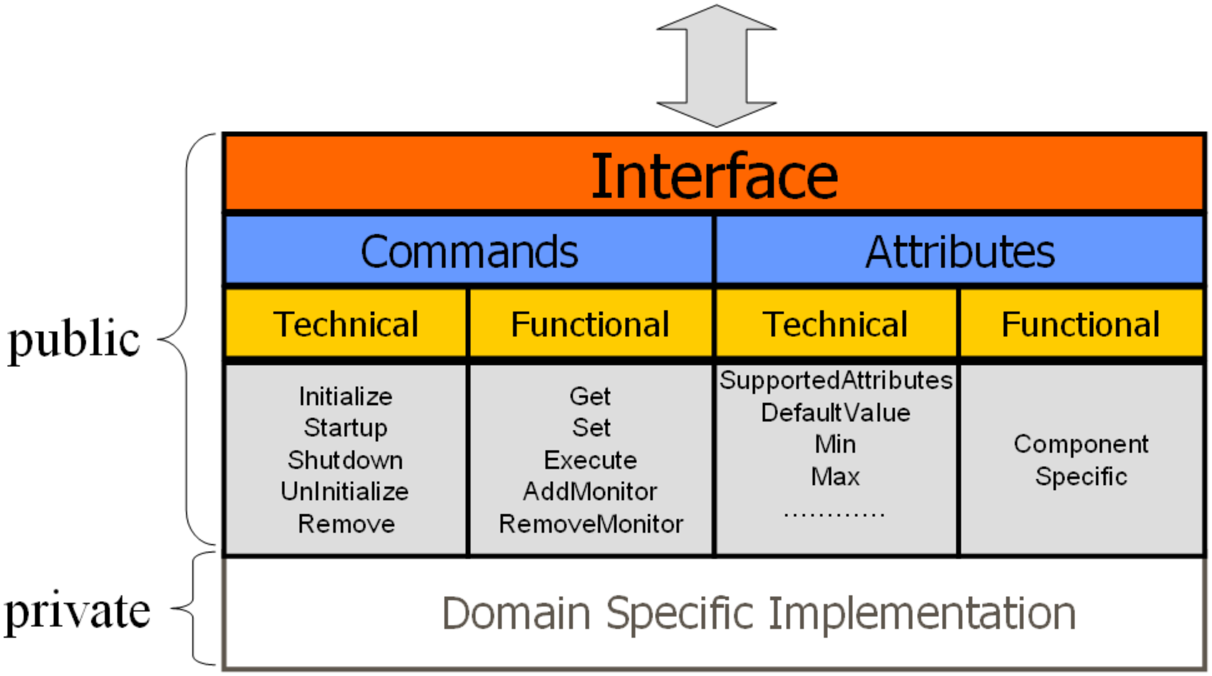


Figure 35: The device interface is composed of commands and attributes.

7.4.1 Technical interface

The device technical interface includes the following methods:

```
public String getName();
public void initialize(IAttributeTable args);
public void startup(IAttributeTable args);
public void shutdown();
public void unInitialize();
public void remove();
public String getLanguage();
public String getHostName();
public String getContainerName();
```

The lifecycle methods `initialize`, `startup`, `shutdown`, `uninitialize`, and `remove` all have stub delegate "perform" methods (e.g., `performStartup`) which should be overridden by the application developer to implement the appropriate device specific lifecycle behavior.

The component lifecycle states were discussed in detail in Section 6.11.1, so only an overview is provided here. Containers create components using the component's default constructor, which should do as little as possible. Initialization activities are implemented in the `initialize` method. At the time initialize is called, the toolbox and a full set of services are available to the component. The base class method calls the configuration service to get the component's configuration data. The delegate method should connect to any device drivers and create any local resources that are needed. The component is put into its operational state by calling the `startup` method. `shutdown` and `uninitialize` are essentially the opposites of `startup` and initialize, placing the component in a stopped state and undoing the initialization tasks, respectively. remove is called before a component is destroyed and allows for resource cleanup.

7.4.2 Functional interface

The device functional interface API was discussed in detail in Section 6.3.1, so only a summary is provided here. The basic interface is implemented through the get, set, execute and xxxMonitor methods. get, set, and execute are implemented in both synchronous and asynchronous forms, the asynchronous forms requiring the use of callbacks. The application specific behavior of the device is achieved using the delegate functions performGet, performSet, and performExecute, which the application developer must implement (the delegate methods are also discussed in Section 6.11.2). The base classes provide stubs of these functions which do nothing and should be overridden to provide the desired application behavior. The API is as follows:

- Get:

```
AttributeList get(AttributeList requestList);
boolean get(AttributeList requestList, ICommandCallback callBack);
```

- Set:

```
AttributeList set(AttributeList setList);
boolean set(AttributeList setList, ICommandCallback callBack);
```

- Execute:

```
boolean execute(CommandSet commandSet, ICommandCallback callBack);
boolean pause(long commandID);
boolean resume(long commandID);
boolean cancel(long commandID);

long getCommandID();
```

- Monitors:

```
boolean addMonitor(String component, IAttributeList attributeList,
  ICommandCallback callback, int rate);
boolean addToMonitor(String component, IAttributeList attributeList);
boolean removeFromMonitor(String component, IAttributeList
  attributeList);
boolean removeMonitor(String component);
boolean refreshMonitor(String component);
```

- Callback methods:

```
AttributeList getComplete(IAttributeList attributeList);
AttributeList setComplete(IAttributeList attributeList);
AttributeList executeComplete(IAttributeList attributeList);
AttributeList monitorTriggered(IAttributeList attributeList);
```

7.5 Commands

There are two basic classes of commands used in KCSF:

- Lifecycle commands. These are commands used by KCSF 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 KCSF infrastructure.
- Functional commands. These are commands that implement the specific functional characteristics of a component. Because the KCSF uses a narrow command interface, the number of APIs to support functional commands is quite small though the implementation can be as rich as a developer needs.

Commands are implemented using two simple paradigms: command/response for synchronous commands, and command/action/response for asynchronous commands. Every command has an associated response which indicates the command completion status. Both paradigms isolate the transmission of the command from the resulting action that is performed.

7.5.1 Middleware Binding for Commands

The framework is designed to be middleware neutral. More than one third-party communication middleware can be bound to the framework. This section describes how the middleware is encapsulated into a simple interface seen by the user.

The `bind` method associates a unique name to a component instance to act as a global address for the object. Although not every server object needs to be addressable, every published component name must map to one and only one component instance. The `connect` method is used by the client to find and open a connection to a component identified by the unique name.

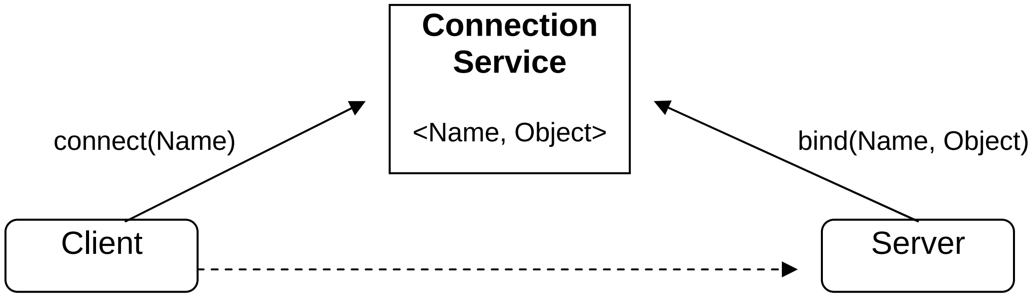


Figure 36: Connection Service Overview.

In order to hide the details of the underlying communication middleware, abstract object wrappers are used to bridge the application layer with the communication protocol. Wrappers are separated into two main types: proxies and stubs. For each proxy implementation there is a corresponding stub implementation, and vice versa. Proxies and stubs expose an interface identical to the component they wrap, and are responsible for translating the data that is sent across the wire.

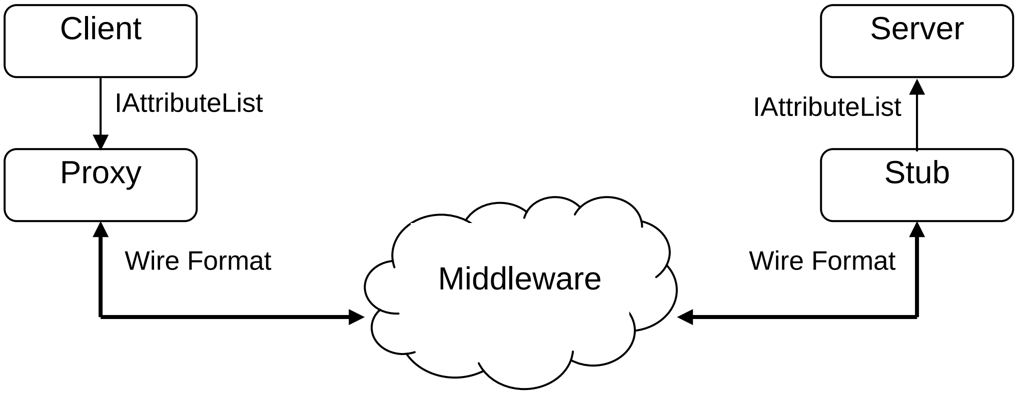


Figure 37: Proxies and Stubs.

On the client side, a proxy is used to represent the existence of a remote object and maintain an open connection to the component. Since the proxy exposes an interface identical to the actual remote object instance, the client can treat it as a local object, without knowledge of the communication protocol or concern for its location. Any data that is passed during a call to one of the proxy's methods is translated to the format required by the communication protocol and sent on the wire. The proxy may then wait for a response from the remote component, and dispatch any callbacks events that occur.

On the server side, a component stub is used to delegate incoming communication requests to the component implementation. The stub performs a mirror opposite form of data translation to that of the proxy, by converting the data received on the wire to the original format expected by the component middleware. The corresponding method on the wrapped component is then executed, passing the data to the instance. If necessary the stub will wait to receive a result status from the component and issue a callback event to the client proxy.

To aid in the creation of proxies and server stubs a set of object factories have been developed known as Connectors. The connection service uses connectors with the `bind` and `connect` operations to determine the type of proxy and stub to incarnate.

7.5.1.1 Connectors

Connectors are used by the connection service to determine the appropriate stub and proxy to use for binding and connecting to components. Essentially, connectors are a conditional proxy/stub factory, where the successful creation of a wrapper is dependent on the type of component. As with proxies and stubs, there is a corresponding connector implementation for each class of object. Specifically, the following connectors will be implemented,

- `IRemoteConnector`
- `IDeviceConnector`
- `IContainerConnector`
- `IContainerManagerConnector`

The actual proxies and stubs created by a connector are related to the class type of the connector. For example, a Controller Connector will only produce Controller proxies and stubs. Connectors expose a very simple interface for creating proxies and stubs:

```
Class IConnector {
   Stub bind(ILocal object);
   Proxy connect(IRemote object);
}
```

In the above pseudo-code, `bind` and `connect` methods are provided that accept an instance of an `ILocal` and `IRemote` base object. All server-side component implementations inherit from the `ILocal` base class. All client-side proxy implementations inherit from the `IRemote` base class, which is itself a subclass of `ILocal`. Both interfaces represent a component level software object that is communication middleware agnostic. The connector is responsible for generating a communication middleware specific wrapper that is capable of accepting and delegating commands between the transport and component.

However, not just any wrapper can be created: it has to be specific to the actual class of component passed to the connector. To this end, the connector will test the `ILocal`/`IRemote` reference to see if it is an instance of the class type implemented by the connector. If it is, a corresponding proxy/stub is created for the class type and returned to the connection service. If the object reference is not an instance of the connector's class type, a null object is returned.

The advantage of using connectors is that proxy and stub creation logic can be removed from the Connection Service and contained within simple object factories. This continues the inversion of control principal promoted throughout the framework. Since connector types are built in an increasing complexity class hierarchy that mirrors the class hierarchy of components (`Remote` ← `Component` ← `Controller`, etc.), the creation of stubs and proxies can be done by simply iterating through a list of connectors and attempting to bind/connect a component until a valid wrapper is returned. Connectors are arranged in the sequence from most complex to least (`ContainerManagerConnector` → `LocalConnector`). The first valid wrapper returned by a connector represents the highest level type of the component. The Connection Service will use the wrapper returned by the connector to bridge the component and communication middleware layers.

7.5.1.2 Proxies

Proxies are the client-side interface to distributed objects, and implement an identical set of methods and class signature as the component they represent. Proxies are designed to give the client the sense that they are directly operating on a local object. Any data marshaling and formatting required by the communication middleware is handled by the proxy opaquely: the client does not need to be aware of how the data is transported.

There are five types of proxies representing each of the main base classes of objects that can be developed with the Keck Component Framework:

- `IRemoteProxy`
- `IDeviceProxy`
- `IContainerProxy`
- `IContainerManagerProxy`

Each of these proxies implements the client-side interface for the `IRemote`, `IDevice`, `IContainer`, and `IContainerManager` classes, respectively. Proxies may also implement additional methods outside of those defined by their associated component. For example, proxies inheriting the IRemote interface will implement asynchronous versions of the standard get and set methods:

```
      IAttributeList get(IAttributeList);     // Synchronous implementation
      boolean get(IAttributeList, ICallback) // Asynchronous implementation

      IAttributeList set(IAttributeList);     // Synchronous implementation
      boolean set(IAttributeList, ICallback) // Asynchronous implementation
```

The asynchronous version accepts an ICallback object and returns a boolean value. The return value indicates whether or not the get command could be dispatched to the communication middleware. This call should return immediately, and will fail only if there are network issues. The callback object will receive the response from the server when the task has been completed. This allows the client to receive and handle event status without blocking the issuing thread.

Proxy objects are generated and returned to the client during the call to the connection service's connect method. This method will find and connect to the remote object specified by the component name, and then attempt to determine the appropriate proxy to create by iterating through the various connectors. When the respective connector has been found the proxy will be generated and returned to the client. The proxy will remain valid for the lifetime of the remote object or until the network connection is broken or client application is shutdown. Depending on the capabilities of the communication middleware, a variety of fault recovery and operation failure techniques may be implemented into the proxies. This can include automatic reconnection, server redundancy and failover, and invocation retry. Additionally, proxies may implement middleware specific efficiency and quality of service strategies to improve throughput and bandwidth utilization, including batching and one-way invocations.

7.5.1.3 Stubs

Stubs implement the bridge between the communication middleware and the server components. As with proxies, stubs implement an identical interface to the component they represent. Stubs however, work as the inverse to proxies by forwarding data and commands from the communication middleware to the components. Stubs are essentially a wrapper for component implementations to bridge the communication specific environment with the abstract KCSF middleware. Components implemented in KCSF are completely unaware of how their methods are invoked, and do not need to take any specific action to format data for one stub implementation over another. The logic to transform and send data is handled entirely by the stub.

There are five classes of stubs:

- ILocalStub
- IDeviceStub
- IContainerStub
- IContainerManagerStub

Each of these stubs implements the server-side bridge between the middleware and the ILocal, IDevice, IContainer, and IContainerManager classes. Stub implementations react to events from the communication middleware by processing and pushing data to components. The stub receives information from the client proxy to determine the operation to be executed in the component, and invoke the corresponding method. When the operation has completed, any data returned by the component method call will be transformed and returned to the invoking client. From the perspective of the component, all calls appear to be performed by local clients. Data is processed and returned as usual, without required knowledge of whether the client is local or remote.

