NGAO Software Architecture
KAON 676: Configuration Service

Author	Modified	Notes
Douglas Morrison	7/27/09	NGAO Configuration Service

1.0 INTRODUCTION 3

2.0 CONFIGURATION MODEL 4

2.1 DATABASE SCHEMA 5
2.2 VERSIONING 6
2.2.1 TAG 7
2.2.2 CHECKOUT 8

3.0 CONFIGURATION LIBRARY 9

3.1 DESIGN 9
3.2 CLIENT INTERFACE 9
3.3 ADMINISTRATOR INTERFACE 9

4.0 CONFIGURATION SERVICE API 11

4.1 GETCONTAINERMANAGERCONFIGURATION 11
4.2 GETCONTAINERCONFIGURATION 11
4.3 GETCOMPONENTCONFIGURATION 12
4.4 GETMETADATA 12

5.0 CCM CONFIGURATION MODEL 13

5.1 CONTAINER MANAGERS 13
5.2 CONTAINERS 13
5.3 COMPONENTS 13

6.0 ADMINISTRATION TOOLS 15

6.1 SCRIPTS 15
6.1.1 DATABASE BUILDER 15
6.2 CONFIGURATION GUI 16
6.2.1 CLASS DEFINITIONS 16
6.2.2 INSTANCES 17
6.2.3 NAVIGATION 17

APPENDIX 20

1.0 Introduction

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.

The Configuration Database will be managed and accessed through a set of tools created for the KCSF. A distinct separation will be made between administrative and client database access. Administrative tools, such as the Configuration GUI, will be permitted to modify the state, contents, and properties of the database. Clients on the other hand will be restricted to read-only access of the configuration data. The KCSF Configuration Service will provide the primary means of access to clients, and will be responsible for opening and maintaining a connection to the database, as well as retrieving and formatting data for the clients.

To improve reusability and efficiency when communicating with the Configuration Database, a Java library will be created to implement the administrator and client interfaces. This library will encapsulate and hide the majority of the database details, as well as format and convert data to and from application level objects to database queries. Clients and applications will utilize one of the administrator or client interfaces defined in the library to access the configuration data.

The following sections detail the configuration model.

2.0 Configuration Model

The configuration system for the KCSF infrastructure is designed as a multilayered distributed model.

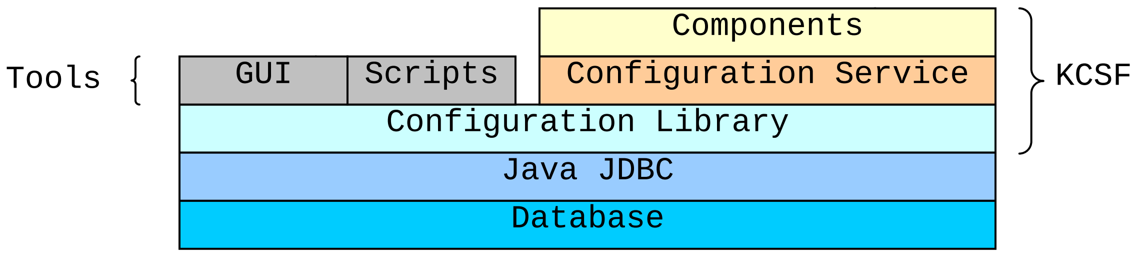


Figure 1: Configuration Layers

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. (See section 3.0 for more information on the Configuration Library).

Above the Configuration Library sits 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 Services 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.