7.6 Events

Events are transmitted using a publish-subscribe decoupled pattern. Events allow a collection of data in the form of attribute lists to be given a name and published in a fire-and-forget manner. Interested parties can subscribe to these named events and will receive them anytime they are posted. Subscribers can come and go without affecting publishers.

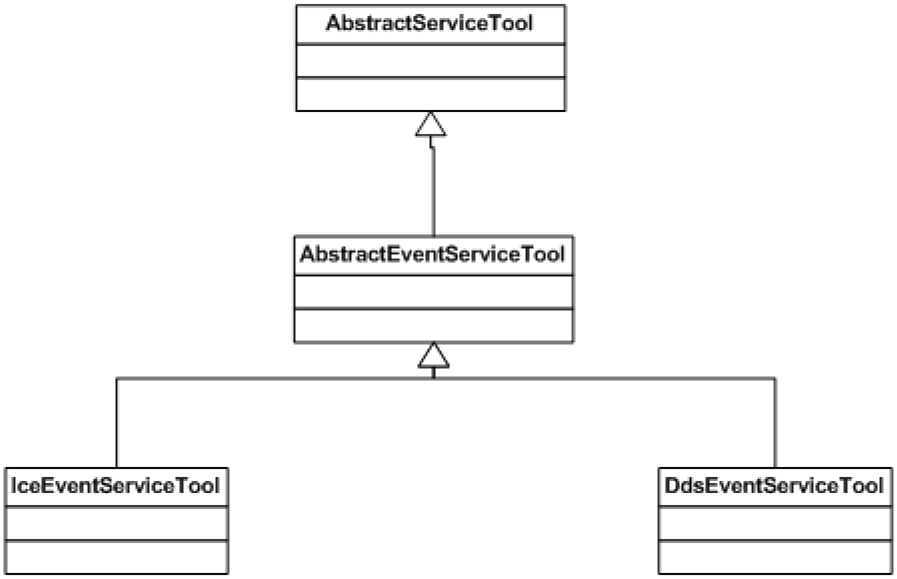


Figure 38: Event service class hierarchy.

The event service can use any middleware that has publish-subscribe capabilities. CSF provides an implementation based on ICE, and KCSF has prototyped implementations based on two different implementations of DDS. In either case the actual implementations extends from `AbstractEventServiceTool` and present the same API to the end user.

Event subscriptions are kept track of by the service and are organized by source and event name in a hash table. The table `subInfo` contains the subscriptions by `eventName` (or topic). Each `eventName` may have multiple subscriptions.

7.6.1 ICE Event Service Tool

The ICE implementation makes use of an ICE service called IceStorm. IceStorm is a publish–subscribe event distribution service and is a standard part of the ICE package.

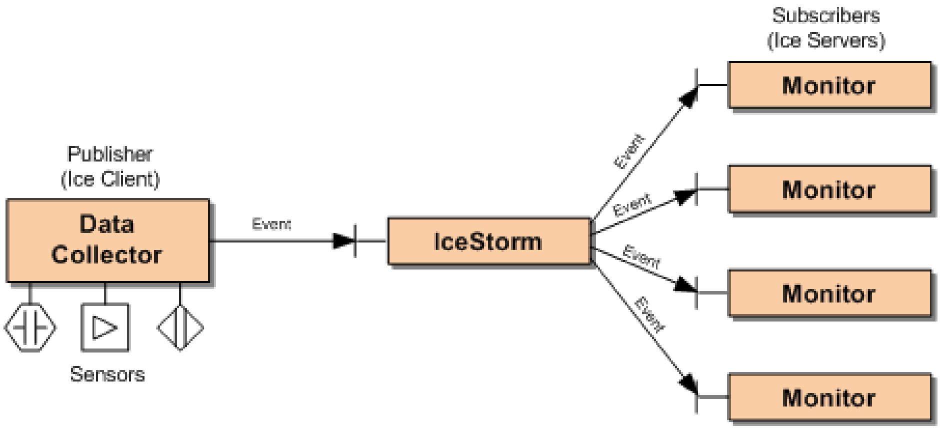


Figure 39: ICE Event Service Tool.

IceStorm provides a lot of flexibility: topics can be federated to provide custom event propagation. In addition IceStorm can operate in a high-availability mode where a number of servers form a replica group. If a server goes down, IceStorm automatically reroutes communication to servers that are still running, and when a server comes up, it automatically resumes event delivery via that server without any manual intervention. Event subscriptions can be made persistent, so clients do not require special action after a restart for event flow to resume. IceStorm also provides a configurable mechanism that allows you to deal with stale subscriptions to clients that become dysfunctional, for example, by being disconnected from the network for an extended time.

7.6.2 DDS Event Service Tool

The DDS implementation is based directly on DDS which is by design a publish-subscribe middleware. Unlike the ICE implementation there is no intermediary server (IceStorm) required. Posted events go directly to the subscribers. The DDS event service uses a single DDS Topic called `EventTopic` which contains the source name, event name and attribute list. The event name is configured to be a DDS key.

7.6.3 Strategies

Regardless of the implementation there are various strategies that can be applied to tweak or fine tune the service as needed. Some of these have already been explored by ATST for ICE others can be explored further with DDS. Some of the different strategies that can be explored further are:

- Batching of events to improve performance (at the risk of latency)
- Use of best effort delivery instead of reliable
- Executing event callbacks on separate threads to increase decoupling
- Use of single vs. multiple data writers and readers for DDS

7.6.4 Event Callback Adapter

KCSF provides an adapter class that can be extended by subscribers to simplify callback processing. This class is called `EventCallbackAdapter` and implements `IEventCallback`. Use of this class is not mandatory but it provides a simple standardized way to handle callbacks and retrieve event data.

7.7 Tasks

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

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

7.7.1 Asynchronous Task Development

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

```
class PointTelescope extends Task {

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

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

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

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

         return res;
      }
   }
}
```

This PointTelescope class only overrides the base class methods for initialize, start and wait. The class constructor will perform the required preparation for the task. The constructor arguments are passed up to the superclass to be saved in the params member of the base class. As an attribute list these arguments can be passed directly to target components in the system, or processed for specific formatting prior to command execution.

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

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

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

7.7.2 Synchronous Task Development

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

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

The base `Task` class is all set up to enable synchronous-style tasks, where the subclass only needs to provide an execute method. If a task subclass does not override it, the task's `start` method simply adds the task instance (itself) to the desired task executor. When the resources are available to process the task the executor will activate and execute the task. Similarly, if not overridden, `wait` understands how to listen and block for the result of a task and obey the timeout parameter. This is shown in Figure 40.

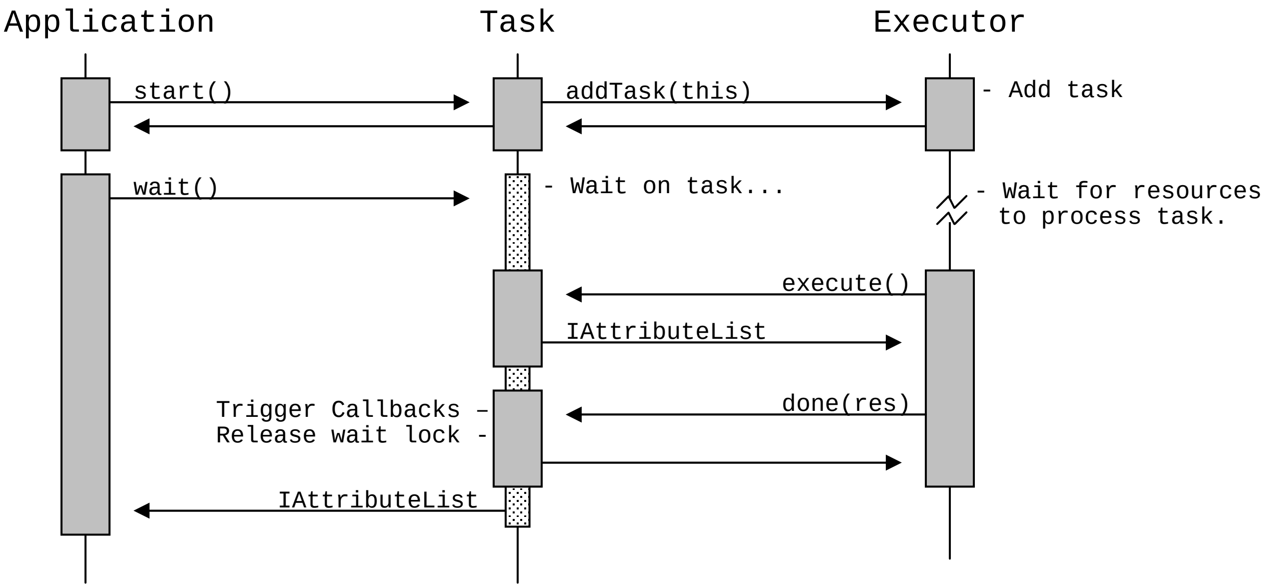


Figure 40: Task Execution.

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

```
class PointSpread extends Task {

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

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

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

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

      // Calculate point-spread function on image.
      …

      return res;
   }
}
```

This task inherits the `start` and `wait` methods from base Task, while overloading the `initialize` and `execute` methods to implement the task logic. By inheriting most of the default methods, synchronous simple tasks can clearly express the business logic of a task without much extraneous task-related detail cluttering up the code. The full range of the Java standard library is available, and the task is free to define other methods to subdivide up the problem and make the program structure more manageable, provided they conform to the Command pattern and do not conflict with the task interface method names.

7.7.3 Task Executors

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

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

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

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

7.7.3.1 Command Thread Pool

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

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

The conceptual model of a thread pool is simple: the pool starts threads running; work is queued to the pool; available threads execute the queued work. In our case all tasks will be handled identically because of the standardized interface. In the thread pool pattern, a number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. Generally a thread pool allows a server to queue and perform work in the most efficient and scalable way possible. As soon as a thread completes its task, it will request the next task from the queue until all tasks have been completed. The thread can then terminate or sleep until there are new tasks available.

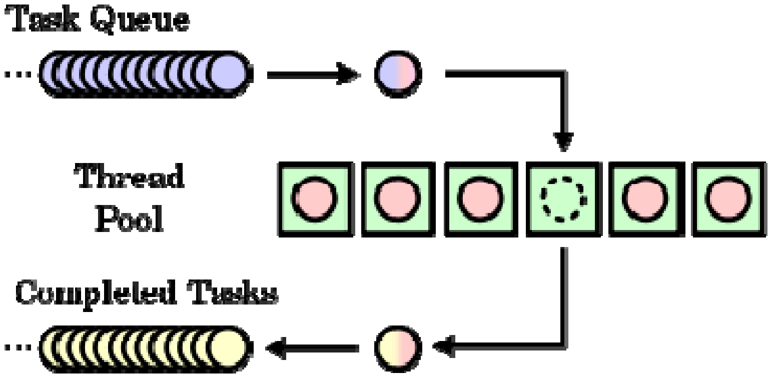


Figure 41: Simple Thread Pool.

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

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

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

The following pseudo-code details the CommandThreadPool executor interface.

```
public class CommandThreadPool implements ITaskExecutor {

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

   public CommandThreadPool(int poolSize);

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

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

The CommandThreadPool class makes use of the Java ExecutorService to create and manage an internal pool of threads of a desired size. Each of the threads created by the executor will be started in its own instance of the task processing control class, ThreadControl. This class implements the individual thread logic to pop a task from the queue, call execute, and finally invoke the task's done method before returning for new tasks. The queue is implemented as a blocking queue, where thread access to items is synchronized and blocking is performed until items are available for processing.

The following methods are defined for the CommandThreadPool class:

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

7.7.4 Handling Errors, Dynamic Task Control

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

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

7.8 Sequencers

Sequencers are implemented as a state-driven KCSF Controller. Each sequencer has its own unique state mappings based on the role it will fill. These mappings are typically built into the code of the sequencer or represented by an external class. A suggested freeware tool that can be used to develop state mappings is the State Machine Compiler (SMC). This java application takes a file containing the user defined state-transition mappings and generates a fully operable state machine in the desired target language (e.g. Java). The sequencer implements the business logic for the transitions, and binds to the state-machine instance. Invocating transitions on the state-machine executes the corresponding tasks and sets the new state.

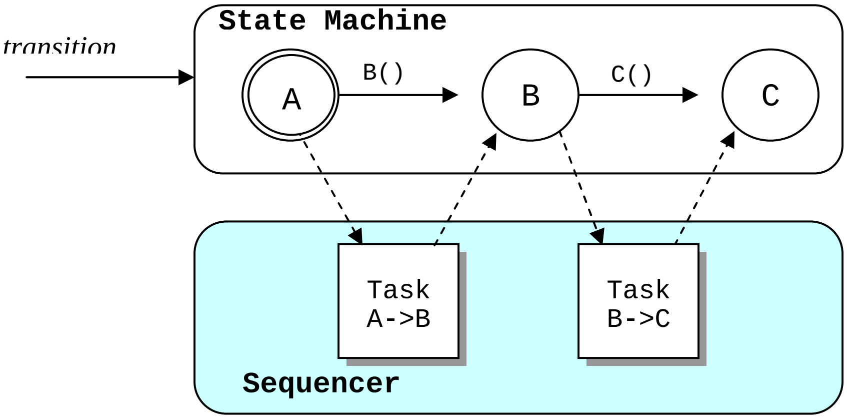


Figure 42: Executing Transition Logic.

Developing a state-machine, and by extension, a sequencer, requires careful consideration to account for all of the intended responsibilities of the design. Each state should reflect a specific configuration or step identified within an observing or motion control sequence.

7.8.1.1 Common States

Although each sequencer will have many unique states, there are a set of states that will be common to all sequencers. These states are derived from the typical state-machine design used by hardware devices – the control targets for the majority of sequencer implementations.

- `START` – The entry state of the sequencer. It is assumed that a sequencer in this state has recently been created, but not yet initialized or configured.
- `INIT` – The initialization and configuration state of the sequencer. Typically the operations performed during this state only need to be performed once after startup. This may include gathering configuration information, creating and initializing class members, and obtaining services.

- REINIT – A fast or repeatable initialization phase. This state typically implements a set of the sequencer initialization that may have to be repeated multiple times during a night (e.g. connecting to devices, refreshing telemetry, etc.) Usually execution of the INIT state will automatically transition through REINIT, and then to STANDBY.
- HALT – Indicates that an operation has been interrupted. This is usually entered by explicit command from the operator.
- STANDBY – the sequencer is in an idle state, and is ready to receive and process commands.
- SHUTDOWN – the termination state of the sequencer. This state is responsible for closing connections, releasing resources, and shutting down the sequencer.
- FAULT – this state indicates that an unhandled or non-recoverable error has occurred and the sequencer had to stop. Operator intervention is expected to resolve the problem.

In addition, Sequencers that are responsible for acquiring targets or positioning devices will also typically use a SLEW/TRACK state pattern. In this design, any sequencer operation that involves starting a process and then waiting for one or more devices to achieve a specific state will quickly execute the task(s), and enter a SLEW state. The sequencer will wait in the SLEW state until the tasks are complete.

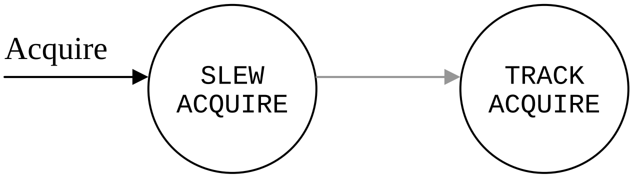


Figure 43: SLEW-TRACK States.

The SLEW state is known as a *transient* state: a temporary state that will automatically transition out when an internal event occurs (i.e. not caused by an explicit user action). When the devices have reported in position the task will issue its own transition to move to the TRACK state. As with other states, if there is a problem during the task execution of a transient state the state machine will typically enter FAULT to signal a system error.

7.8.1.2 Developing with SMC

The State Machine Compiler library is used to convert state-transition mappings into a callback processing class. Developers define a state machine using the SMC syntax. For each state, the mapping will define each of the available transitions and the target state when the transition completes. The following example illustrates a simple SMC mapping for a single state.

```
START
{
// START will recognize and accept an Init(void) transition.
   Init()
      // When the transition completes the state machine will
      // be put into the INIT state.
      INIT
      {
         // The transition process will execute the
         // InitializeTask routine defined by the Sequencer.
         initializeTask();
      }

   Fault()
      FAULT
      {
      }
}
```

The compiler will read this definition in the following way:

- The state machine will posses a `START` state.
- The `START` state can be left by issuing an `Init` or `Fault` transition.
- When an `Init` is issued the machine will transition to the `INIT` state, after executing `InitializeTask`.
- When a `Fault` is issued the machine will transition directly to the `FAULT` state.

Only those transitions and tasks defined in the mapping will be permitted. If the client attempts to issue a transition not recognized by the current state an exception will be thrown. The following pseudo-code details the interface produced when the mapping is compiled.

```
public class GeneratedStateMachine  {
   public GeneratedStateMachine(<Type> machineImplementation);

   public void Init();
   public void Fault();
}
```

The `machineImplementation` parameter of the class constructor refers to the actual object that implements the state machine transition tasks (e.g. `InitializeTask`). This will be a reference to the Sequencer itself.

```
public class Sequencer extends Controller {
   public Sequencer(…) {
      this.stateMachine = new GeneratedStateMachine(this);
   }

   public void initializeTask() {
      // Implements the actual work to be performed for an INIT
   }

   ...
}
```

The SMC syntax offers additional capabilities including transition parameters, conditions, guards, code injection, default transitions and more. See the http://smc.sourceforge.net/ for more information on the SMC compiler.

7.8.1.3 Processing Transition Requests

As Controller subclasses, Sequencers will receive transition requests from clients through the `execute` method. Transitions are defined in Attribute Lists in a similar way as standard device commands,

- The reserved *_Action* keyword defines the transition. This may be a string, integer, or other compatible type.
- Parameters custom to the transition will be defined in additional attributes. It is the responsibility of the client to be aware of all the attributes required by the sequencer.

The `doExecute` of the Sequencer is responsible for transitioning the state machine, and may be implemented in the following way.

```
void doExecute(IAttributeList command) {
   int transition = command.getString("_Action");
   switch(transition) {
      case INIT:
         // Get additional parameters, if required by transition.
         // Issue transition.
         this.stateMachine.Init();
         break;
      ...
   }
}
```

When executing a transition on the internal state machine, the Sequencer's thread will block until the transition task has completed, and the state machine enters the target state. This should be kept in mind when determining the default size for the Controller's thread pool. At a minimum two threads should be active to allow the user to halt or asynchronously command the sequencer while a transition is being processed.

7.9 Key differences between KCSF and CSF

Figure 44 shows a snapshot of the current code base from ATST. Items shaded in green represent code that is completely reusable from ATST. Items shaded in blue represent completely new code that will be provide by Keck. Items in white represent code not used as part of KCSF. Dual-colored items represent ATST code that has been modified to better suit Keck needs. Note: that if Keck decides to stay with ICE as the communication middleware then the DDS code will not be needed.

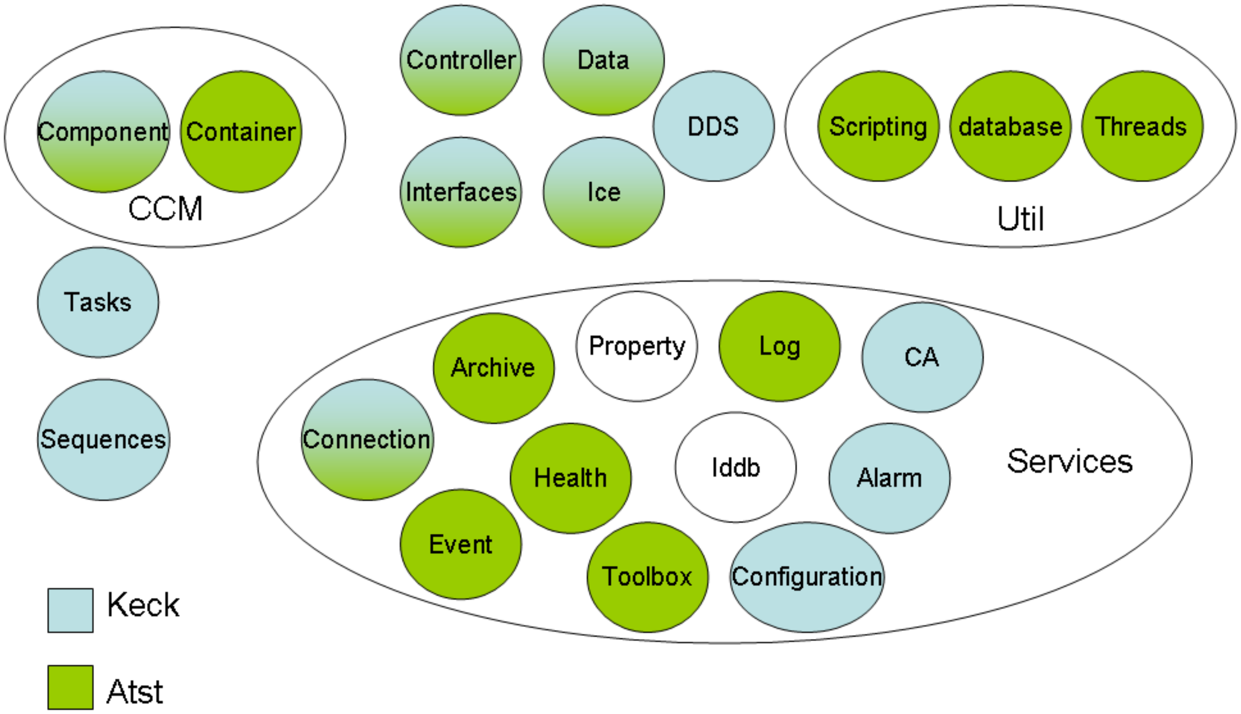


Figure 44: KCSF Code base

Many changes for KCSF are minor, in many cases resulting in method or class name changes for the sake of clarity or aesthetics. Examples include the renaming of `IAttributeTable` and `AttributeTable` to `AttributeList` (since it was actually a list and not a table). These decisions can be revisited as needed.

Fundamental design changes are related to the concept of components, controllers, their interfaces and interactions and the data transfer objects used to pass data between them.

KCSF currently does not plan to use the CSF property service of the IDDB service. New services introduced by KCSF are the alarm service, the CA service and configuration service (which replaces the CSF property service)

7.9.1 Attribute Internal Representation

KCSF shares the same concept of `Attributes` (as named value pairs) and `AttributeLists` (a collection of attributes) as CSF. The `IAttribute` and `IAttributeList` interfaces in KCSF are almost identical to CSF. The two frameworks differ though in how an `Attribute` internally represents its data and how they are marshaled on the wire. In ATST an attribute value is represented as an array of strings. In KCSF an attribute value is represented as a union of raw data types which can be scalar or multi-dimensional. These changes constitute the code changes to the “data” package.

7.9.2 Components, Controllers and their interfaces

The greatest difference between CSF and KCSF relates to Components, Controllers, their interfaces and commands. It should be noted that both KCSF and CSF share the same Container

Component Model and technical interfaces. In CSF a Component is the foundation for all applications in ATST. Most ATST applications extend the Controller class, 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.

In ATST a component refers to a concept, i.e. a component is a self-contained piece of software that conforms to a well-defined interface, and to an actual implementation, in that a component is a base class from which ATST software developers are generally expected to use derivatives of (e.g. Controller). For KCSF we wanted to have a clearer distinction between the concept of a component as part of CBD and the framework basic building block. For this reason in KCSF the basic building block for application development is a device.

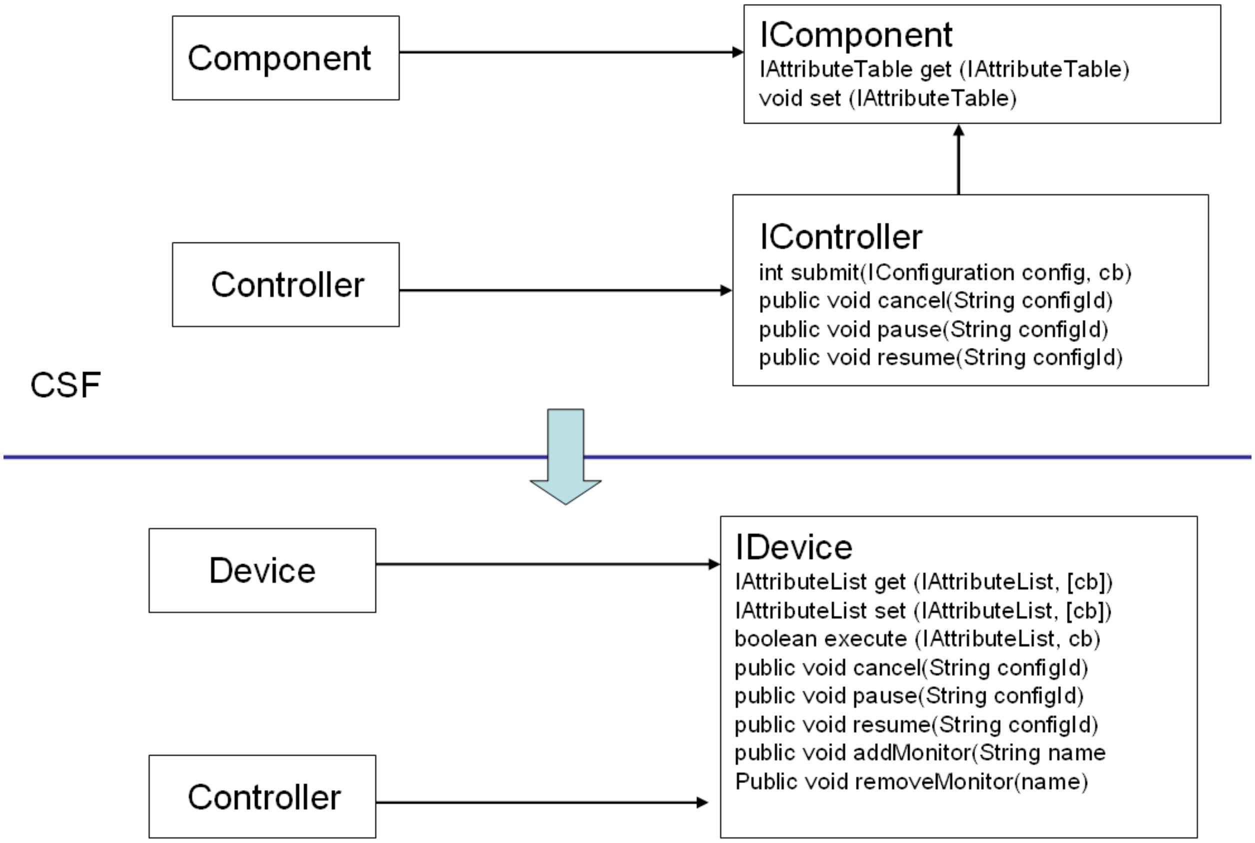

Component
IComponent
IAttributeTable get (IAttributeTable)
void set (IAttributeTable)
Controller
IController
int submit(IConfiguration config, cb)
public void cancel(String configId)
public void pause(String configId)
public void resume(String configId)
CSF
Device
IDevice
IAttributeList get (IAttributeList, [cb])
IAttributeList set (IAttributeList, [cb])
boolean execute (IAttributeList, cb)
public void cancel(String configId)
public void pause(String configId)
public void resume(String configId)
public void addMonitor(String name
Public void removeMonitor(name)
Controller

Figure 45: Component and Controller Differences between CSF and KCSF

Figure 45 shows the main differences between the two frameworks. In KCSF we have chosen to represent everything as a device. A controller is a special type of device that can handle multiple simultaneous commands. There are a number of differences between the `IDevice` interface and the combined `IComponent` and `IController` interfaces. These are summarized below:

- In KCSF `execute` (similar to CSF `submit`) is available to both a device and controller.

- In KCSF get, set can be called synchronously or asynchronously.
- In KCSF there is the concept of monitors that can be applied directly to a device.

With KCSF a device is the basic building block. The get and set methods support the viewing and changing of device attributes while the execute method allows operations to be performed.

7.10 Service Details

The KCSF provides developers with a suite of tools and services that simplify a wide range of typical software tasks. All service tools must implement or extend the IServiceTool interface.

A ServiceTool can perform special processing on service access requests. All service helpers must support wrapping of other service helpers. This interface defines the base methods that all service helpers must provide. Rather than directly implement the interface, this interface is implemented by KCSF.cs.services.AbstractServiceTool which all service helpers should subclass.

The IServiceTool API is as follows, and allows a service to be reset, started, stopped, destroyed or chained. Chaining allows a new service helper to be placed onto the front of the service list and unchaining removes the top service helper.

```
public IServiceToolAdmin<B> getTool();
public boolean sameAs(IServiceToolAdmin<B> sTool);
public void reset(B tb);
public void startService(B tb, String appName);
public void stopService(B tb, String appName);
public void killService(B tb, String appName);
public void chainTool(IServiceToolAdmin<B> helper);
public void unChainTool();
```

Typically any new type of service will extend IServiceTool adding methods unique to that service as it does so. For example we see that

```
public interface IEventServiceTool extends IServiceTool
public interface IHealthServiceTool extends IServiceTool
```

and so on. IEventService adds methods for post, subscribe and other event related calls.

Although not mandatory it is expected that in most cases there will be an extension of AbstractServiceTool for each service. For example

```
public abstract class AbstractEventServiceTool extends
    AbstractServiceTool<IToolBoxAdmin> implements IEventServiceTool


public abstract class AbstractArchiveServiceTool extends
    AbstractServiceTool<IToolBoxAdmin> implements IArchiveServiceTool
```

This allows for different flavors of services to be developed while maintaining common code in the abstract. An example might be the Log service where they could be implementations to write to file, a database, console IO etc.:

```
public class BufferedLogServiceTool extends AbstractLogServiceTool
public class LogServiceTool extends AbstractLogServiceTool
...
```

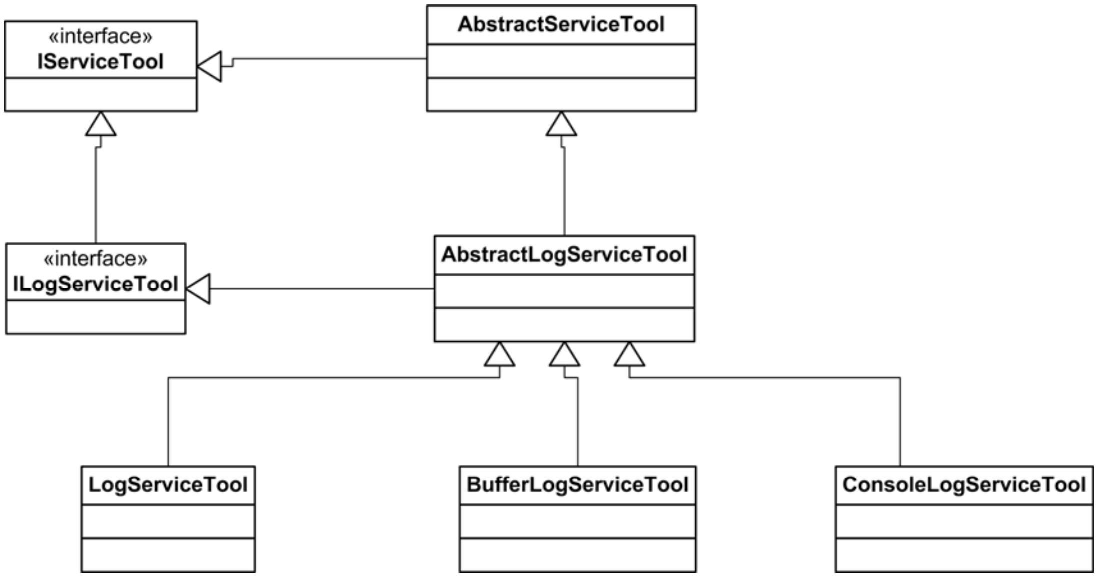


Figure 46: Log Service Example

Like any wrapper the `AbstractServiceTool` takes care of much of the housekeeping functions and delegates the interfaces method calls as needed to the real implementation.

7.10.1 Connection

The API for the connection service is as follows:

```
public interface IConnectionServiceTool extends IServiceTool {
   /*
    * Makes a component available to the network.
    */
   void bind(IToolBoxAdmin, String name, ILocal object);

   /*
    * Removes a component from the network.
    */
   void unbind(IToolBoxAdmin, String name);

   /*
    * Connects to a remote object and returns a proxy.
    */
   IRemote connect(IToolBoxAdmin, String target);

   /*
    * Disconnects a proxy from a remote object.
    */
   void disconnect(IToolBoxAdmin, IRemote proxy);

   /*
    * Returns a list of all bound objects of the specified type.
```

```
      */
     String[] allRegistered(IToolBoxAdmin, String Type);
  }
```

All connection service implementations implement this interface and extend `AbstractConnectionServiceTool`. The connection service's two main methods are `bind` and `connect`. The `bind` method accepts an instance of a component object and its unique name. The `bind` will create an appropriate stub for the component, publish the existence of the object over the network, and map the name to the object reference. The `connect` method accepts a string representing the unique name of a bound object. A corresponding proxy will be determined for the remote object and returned to the client.

7.10.2 Archive

Modern systems typically involve a database somewhere in the design. The database creation, updating and management should be abstracted so that the choice of RDBMS has little or no impact on the overall framework. The database support includes software on which one can implement persistent stores, engineering archives, and data repositories. Examples of database usage in KCSF are:

- Configuration. Components may retrieve information from a persistent store. Configuration parameters for components are maintained in the persistent store and are retrieved by components on initialization.
- Alarm Logging
- Logging
- Data Archiving

7.10.2.1 Persistent store

The term persistent store will be used to describe one or more databases. The databases are transaction-based, high-performance databases. Database queries are available via SQL, as are database insertions. Database accesses however, are hidden behind a persistent store interface to allow replacement of one database implementation with another. This persistent store interface provides a mapping between architectural concepts in KCSF and the database model. The most likely database candidates are PostgresSQL and MySQL. Depending on the application needs some stores may be optimized for insertion, others may be optimized for queries.

7.10.2.2 Database abstraction

The first level of abstraction will be in the use of a JBDC driver. JDBC is an API for Java that defines how a client may access a database. It provides methods for querying and updating data in a database. JDBC is oriented towards relational databases. Through JDBC the steps required to create a connection to a database, perform queries etc. are uniform regardless of the underlying database.

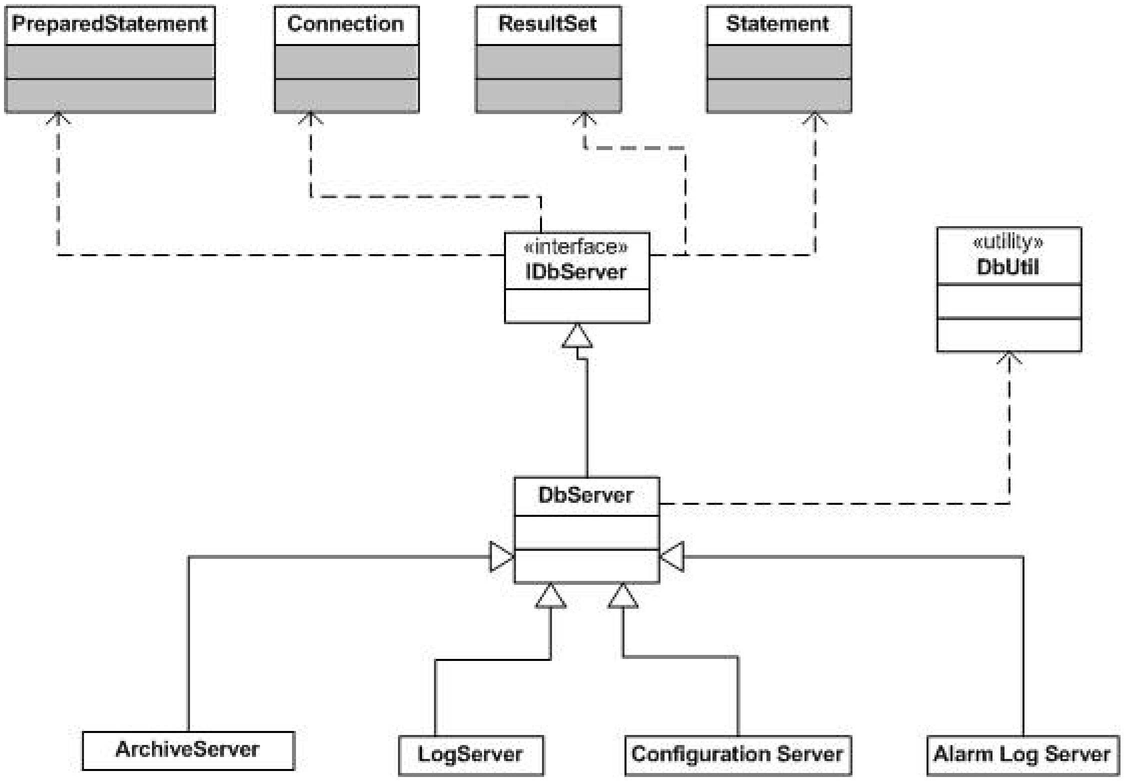


Figure 47: Database Support.

An `IDbServer` interface defines the methods applicable to a database server. A base implementation called `DbServer` will provide application access to a specific database while hiding any database specifics. Using a utility library, operations such as selecting the appropriate JDBC driver, building queries, opening, closing, inserting, deleting etc. will be handled by the `DbServer`. The items color coded in grey above represent the JDBC classes; items in white are part of the framework. Each service that has a property store will have an associated server which extends `DbServer`. These classes will be tailored to the particular needs of that service.

For Archiving there is an `ArchiveServer`. The server contains code to be able to create an archive database, backup an archive database, perform single and batch updates to the Archive table and to perform other maintenance tasks. The `ArchiveServer` is used directly by the service where the `record` method calls the server to perform the insert. Once in the database the saved attributes can be viewed with the `ArchiverView` UI and managed with the Archive Manager UI.

7.10.3 Health

The health server is virtually fully automatic and implemented in the technical architecture. Developers need only implement a single method as part of the component:

```
      void performCheckHealth();
```

In that method, developers determine the health of the component and sets it with

```
      Health.setHealth(String healthLevel, String msg);
```

Where healthLevel can be one of: Health.GOOD, Health.ILL, Health.BAD and msg is a human-readable description of health cause.

Example:

```
   public void performCheckHealth() {
      switch (sockets.numAvailable()) {
      case 0:
         Health.setHealth(Health.BAD, “No sockets left”);
         break;
      case 1:
         Health.setHealth(Health.ILL, “Sockets running low”);
         break;

         Default:
            Health.setHealth(Health.GOOD, null);
         break;
      }
   }
```

The health system works as follows. During container initialization a class called HealthServer is created. The HealthServer is made aware of all components added to or removed from a container. Its sole purpose is to periodically walk through the list of components and to call checkHealth (which will delegate to performCheckHealth). For each call a HealthReport event will be issued through the event service. These events can be subscribed to by a health monitor, typically a UI or it could be a dedicated application.

7.10.3.1 Health Event / Report

A health update consists of the following fields:

- name -- name of the application this record is about
- health -- last reported health (Good, Bad, Ill, Unknown)
- reason – message explaing the health
- repeat -- "true" if this is a repeat health status, else "false"
- simulated -- "true" if application is simulated, else "false"
- acked -- "true" if status has been acknowledged, else "fasle" (can be used by monitor)

7.10.4 Logging

As with any complex software project, systems developed with KCSF will have the need to log a wide range of run-time information from simple trace messages to system critical errors. In order to simplify the logging process a log service will be developed as part of the KCSF infrastructure. The log service will provide developers with the ability to open and write to multiple log streams, each one with the option of individual customization. The log service will be built upon the Java utility logging package. This package provides developers with the basic framework to issue and publish log messages. Within the KCSF, the abstract log service design will allow for a variety of logging mechanisms and formats including console, file and database.

The Java logging package follows a simple implementation divided into three concepts: loggers, handlers, and formatters. Loggers are responsible for accepting and processing log messages from the application: they are the main interface to the logging system for clients (i.e. the Log Service). Handlers implement the stream functionality of a specific logging mechanism.

Common handlers such as the console and file handler are implemented as part of the Java package. Additional handlers can be developed to write to databases or over a network by extending the base `Handler` class. Formatters are responsible for converting a log record into readable text. As with handlers, custom formatters can be developed to suit the user’s needs. The Java package also provides a default formatter that implements a simple time stamped event output. Each handler can use its own unique formatter, or the same one can be used for all handlers.

7.10.4.1 Handlers

The Handler class is the base class for all log publishing objects and is primarily responsible for opening, maintaining, and closing connections to output streams.

```
public class ILogHandler {

   public ILogHandler();

   /*
    * Set the log level.
    */
   public void setLevel(LogLevel level);

   /*
    * Return the current log level.
    */
   public LogLevel getLevel();

   /*
    * Return the handler.
    */
   public java.util.logging.Handler getHandler();


   // The java handler object.
   protected java.util.logging.Handler mHandler;

   // Current log leve.
   protected LogLevel mLevel;
}
```

Log messages are pushed to handlers through the `publish` method. This method accepts a `LogRecord` instance that defines a single log message – including the level, time of creation, and the raw application message. The publish operation is responsible for formatting the log record, and writing it to the output destination.

As with the Logger class, Handlers can be configured to filter messages based on severity levels through the `setLevel` method. This allows a user to restrict the logging of application messages on a handler-to-handler basis.

The Java logging package defines two fully implemented handlers for client use: `ConsoleHandler` and *`FileHandler`*. The Console Handler is used to write data to an xterm or OS console. The File Handler writes log messages to a file using a rotating file name (as a file reaches a defined maximum file size it is closed and a new file is created appended with an integer suffix.) There is also a base class implementation for a `SocketHandler`, which provide a simple network interface, and a generic `StreamHandler` for stream based logging. Additional handlers will be developed for KCSF, including a database handler.

7.10.4.2 Formatters

The Formatter is a simple class used by handlers to convert a log record into a human readable string that can be written to an output. Formatters are invoked during the `publish` operation of the handler through the `format` method. This method accepts a raw `LogRecord` instance and is intended to return a single printable string. The handler is responsible for publishing this string to the output.

Only one formatter can be assigned to a handler at a time, but each handler can have its own unique formatting. For consistency it is recommended (and will be enforced by the KCSF) that a single formatter be used for all human readable outputs. For log files that are intended for use by applications or other post processing tools, a character- or whitespace-delimited format may be required.

The Java logging package provides a simple default formatter for basic logging purposes, as well as an XML formatter that will convert a `LogRecord` into XML compliant output.

7.10.5 CA Client Service

Using the Cosylab Channel Access for Java Library makes the implementation of a CA client service very straight forward. Channel Access for Java (CAJ) is a 100% pure Java implementation of the EPICS Channel Access server and client library. The steps are as follows:

- In **doStartService** create a client context
  ```
  clientContext = (CAJContext) JCALibrary.getInstance()
                  .createContext(JCALibrary.CHANNEL_ACCESS_JAVA);
  ```
- In **doStopService** remove all monitors, channels and free the context
  ```
  Enumeration<String> monitorNames = monitors.keys();
  for (Enumeration<String> monitorName = monitorNames;
      monitorNames.hasMoreElements();)
          removeMonitor(monitorNames.nextElement());


  Enumeration<String> channelNames = channels.keys();
  for (Enumeration<String> channelName = channelNames;
      channelNames.hasMoreElements();)
      removeChannel(channelNames.nextElement());

  clientContext.destroy();
  ```

- For every `put` or `get` check to see if we have a channel created. If not create one and add it to the cache. Then call the channel put or get method.
  ```
  // Example on how to create a channel and add to internal cache
  Channel channel = clientContext.createChannel(name);
  if (channel != null) {
     channels.put(name, channel);
     clientContext.pendIO(timeout);
  ```

```
      }

      // Example implementation for put
      Channel channel = findChannel(name);
      if (channel != null) {
         channel.put(value);

         // Example implementation for get
         int nativeCount = channel.getElementCount();
         DBR dbr = channel.get(DBR_Byte.TYPE, nativeCount);
         clientContext.pendIO(timeout);
         return ((DBR_Byte) dbr).getByteValue()[0];
   }
```

7.10.6 CA Server

This is not technically a service but a component. It is described here since it is so closely related to the CA client service in terms of related functionality and implementation.

The CA Server will be implemented using the Channel Access Java (CAJ) and Java Channel Access Server (JCAS) libraries provided to the EPICS community by Cosylab. This is a 100% pure Java implementation requiring no JNI and so can work on any platform. The framework is simple. JCAS is an addition to the existing Channel Access in Java (CAJ) client library; both share common code and are packed in a same Java Archive (JAR) file.

Creating a server is very straightforward. `ServerContext` requires an implementation of the `Server` interface. This interface has two methods which must be implemented:

- `processVariableExistanceTest`
    - This method is called each time a CA client broadcasts a search for a particular process variable, identified by a name. If a positive answer is returned by the method, JCAS library will announce that it hosts that process variable.
- `processVariableAttach`
    - Once a CA client is knows a process variable exists it will most likely issue a request for channel creation. A channel is a connection between the server and client through which a single process variable is accessed. The client never talks directly to a process variable, only through the channel. To create a new channel `ServerContext` will request a `ProcessVariable` instance by calling the `processVariableAttach` method. Then it will create a channel instance by calling the `processVariable createChannel`() method.

Figure 48 shows the sequence diagram for a client interaction with the CA server.

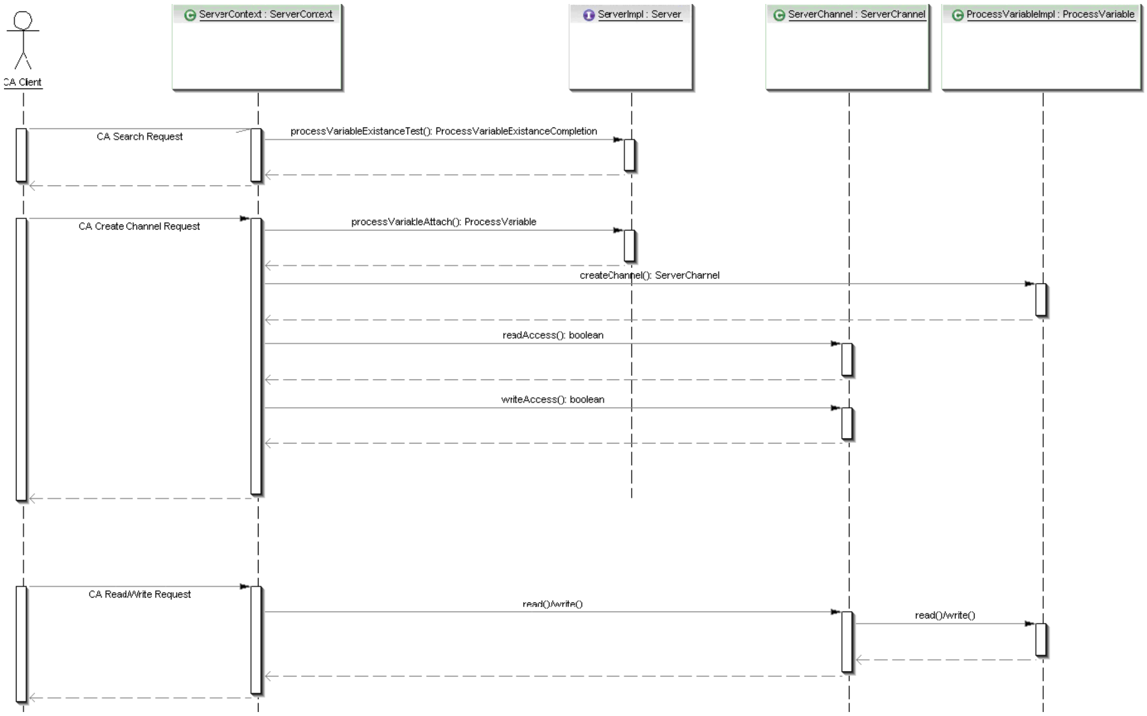

CA Client
ServerContext : ServerContext
ServerImpl : Server
ServerChannel : ServerChannel
ProcessVariableImpl : ProcessVariable
CA Search Request
processVariableExistanceTest(): ProcessVariableExistanceCompletion
CA Create Channel Request
processVariableAttach(): ProcessVariable
createChannel(): ServerChannel
readAccess(): boolean
writeAccess(): boolean
CA Read/Write Request
read()/write()
read()/write()

Figure 48 Sequence Diagram for CA Server

7.10.6.1 Process Variables

`ProcessVariable` is an abstract class that should be extended that provides a default implementation that returns a `ServerChannel` instance. This implementation is a default implementation of a channel and grants all read and write rights to any user.

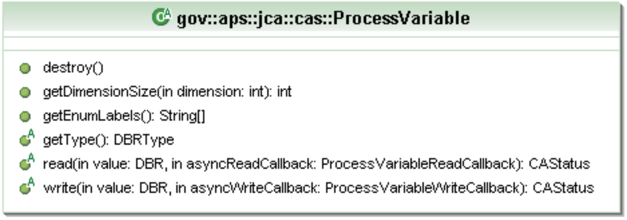

gov::aps::jca::cas::ProcessVariable
destroy()
getDimensionSize(in dimension: int): int
getEnumLabels(): String[]
getType(): DBRType
read(in value: DBR, in asyncReadCallback: ProcessVariableReadCallback): CAStatus
write(in value: DBR, in asyncWriteCallback: ProcessVariableWriteCallback): CAStatus

Figure 49: ProcessVariable class with methods that are important for a developer.

The `ProcessVariable` class has three abstract methods to be implemented:

- `getType()` – Return PV native data type. Types used to read/write.
- `write()` – Write value to PV. Given DBR type is basic native data type (e.g. `DBR_Double`).
- `read()` – Read value from the PV. Given DBR type is always at least of DBR `TIME` type. If of `GR` or `CTRL` type, it depends on the PV to support it.

Methods `getDimensionSize()` and `getEnumLabels()` are most likely to be overridden. However, it is recommended that the default PV implementations located in `com.cosylab.epics.caj.cas.util` package be used. They provide instructions on how to implement your own process variables.

7.10.7 Alarm

An alarm can be considered an event except that is has an associated condition, one which is deemed to be abnormal and requiring special attention. A condition is associated with a Source which is a typically KCSF component. In terms of delivering alarms when a component issues a set or clear alarm call the framework will use the event service to propagate the alarm information.

7.10.7.1 Definitions

The following provides more definition for the main alarm concepts.

- **Alarm**
  An *alarm* is an abnormal *condition*, which requires special attention outside the control application. Each alarm instance is associated with a *source* (or *owner)* device. For example, the device `AO.Cameras.TT` (the *source*) may have an active `HighAlarm` *condition* or the device `AO.MCS.M1` (the *source*) may have an active `LowAlarm` *condition*. Each alarm instance is uniquely identified by the combination of *source*, and *condition*.

- **Condition**
  An alarm *condition* is a named abnormal state for a *source*. Examples of alarm *conditions* are: `LowAlarm`, `HighAlarm`, `HighHighAlarm`, `DeviationAlarm`, etc.

- **Source**
  All alarms are owned by named items in the domain. Any component can be an alarm source. A source may be the owner of several alarm conditions. The term *owner* is also sometimes used in place of term *source*.

- **Severity**
  This specification defines *severity* as an indication of the urgency of the alarm. This value may range from 1 to 1,000, with 1 being the lowest urgency and 1,000 being the highest.

- **Alarm Area**
  This specification defines an *Alarm Area* as a collection of alarm groups. A client of the Alarm Manager may request a list of all active and unacknowledged alarms for a specified area.

- **Alarm States**
  An alarm occurrence can be in one of four states:
  - Inactive/Acknowledged (not included in active alarm list)
  - Active/Unacknowledged
  - Active/Acknowledged
  - Inactive/Unacknowledged

- **Alarm Instance**

It is expected that the Alarm manger will maintain the following data (or similar) per alarm instance:

Source	**String**	**Identifier for the alarm owner**
Condition	String	Identifier for the alarm condition
Group	String	Identifier for the native alarm group
Annunciation	String	Identifier for annunciation method
TimeStamp	Time	Date and time for last state change
Severity	Integer	Severity level
Active	Boolean	True while alarm is Set
Acked	Boolean	True is alarm has been Acknowledged
Enabled	Boolean	True if the Alarm is Enabled
HelpText	String	
AckRequired	Boolean	Acknowledged Required Flag
Source	String	Identifier for the alarm owner
Condition	String	Identifier for the alarm condition
Group	String	Identifier for the native alarm group
ChangeTime	Time	Time Alarm last changed state
Comment	String	Any user-defined alarm comment
RepeatCount	Integer	Number of set/clears since last ack

- **Alarm Categories**

Each alarm record is associated with one Alarm Category. The alarm category determines the possible alarms for a source. For example:

Alarm records with a category of “Process_Inputs” might include the following alarms:

- Bad input device
- High alarm limit
- Low alarm limit

Alarm records with a category of “System_Status” might include the following alarms:

- Disk space is low
- Virtual memory usage is too high
- CPU usage is high
- Fan has failed

7.10.7.2 Alarm Service

From a components perspective there are just two APIs:

- `SetAlarm(Category, Condition, AlarmValue);`
  This method is used to set an alarm. The service will automatically add the source name.

- `ClearAlarm(Category, Condition);`
  This function is used to clear an alarm.

In order to offer filtered lists a client made need to retrieve information as to what categories, conditions and areas have been defined. The following methods allow clients to get this information:

- `QueryCategories`
  The `QueryCategories` method gives clients a means of finding out the specific categories of alarms supported by a given server. This method would typically be invoked prior to specifying a filter.

- `QueryAreas`
  The `QueryAreas` method gives clients a means of finding out the specific areas supported by a given server. This method would typically be invoked prior to specifying a filter.

- `QueryConditionNames`
  The `QueryConditionNames` method gives clients a means of finding out the specific condition names which the alarm server supports for the specified category. This method would typically be invoked prior to specifying an alarm filter. Condition names are server specific.

- `QuerySourceConditions`
  The `QuerySourceConditions` method gives clients a means of finding out the specific condition names associated with the specified source.

- `QueryAttributes`
  This may or may not be needed. Most likely we just define a set of standard attributes and will not need user-defined ones. Using the categories returned by the `QueryCategories` method, client applications can invoke the `QueryAttributes` method to get information about the vendor-specific attributes the server can provide as part of a notification for an alarm within the specified category. Simple servers may not support *any* vendor-specific attributes for some or even all categories.

7.10.7.3 API Summary

API	**Location**	**Usage**
SetAlarm	Service	Component Calls
ClearAlarm	Service	Component Calls

API	**Location**	**Usage**
EnableConditionByArea	Alarm Manager	Manager Implements
EnableConditionBySource	Alarm Manager	Manager Implements
DisableConditionByArea	Alarm Manager	Manager Implements
DisableConditionBySource	Alarm Manager	Manager Implements
CreateSubscription	Alarm Manager	Manager Implements
AckCondition	Alarm Manager	Manager Implements
Refresh	Alarm Manager	Manager Implements
CancelRefresh	Alarm Manager	Manager Implements

OnAlarm	Alarm Manager	Manager Calls

API	**Location**	**Usage**
QueryCategories	Service	Clients (UI) Calls
QueryConditionNames	Service	Clients (UI) Calls
QuerySourceConditions	Service	Clients (UI) Calls
EnableConditionByArea	Alarm Manager	Clients (UI) Calls
EnableConditionBySource	Alarm Manager	Clients (UI) Calls
DisableConditionByArea	Alarm Manager	Clients (UI) Calls
DisableConditionBySource	Alarm Manager	Clients (UI) Calls
CreateSubscription	Alarm Manager	Clients (UI) Calls
AckCondition	Alarm Manager	Clients (UI) Calls
Refresh	Alarm Manager	Clients (UI) Calls
CancelRefresh	Alarm Manager	Clients (UI) Calls
OnAlarm	Client	Clients Implements (Callback)

7.10.7.4 Alarm Manager

Alarm events can occur as the result of a set alarm, clear alarm or refresh operation. In all cases the data structure that is sent is the same. A proposed data structure modeled after the OPC `ONALARMSTRUCT` is described below.

Member	**Description**
szSource	The source of the alarm notification.
ftTime	Time of the occurrence - for conditions, time that the condition transitioned into the new state or sub-condition. For example, if the alarm notification is for acknowledgment of a condition, this would be the time that the condition became acknowledged.
szMessage	Notification message describing the alarm.
dwEventType	OPC_SIMPLE_EVENT, OPC_CONDITION_EVENT, or OPC_TRACKING_EVENT for Simple, Condition-Related, or Tracking events, respectively.
dwEventCategory	Standard and Vendor-specific category codes.
dwSeverity	Alarm severity (0..1000).
	The following items are present only for Condition-Related Events (see dwEventType)
szConditionName	The name of the condition related to this alarm notification.

wChangeMask	Indicates to the client which properties of the condition have changed, to have caused the server to send the alarm notification. It may have one or more of the following values: OPC_CHANGE_ACTIVE_STATE OPC_CHANGE_ACK_STATE OPC_CHANGE_ENABLE_STATE OPC_CHANGE_QUALITY OPC_CHANGE_SEVERITY OPC_CHANGE_SUBCONDITION OPC_CHANGE_MESSAGE OPC_CHANGE_ATTRIBUTE If the alarm notification is the result of a Refresh, these bits are to be ignored. For a “new alarm”, OPC_CHANGE_ACTIVE_STATE is the only bit which will always be set. Other values are server specific. (A “new alarm” is any alarm resulting from the related condition leaving the Inactive and Acknowledged state.)
wNewState	A WORD bit mask of three bits specifying the new state of the condition: OPC_CONDITION_ACTIVE, OPC_CONDITION_ENABLED, OPC_CONDITION_ACKED.
wQuality	Quality associated with the condition state.
bAckRequired	This flag indicates that the related condition requires acknowledgment of this alarm. The determination of those alarms which require acknowledgment is server specific. For example, transition into a LimitAlarm condition would likely require an acknowledgment, while the alarm notification of the resulting acknowledgment would likely not require an acknowledgment.
ftActiveTime	Time that the condition became active (for single-state conditions), or the time of the transition into the current sub-condition (for multi-state conditions). This time is used by the client when acknowledging the condition
dwCookie	Server defined cookie associated with the alarm notification. This value is used by the client when acknowledging the condition. This value is opaque to the client.
	The following is used only for Tracking Events and for Condition-Related Events which are acknowledgment notifications (see dwEventType).

szActorID	For tracking events, this is the actor ID for the event notification. For condition-related events, this is the AcknowledgerID when OPC_CONDITION_ACKED is set in wNewState. If the AcknowledgerID is a NULL string, the event was automatically acknowledged by the server. For other events, the value is a pointer to a NULL string.

The Alarm Manager maintains the alarm information it receives from the framework via the `SetAlarm` and `ClearAlarm` APIs. It will propagate these alarms to the client based on the client provided filter (optional implementation) and based on whether or not an area or category is enabled/disabled. The Alarm Manager responds to alarm acknowledgements and will remove the alarms as they are cleared and acknowledged.

7.10.7.4.1 EnableConditionByArea

Places all conditions for all sources within the specified process areas into the enabled state. Therefore, the server will now generate condition-related alarms for these conditions. The effect of this method is global within the scope of the alarm manager. Therefore, if the manager is supporting multiple clients, the conditions are enabled for all clients, and they will begin receiving the associated condition-related alarms.

* Will need to decide if this needs to propagate to each alarm service or if it is just applied to the Alarm Manager.

7.10.7.4.2 EnableConditionBySource

Places all conditions for the specified sources into the enabled state. Therefore, the alarm manager will now generate condition-related alarms for these conditions. The effect of this method is global within the scope of the alarm manager. Therefore, if the alarm manager is supporting multiple clients, the conditions are enabled for all clients, and they will begin receiving the associated condition-related alarms.

* Will need to decide if this needs to propagate to each alarm service or if it is just applied to the Alarm Manager.

7.10.7.4.3 DisableConditionByArea

Places all conditions for all sources within the specified process areas into the disabled state. Therefore, the alarm manager will now cease generating condition-related alarms for these conditions. The effect of this method is global within the scope of the alarm manager. Therefore, if the alarm manager is supporting multiple clients, the conditions are disabled for all clients, and they will stop receiving the associated condition-related alarms.

* Will need to decide if this needs to propagate to each alarm service or if it is just applied to the Alarm Manager.

7.10.7.4.4 DisableConditionBySource

Places all conditions for the specified sources into the disabled state. Therefore, the alarm manager will no longer generate condition-related alarms for these conditions. The effect of this method is global within the scope of the alarm manager. Therefore, if the alarm manager is supporting multiple clients, the conditions are disabled for all clients, and they will stop receiving the associated condition-related alarms.

* Will need to decide if this needs to propagate to each alarm service or if it is just applied to the Alarm Manager.

7.10.7.4.5 Create Subscription

This allows a client (for example, UI or logging service) to connect to the Alarm Manager and to start receiving alarms. Alarms are sent to the client(s) via an OnAlarm callback.

7.10.7.4.6 AckCondition

The client uses the `AckCondition` method to acknowledge one or more conditions in the Alarm Manager. The client receives alarms notifications from conditions via the `OnAlarm` callback. This `AckCondition` method specifically acknowledges the condition becoming active. One or more conditions belong to a specific source – the source of the alarm notification. For each condition-related alarm notification, the corresponding Source, Condition Name, Active Time and Cookie is received by the client as part of the `OnAlarm` callback parameters.

7.10.7.4.7 Refresh

Force a refresh for all active conditions and inactive, unacknowledged conditions whose alarms notifications match the filter of the subscription. Clients will often need to get the current condition information from the alarm manager, particularly at client startup, for things such as a current alarm summary. The Alarm Manager supports this requirement by resending the most recent alarm notifications which satisfy the filter in the alarm subscription and which are related to active and/or unacknowledged conditions. The client can then derive the current condition status from the “refreshed” alarm notifications.

7.10.7.4.8 CancelRefresh

Cancels a refresh in progress for the alarm subscription. If a refresh is in progress, the alarm manager will send one final callback with the last refresh flag set and the number of alarms equal to zero.

7.10.7.4.9 OnAlarm

The Alarm Manager invokes the `OnAlarm` method to notify the client of alarms which satisfy the filter criteria for the particular alarm subscription. Note that callbacks can occur for two reasons: alarm notification or refresh. A client can determine the ‘cause’ of a particular callback by examining the `bRefresh` parameter in the `OnAlarm` callback.

7.10.7.5 Alarm Summary Display

The Alarm Summary displayed is discussed previously. This display allows alarms to be viewed, sorted, filter and acknowledged.

7.10.7.6 Alarm Logging

All alarms can be logged to a database. It is expected that there will be an Alarm Logging Display that can be used to view all recent alarm state changes and events. This display may be used to query the alarm history files in many ways. The alarms may be logged by the Alarm Manager or there may be a separate system wide component that is responsible for monitoring and logging alarms.

7.10.7.6.1 Alarm Table

The following shows a possible schema for the alarm table:

- **TimeStamp**: UT time when the alarm state change or event occurred.
- **Category**: Alarm category for this alarm. Alarm categories are defined by the alarm system configuration and are meant to provide logical alarm groupings.
- **Source**: Owner of the alarm or event.
- **Alarm Text**: Description of the alarm or event.
- **Action**: A classification of the alarm state change or event. Typical actions are Set Alarm, Clear Alarm, or Acknowledge Alarm.
- **Value**: This field may be used to save the value of some key variable associated with the alarm or event. Any value shown represents the value at the time of the alarm or event occurrence.
- **Operator**: May be used to record the operator performing the event or action.
- **Severity**: The severity (or priority) of the alarm or event. This may range from 1 to 1,000. Low values indicate a low urgency and high value represent a higher urgency.

7.10.8 Configuration

Most classes within the NGAO system will have some set of configurable properties associated with each instance. When objects are instantiated these properties and attributes will be undefined, and will need to be set for the object before standard operations can begin. A convenient way to store and retrieve object properties is through a Configuration Database. This database would define all of the configuration information required by each class instance, as well as any additional metadata and run-time information needed by the system.

7.10.8.1 Database Schema

The database schema is divided into three sets: class definitions, instances, and management. Class definitions constitute all of the relational tables that define the structure, properties, and default values of KCSF classes. Every component in the system that utilizes configuration must have a corresponding class definition in the database. As classes are added to the KCSF infrastructure new definitions must be created to map configuration attributes to code. The following schema defines class structure within the database, and will be used to generate instances.

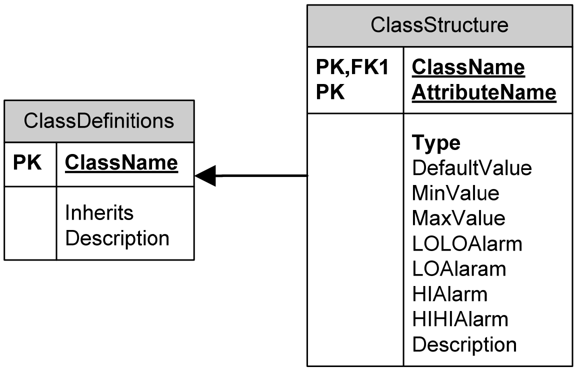


Figure 50: Class Schema.

Instances cover all tables and properties that represent application components and define their configuration state. Every instance of a KCSF component that utilizes configuration will have a dedicated database instance. All instances in the database are mapped to one of the database class definitions. At run-time KCSF components will query the database for their configuration to obtain their initial state and properties. The following diagram details the hierarchical relationship of instance items in the database.

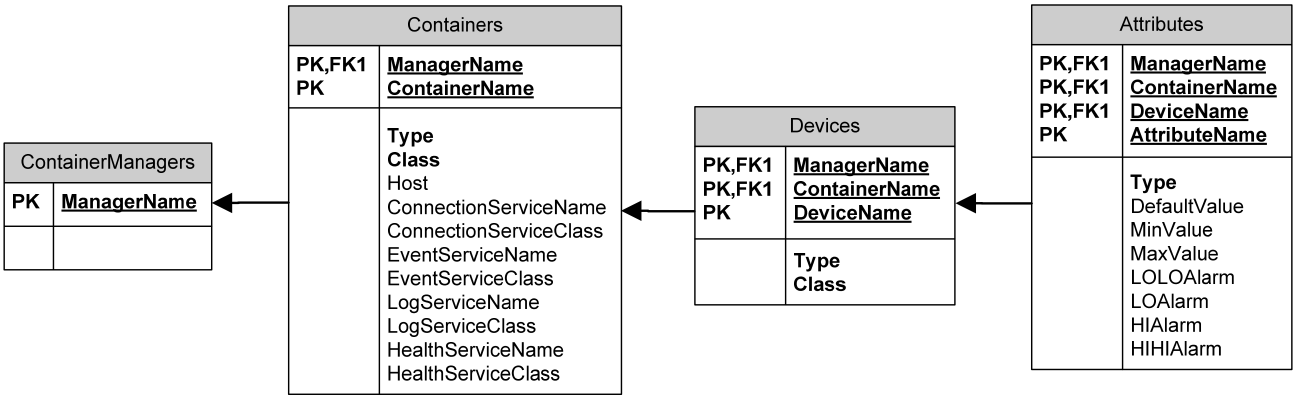


Figure 51: Instance Schema.

The final set of tables is responsible for the management, versioning, and administrative requirements of the system. These tables relate primarily to the general requirements of maintaining a database.

7.10.8.2 Versioning

The database model will be designed to provide versioning capabilities for configuration information. Versioning allows developers to take a snapshot of the active database and preserve it in a read-only format, making it available for restoration.

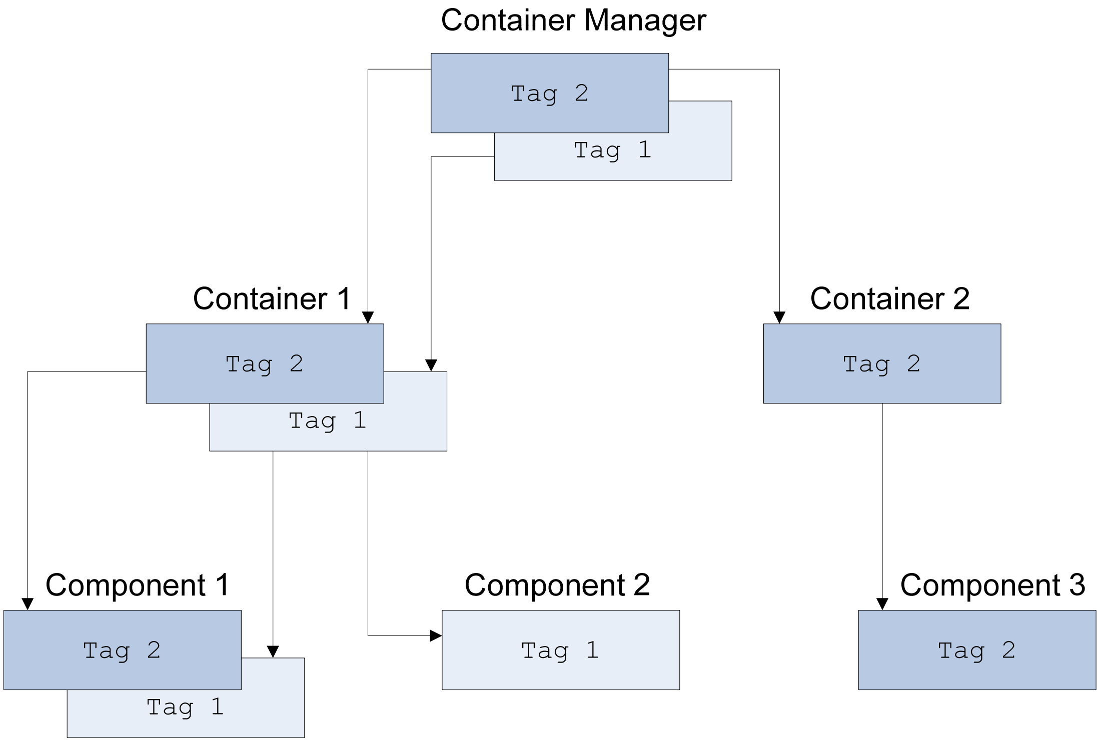


Figure 52: Data Versioning.

Figure 52 above shows a versioning example for a simple KCSF system. The system is comprised of a container manager, containers, and components. Two versions of the system exist, identified by unique version ids: '*Tag 1*' and '*Tag 2*'.

Tag 1 represents an early version of the system. In this version there was a single container that managed two components (*Component 1* and *Component 2*). At some point the developers decided that it was necessary to make modifications to the design. The current database configuration was tagged, and the developers started with the modifications.

A new component (*Component 3*) needed to be added to the system, and based on resources it was decided a separate container would be used to manage this component. After a design review the developers realized that they could combine the functionality of Component 1 and Component 2, simplifying the overall system communication. The product of their upgrade work can be seen in the database layout identified by '*Tag 2*'. In this version, a new container and component ('*Container 2*' and '*Component 3*') were added to the system. Component 2 being obsolete is no longer part of the second version, and is not referenced by Container 1.

7.10.8.3 Tagging the Database

At some point in the development and testing of a system the configuration data will need to be preserved. This process is known as tagging, and is responsible for taking a snapshot of the active database and associating it with a user defined ID. The database allows users to create new tags or overwrite old tags. When an old tag is overwritten, all of the items' original configuration values that are affected by the tag process will be lost (retrievable only by restoring the database from a previous backup). Each unique tag operation creates a new version in the database. The version is a complete duplication of the database, and is maintained as read-only.

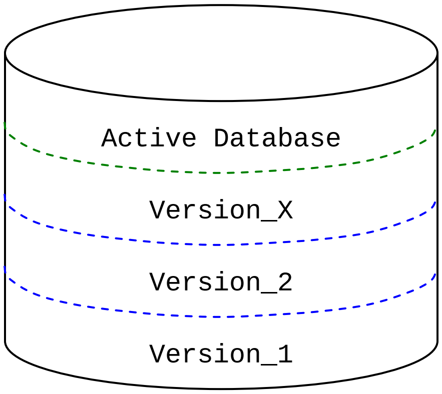


Figure 53: Database Versions.

A database can hold multiple versions simultaneously making it easier and faster to revert to a previously tested version. Although you can not modify the state of a version directly, you can overwrite an existing version, completely replacing it in the database.

7.11 Scripting Support

Direct scripting support will be provided by the framework by utilizing the standard Java Scripting API. The scripting API consists of interfaces and classes that define Java TM Scripting Engines and provides a framework for their use in Java applications. This API is intended for use by application programmers who wish to execute programs written in scripting languages in their Java applications. The scripting language programs are usually provided by the end-users of the applications. The API is available through the javax.script package. The main areas of functionality of javax.script package include

- **Script execution**: Scripts are streams of characters used as sources for programs executed by script engines. Script execution uses eval methods of ScriptEngine and methods of the Invocable interface.
- **Binding**: This facility allows Java objects to be exposed to script programs as named variables. Bindings and ScriptContext classes are used for this purpose.
- **Compilation**: This functionality allows the intermediate code generated by the front-end of a script engine to be stored and executed repeatedly. This benefits applications that execute the same script multiple times. These applications can gain efficiency since the engines' front-ends only need to execute once per script rather than once per script execution. Note that this functionality is optional and script engines may choose not to implement it. Callers need to check for availability of the Compilable interface using an instanceof check.
- **Invocation**: This functionality allows the reuse of intermediate code generated by a script engine's front-end. Whereas Compilation allows entire scripts represented by intermediate code to be re-executed, Invocation functionality allows individual procedures/methods in the scripts to be re-executed. As in the case with compilation, not

- **Script engine discovery and Metadata**: Applications written to the Scripting API might have specific requirements on script engines. Some may require a specific scripting language and/or version while others may require a specific implementation engine and/or version. Script engines are packaged in a specified way so that engines can be discovered at runtime and queried for attributes. The Engine discovery mechanism is based on the Service discovery mechanism described in the Jar File Specification. Script engine implementing classes are packaged in jar files that include a text resource named META-INF/services/javax.script.ScriptEngineFactory. This resource must include a line for each ScriptEngineFactory that is packaged in the jar file. ScriptEngineManager includes getEngineFactories method to get all ScriptEngineFactory instances discovered using this mechanism. ScriptEngineFactory has methods to query attributes about script engine.

By providing components with the ability to directly support scripting the end users and application developers can access and control KCSF devices and controllers from a scripting environment, using the language of their choice (KCSF will support a number of different scripting languages) .

7.11.1 Implementation Details

There are a number of ways to provide scripting support from Java, the two most consistent approaches that will cover a large number of scripting languages are:

- Using shells and external wrappers
- Using Javax Scripting

Either approach allows scripting support to be added to KCF in a very generic way. The architecture is designed to allow for executing disparate scripts, and nothing in the programming model depends on a specific scripting language. In either case the full Java library as well as whatever parts of KCF we choose to expose are fully available to the scripts.

7.11.1.1 Using shells and external wrappers

This is the approach ATST CSF is currently using. It currently supports Groovy, bsh and Python. Generic IInterpreter and IExecutor interfaces are defined to set and get script parameters and evaluate or execute script code. A concrete implementation of these is provided for each scripting language supported. This is shown in Figure 54 below. The concrete implementation itself wraps a shell or interpreter for that language available as a 3rd party library. The factory classes can decide which script specific executor or interpreter to invoke based on file extension or other means.

There are differences between a script Executor and a script Interpreter:

- An Executor can parse a script and report errors before running.
- An Interpreter can execute a script in fragments, identifying parse errors fragment by fragment.

- An Executor runs a script as a separate entity — log messages, events, etc. appear to come directly from the script.
- With an Interpreter, log messages, events, etc. appear to come from the surrounding application.

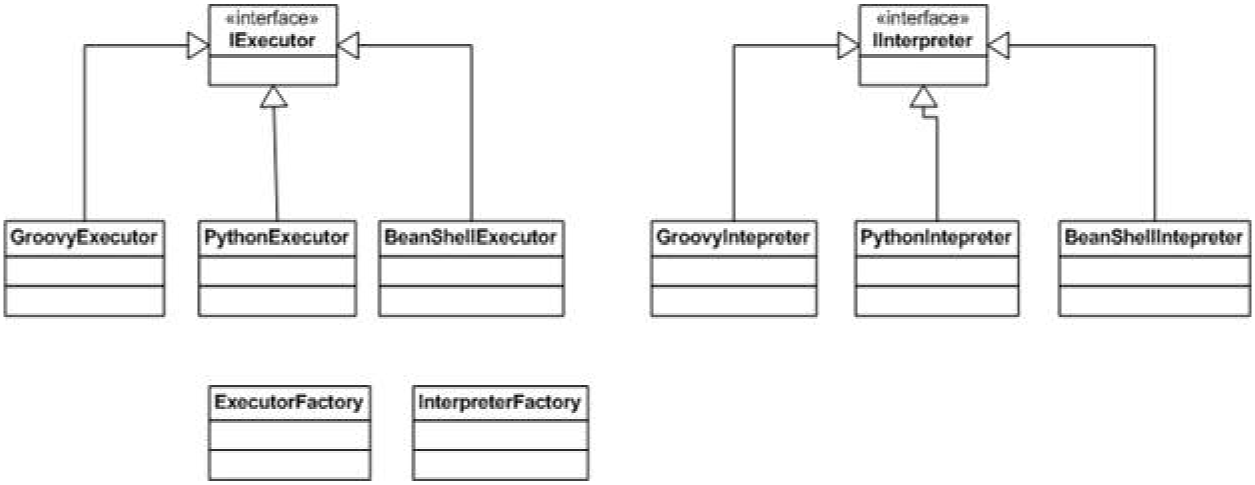


Figure 54: The `IExecutor` and `IInterpreter` scripting classes.

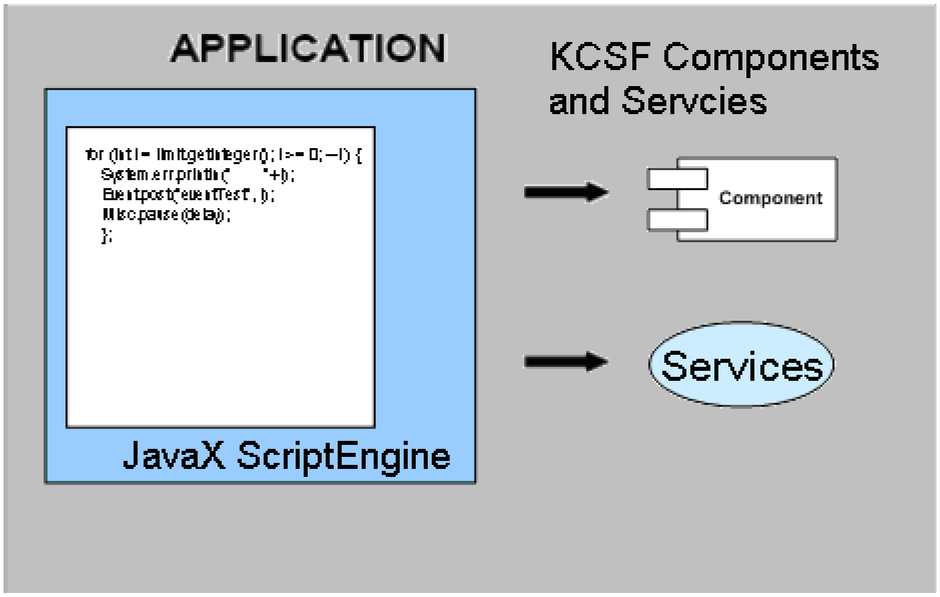
The following are short examples of script usage:

```
for (int i = limit.getInteger(); i >= 0; --i ) {
    System.err.println("        "+i);
    Event.post("eventTest", i);
    Misc.pause(delay);
    };

for i in range(limit.getInteger(),-1,-1):
    print '\t'+`i`
    Event.post('eventTest', i)
    Misc.pause(delay)
```

7.11.1.2 Using Javax.Scripting

Since Java 6, the Java now comes with a framework that abstracts various scripting engines and thus creates general scripting support for Java applications. This support is available through the Scripting API (javax.script). It enables easy engine registration, instantiating engines through factory methods and sharing the context between them. Essentially it provides a pluggable framework for third-party script engines. This concept is shown in There are a large number of scripting engines now available that are compatible with this API, by default the package comes with JScript enabled.



```
for (int i = lim.getInteger(); i >= 0; --i) {
  System.err.println("      "+i);
  Event.post("eventTest", i);
  Misc.pause(delay);
  };
```

Figure 55: Multiple script engines are supported through the Java Scripting API.

Getting started with scripting is easy. The starting point is always the ScriptEngineManager class. A ScriptEngineManager object can tell you what script engines are available to the Java Runtime Environment (JRE). It can also provide ScriptEngine objects that interpret scripts written in a specific scripting language. The simplest way to use this API is to do the following:

- Create a `ScriptEngineManager` object.
- Retrieve a `ScriptEngine` object from the manager.
- Optionally add script variables
- Evaluate a script using the `ScriptEngine` object.

The following code example performs all three steps, printing `Hello, world!` to the console.

```
ScriptEngineManager mgr = new ScriptEngineManager();
ScriptEngine jsEngine = mgr.getEngineByName("JavaScript");
try {
    jsEngine.eval("print('Hello, world!')");
} catch (ScriptException ex) {
    ex.printStackTRace();
}
```

The API is only slightly more complex if you want to query the list of supported scripting engines, to pass values back and forth to the scripting environment, or to compile a script for repeated execution. Additional APIs allow you to query the `ScriptEngineManager` for engines that associate a particular file extension, to execute the script from a file, and to call a specific function in a script.

Conceptually the API is small and simple

- ScriptEngine
    - Components that execute scripts.
- ScriptEngineManager

    - Used by host application to locate and instantiate ScriptEngines.
- ScriptContext/Bindings
    - Provide view of host application to ScriptEngine
    - Script variables<====>Application objects.

7.12 User Interfaces

The goal of the framework is to provide a user interface solution that interacts well with the framework, is full featured, promotes UI widget reuse, has a common look and feel, and can meet the needs of the observatory. That said, working with the end user to establish UI layouts, workflow and other operational issues such as how sorting, selection, tooltips etc., should work is perhaps more important that any technical implementation. It is expected that significant work will go into working with the OAs, SAs and others to iterate over UI prototypes.

Regardless of the UI technical solution the framework will be developed to ensure that there is a clear division between domain objects that model our perception of the real world, and presentation objects that are the GUI elements we see on the screen. Domain objects should be completely self contained and work without reference to the presentation; they should also be able to support multiple presentations, possibly simultaneously. The basic idea in all cases is that user gestures are handed off by the widgets to a controller, changes in the domain model are coordinated separately from the UI, and finally the UI is updated from the model as needed. There are a number of design patterns that support this which have evolved from the “Model View Controller” pattern, namely “Model View Presenter” and “Supervising Controller” with “Passive View”.

There are currently a number of technical solutions identified that need to be explored further.

7.12.1 Java Swing

MAGIQ has shown that feature rich performant UIs can be written in Swing and easily and quickly adapted to user needs. MAGIQ used presentation separation and a number of free open source third party libraries, namely:

- JAI – Java Advanced Imaging
- InfoNode – For docking windows
- JFreeChart – For graphs
- Nom Tam – For FITS processing

Responsiveness, as defined by Jeff Johnson in his book *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers,* is "the software's ability to keep up with users and not make them wait." Responsiveness is often cited as the strongest factor in users' satisfaction with software applications. Poor responsiveness can render an otherwise well-designed application unusable.

Poorly-written Swing applications can appear to freeze when time-consuming code is executing. This unresponsiveness happens because all Swing applications handle events on the event dispatch thread. This means painting, layout updates, button presses, rollover effects, and Swing component interactions. When time-consuming code is executed on the event dispatch thread, all other-event driven activity waits and the application appears frozen.

Within any Swing application there are essentially two factors that affect the user's perception of responsiveness. These are the swing single threaded model as described above and graphic performance. How these potential issues are addressed is described below.

7.12.1.1 Concurrency

When a user interacts with Swing components, whether it is clicking on a button or resizing a window, the Swing toolkit generates event objects that contain relevant event information. The event objects are then placed onto a single event queue ordered by their entry time. While that happens, a separate thread, called the event-dispatch thread, regularly checks the event queue's state. As long as the event queue is not empty, the event-dispatch thread takes event objects from the queue one by one and sends them to the interested parties. Finally, the interested parties react to the event notification by processing logic such as event handling or component painting, as shown in Figure 56.

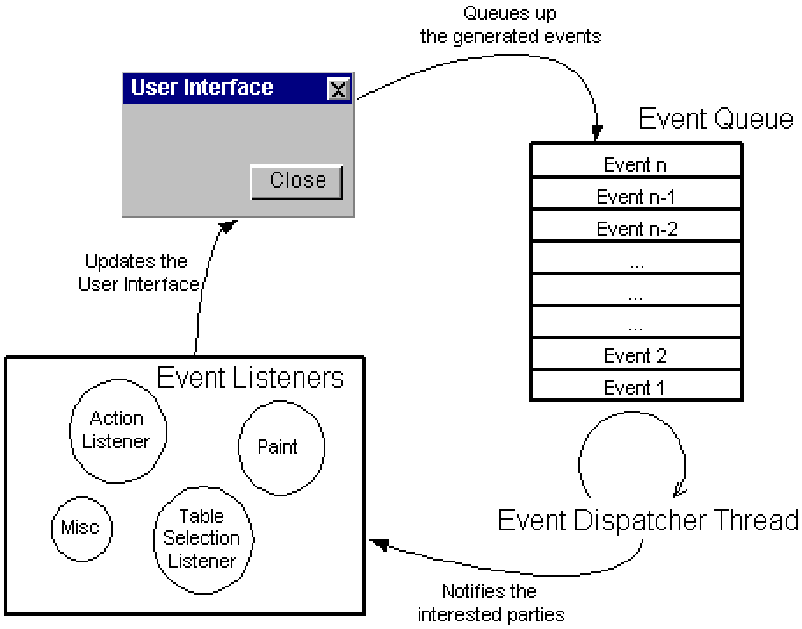


Figure 56 Swing event-dispatch model

Since the event-dispatch thread executes all event-processing logic sequentially, it avoids undesirable situations such as painting a component whose model state is partially updated. With the exception of a very small number of thread-safe methods all SWING code <u>**must**</u> execute on the `EventDispatcher` thread. The down side to this is that any operation that takes a significant amount of time to cause the event queue to backup and the UI to appear frozen.

For UI applications this has a number of consequences:

- In order to keep the UI responsive, any potential long running tasks must execute in their own thread.
- The results of long running tasks need to make their way back onto the event dispatch thread.

- Various callbacks that change the model state happen on threads that originate outside the UI, examples include application manager callbacks for image update events, telemetry updates etc. If Swing components, or other non thread safe entities, need to be updated as part of these callbacks they cannot be updated directly and the updates need to be moved to the event dispatch thread.

In summary when writing a multi-threaded application using Swing, there are two constraints to keep in mind:

- Time-consuming tasks should not be run on the Event Dispatch Thread. Otherwise the application becomes unresponsive, as described above.
- Swing components should be accessed on the Event Dispatch Thread only since the components are not thread safe.

Java 6 provides two classes that help simplify concurrent operations by providing:

- A way to ensure that any threaded code will run on the event-dispatch thread (SwingUtilities)
- An easy way to create background threads that can interact with the UI (SwingWorker)

7.12.1.2 SwingUtilities

In Swing programs, the initial threads don't have a lot to do. Their most essential job is to create a Runnable object that initializes the GUI and schedule that object for execution on the event dispatch thread. Once the GUI is created, the program is primarily driven by GUI events, each of which causes the execution of a short task on the event dispatch thread. The SwingUtilities `invokeLater` method is used to bootstrap the UI onto the event dispatch thread.

Application code can schedule additional tasks on the event dispatch thread (if they complete quickly, so as not to interfere with event processing) using the SwingUtilities class. SwingUtilities has two methods related to threads these are `invokeLater` and `invokeAndWait`. The `invokeLater` method creates a special event that wraps around the runnable object and places it on the event queue. When the event-dispatch thread processes that special event, it invokes the runnable object's run method in the same thread context, thus preserving thread safety. `invokeAndWait` does the same things but waits for the event to complete.

7.12.1.3 SwingWorker

The SWING constraints mean that a GUI application with time intensive computing or methods that could block needs at least two threads:

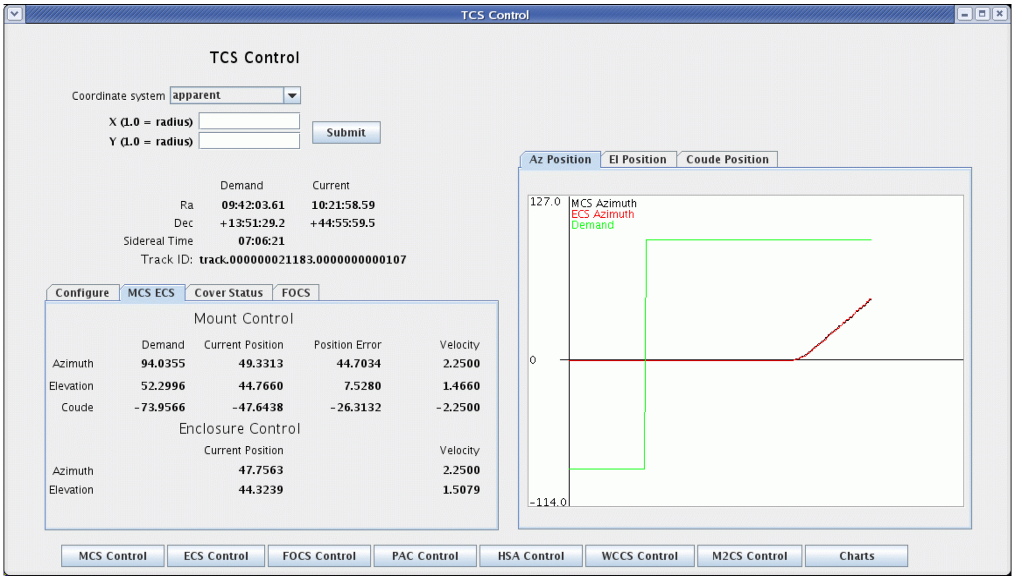
- A thread to perform the lengthy task
- The Event Dispatch Thread (EDT) for all GUI-related activities.

SwingWorker is designed for situations where you need to have a long running task run in a background thread and provide updates to the UI either when done, or while processing.

MAGIQ has made significant use of the SwingWorker and SwingUtilities to ensure robust and performant user interfaces.

7.12.2 CSF Java Engineering Screens

CSF has the start of a UI framework that came from Observatory Science Laboratories (OSL). It is based on Swing and is called Java Engineering Screens (JES). This has some potential at least for engineering screens (basically DM replacements). Screen layout and composite are maintained in an XML file. JES supports an Edit and Execute mode. These can be toggle on the fly. In edit mode widgets can be added to a screen and repositioned as needed and then the UI can be executed. A sample image is shown below.



TCS Control

Coordinate system: apparent

X (1.0 = radius)

Y (1.0 = radius)

	Demand	Current
Ra	09:42:03.61	10:21:58.59
Dec	+13:51:29.2	+44:55:59.5
Sidereal Time	07:06:21	

Track ID: track.000000021183.0000000000107

Mount Control

	Demand	Current Position	Position Error	Velocity
Azimuth	94.0355	49.3313	44.7034	2.2500
Elevation	52.2996	44.7660	7.5280	1.4660
Coude	-73.9566	-47.6438	-26.3132	-2.2500

Enclosure Control

	Current Position	Velocity
Azimuth	47.7563	2.2500
Elevation	44.3239	1.5079

Figure 57: A sample CSF JES GUI.

The basic architecture is as follows:

- `JesManager` – This is a CSF component and is deployed and managed the same as any other component. It is used to control and manage all JES Screens and event subscriptions
- `JesScreen` – maintains all widgets for a screen, tells widgets if they should be in edit or execute mode, allows pasting of widgets, setting the screen title, saving layouts and widget grouping
- `JesWidget` – base type for all widgets, supports events, get/set etc. This is derived from Swings JPanel.
- Widgets – numerous widgets based on Swing and derived from `JesWidget` such as comboBox, tables, lists, graphs etc.

7.12.3 GTK+

Unlike Swing and JES which are Java based, GTK+ is an open source GUI toolkit with cross platform support including Linux, Unix, Windows, and Mac OS X. It was used for the GNOME desktop and GIMP image processing program and is licensed under the GNU LGLP 2.1

GTK+ is a highly usable, feature rich toolkit for creating graphical user interfaces which boasts cross platform compatibility and an easy to use API. GTK+ is written in C, but has bindings to many other popular programming languages such as C++, Java, and Python and C# among others. The GNU LGLP 2.1 license allows development of both free and proprietary software with GTK+ without any license fees or royalties. GTK+ is most notably used in GIMP, VMware Workstation, GNOME and Gnumeric.

GTK+ has been around a long time and has been developed for over a decade. It is supported by a large community of developers and has core maintainers from companies such as Red Hat and Novell.

Key features for us are:

- Native look and feel
- Theme support
- Thread safe
- Object orientated approach
- Documentation

GTK+ has not yet been evaluated and there is no direct in house experience with it.

7.12.4 Qt

Qt is another cross-platform application development framework, widely used for the development of UI programs, and also used for developing non-UI programs such as console tools and servers. Qt is most notably used in KDE, Google Earth, Skype, Qt Extended and Adobe Photoshop Album. It is produced by Nokia's Qt Software division.

Qt uses C++ with several non-standard extensions implemented by an additional pre-processor that generates standard C++ code before compilation. Qt can also be used in several other programming languages such as Java and Python via language bindings. It runs on all major platforms. Non-GUI features include SQL database access, XML parsing, thread management, network support and a unified cross-platform API for file handling. It is distributed under the terms of the GNU Lesser General Public License (among others), Qt is free and open source software. An example of a Qt GUI screen is shown below.

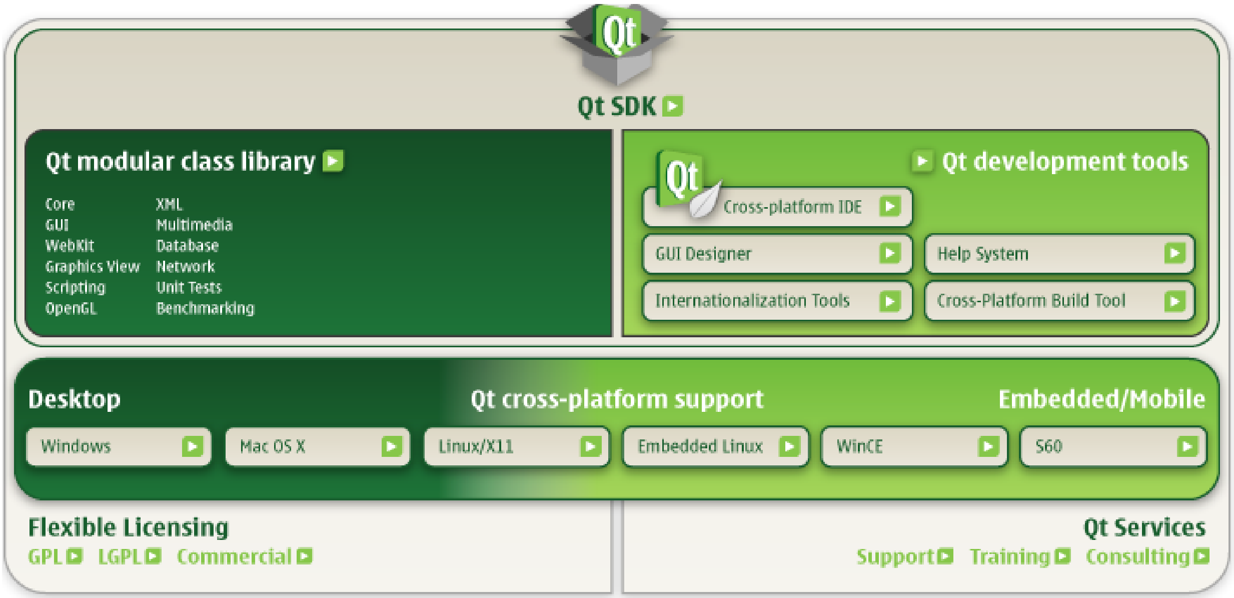


Figure 58: An example Qt GUI screen.

Qt is nice in that it has a GUI designer application. There is also a Qt plug-in available for integrating Qt into the eclipse development. Some quick testing showed that this is an easy feature to get started with.

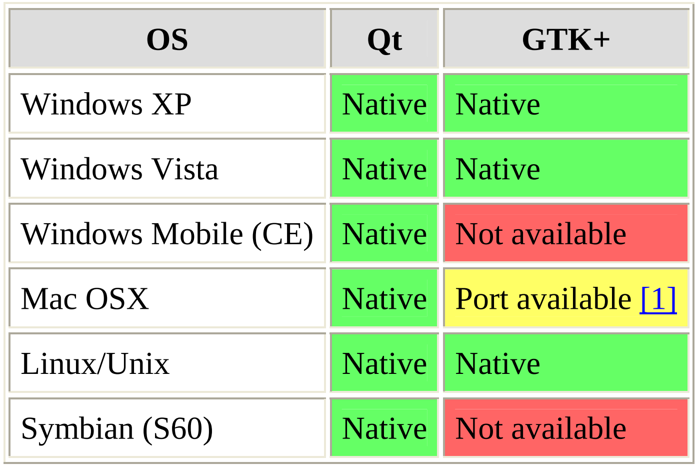
Qt provides the building blocks — a broad set of customizable widgets, graphics canvas, style engine and more — that are needed to build modern user interfaces. It allows easy incorporation of 3D graphics, multimedia audio or video, visual effects, animations and custom styles The Qt Creator cross-platform IDE is expected to make it fast to learn and easy to use. QT can be scripted using JavaScript.

7.12.4.1 QT and GTK+ Comparison

GTK+ and Qt are both open-source cross-platform User Interface toolkits and development frameworks. These are the two most popular frameworks in use for Linux and BSD because they are open-source and give developers a powerful toolkit to design Graphical User Interfaces. GTK+ is used as the standard toolkit for the GNOME and Xfce Desktop Environments while Qt is used for KDE.

Qt is developed by Qt Software, a division of Nokia. GTK+ is developed and maintained by the GNOME Foundation. Qt's API is said to be cleaner and to have better documentation than GTK+'s. Qt comes with Qt Designer, a tool that allows easy layout of widgets and simple connections of slots and signals between widgets, QTK doesn't directly support a builder.

Both Qt and GTK+ were developed from the ground up with Object Oriented Programming in mind. Qt is developed in C++, GTK+ in C in an object oriented manner using the GObject type system. Qt is a complete consistent framework. You can easily connect HTTP events to GUI elements, fill forms with results from a database query or build an interactive visualization of large datasets. GTK+ is only a GUI toolkit. Both Qt and GTK+ are available on most popular desktop Operating Systems. For mobile devices, Qt for Embedded allows Qt to run directly on the hardware, without the need of X11 or a window manager. The first Qt application started becomes the window manager featuring full composition and top-level transparency. GTK+ on embedded devices (e.g. Maemo) requires an X11 server + window manager, resulting in at least three processes running for a hello world application.



OS	Qt	GTK+
Windows XP	Native	Native
Windows Vista	Native	Native
Windows Mobile (CE)	Native	Not available
Mac OSX	Native	Port available [1]
Linux/Unix	Native	Native
Symbian (S60)	Native	Not available

Figure 59: Comparison of Qt and GTK+ window environments.

Qt looks more native than GTK+ on Windows and Mac platforms. This is because Qt tries to use native widgets whenever possible. Even so, neither Qt nor GTK+ will look and feel completely native on Windows or Mac. On the other hand GTK+ brings more consistent user experience when switching from one platform to another, since the look and feel remains unchanged. Natively, Qt has C++ based libraries. It also supports Java, Perl, Python, PHP, and Ruby based development. Qt also ships with the embedded scripting language QtScript, which is an ECMA-Script (JavaScript) implementation. Natively, GTK+ has C based libraries. It supports several languages like for example C++, Java, Perl, Python, PHP, Ruby, and Mono/C#. QT looks like a more modern full featured solution and will be explored first.

8 Glossary

ACS: The ALMA Common Software, a software framework based on CORBA.

ALMA: Atacama Large Millimeter Array.

API: Application Programming Interface.

ASKAP: Australian Square Kilometre Array Pathfinder.

ATST: Advanced Technology Solar Telescope.

ATST CSF: ATST Common Services Framework.

Attribute: A name and value pair used to represent a parameter in a distributed system.

CamelCase notation: A common computer notation where compound words are formed by combining the element words without spaces, with each element's initial letter capitalized. The first letter of the entire compound word may be either upper case or lower case. Typically upper case first letters are used to designate class names. Example: `getUserName()`.

COM: Component Object Model, a binary interface standard for software componentry introduced by Microsoft in 1993.

Component: A self-contained piece of software with a well-defined interface or set of interfaces.

Component-based development: A software architecture and development model that emphasis the decomposition of a distributed system into abstract functional and logical components with well defined interfaces.

Composition: An object oriented design term describing the inclusion of multiple classes in the definition of a new class. The new class is said to be "composed" of the constituent classes.

Container component model: A development model that groups software entities into related sets and provides a distinct separation between the technical and functional tasks of a system through the use of containers and components.

CORBA: Common Object Request Broker Architecture, an object oriented middleware standard defined by the Object Management Group.

DBMS: Database Management System.

DDS: Data Distribution Service, a data-centric middleware based on a publish-subscribe paradigm.

Dependency injection: A type of inversion of control where high level modules are dependent on abstract interfaces, and not low level module implementations. In this way the actual implementation (objects) can be assigned (injected) into the application module.

Encapsulation: The hiding of the internal mechanisms and data structures of a software component behind a simple well-defined interface.

EPICS: Experimental Physics and Industrial Control System, a middleware commonly used to implement control systems for several high energy physics accelerators and astronomical observatories.

Framework: an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code providing specific functionality. Frameworks are similar to software libraries in that they are reusable abstractions of code wrapped in a well-defined API. (From Wikipedia).

Fully qualified name: A name that uniquely identifies a component's attributes across an entire distributed system.

GPL: Gnu Public License.

GUI: Graphical User Interface.

ICE: Internet Communication Engine, an object oriented middleware developed by ZeroC, Inc.

IDL: Interface Definition Language. A standardized language that is used to define the interfaces used by multiple software components interacting with each other.

Inversion of control: A software abstraction where the flow control logic of an application is removed from the user and placed within the application framework.

J2SE: Java version 2, Standard Edition

JDBC: An API for the Java programming language that defines how a client may access a database.

JNI: Java Native Interface, a programming framework that allows Java code running in a Java Virtual Machine (JVM) to call and to be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages, such as C, C++ and assembly. (From Wikipedia)

JVM: Java Virtual Machine, the runtime engine required to execute Java Byte Code.

KCSF: The Keck Common Services Framework

Location transparency: A concept where network resources are accessed by name alone, without regard to their location or the user's location.

LGPL: Lesser General Public License.

Loose coupling: Loose coupling occurs when classes have limited knowledge of other classes, typically through an interface, which can be implemented by any of several concrete classes. This increases flexibility.

Middleware: Communication software that allows components to communicate with each other without regard to their physical location on the network.

Narrow interface: A method interface that is only dependent on a few well-defined parameters.

NDDS: The former name of RTI's DDS product, now simply called RTI DDS.

NOAO: The National Optical Astronomy Observatory.

OLE: Object Linking and Embedding, a technology that allows embedding and linking to documents and other objects developed by Microsoft.

OPC: Open Connectivity, set of standard interfaces and protocols intended to foster greater interoperability between automation/control applications, field systems/devices, and business/office applications in the process control industry.

OS: Operating system.

Peer-to-peer communication: Point-to-point communication between two entities where each is cognizant of the other.

Publish-subscribe communication: A general form of communication between multiple entities where there are multiple publishers and subscribers who, without regard to the presence or location of each other, send and receive data in the form of messages called topics.

RDBMS: Relational Database Management System.

RTC: Real-Time Controller. A CORBA-based framework developed by NASA JPL and used in the Keck Interferometer.

TANGO: TAco Next Generation Objects, an object-oriented open source control system based on CORBA.

Tight coupling: Tight coupling occurs when classes have direct knowledge or dependence on other classes, which limits flexibility.

TINE: Three-fold Integrated Networking Environment, a control system architecture.

UI: User Interface.