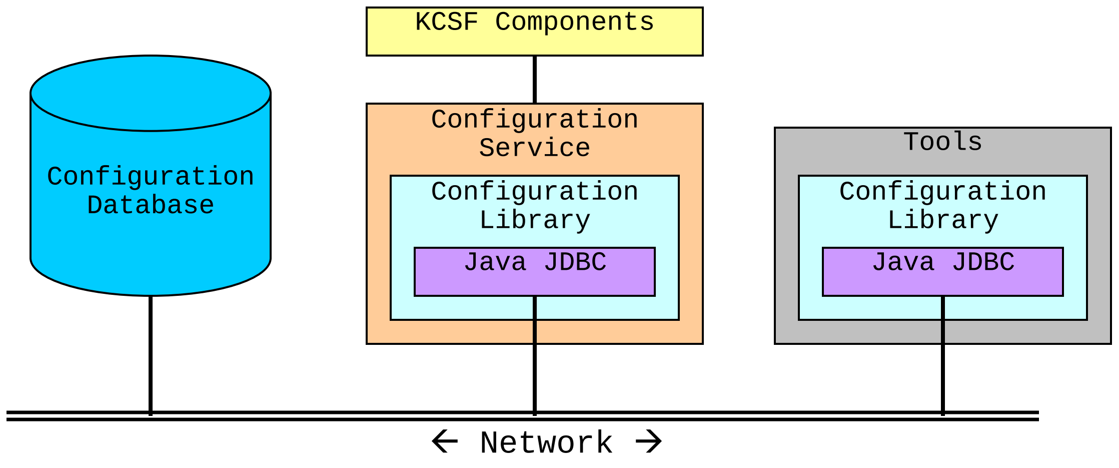


Figure 2: Distributed Database Access

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.

2.1 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.

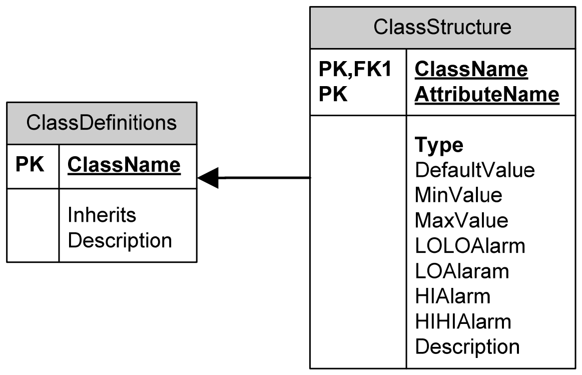


Figure 3: Class Schema

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.

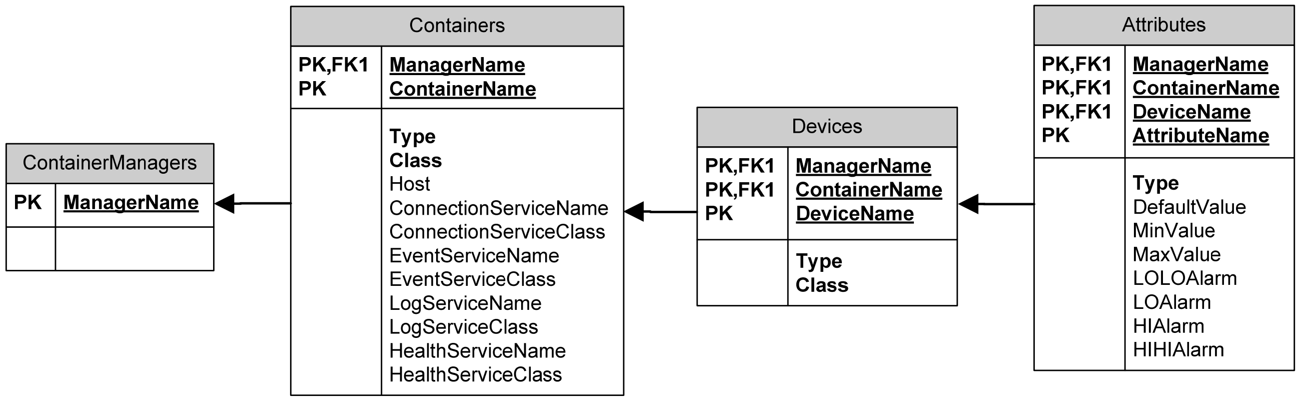


Figure 4: Instance Schema

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.

2.2 Versioning

The database model will be designed to provide versioning capabilities for configuration information.

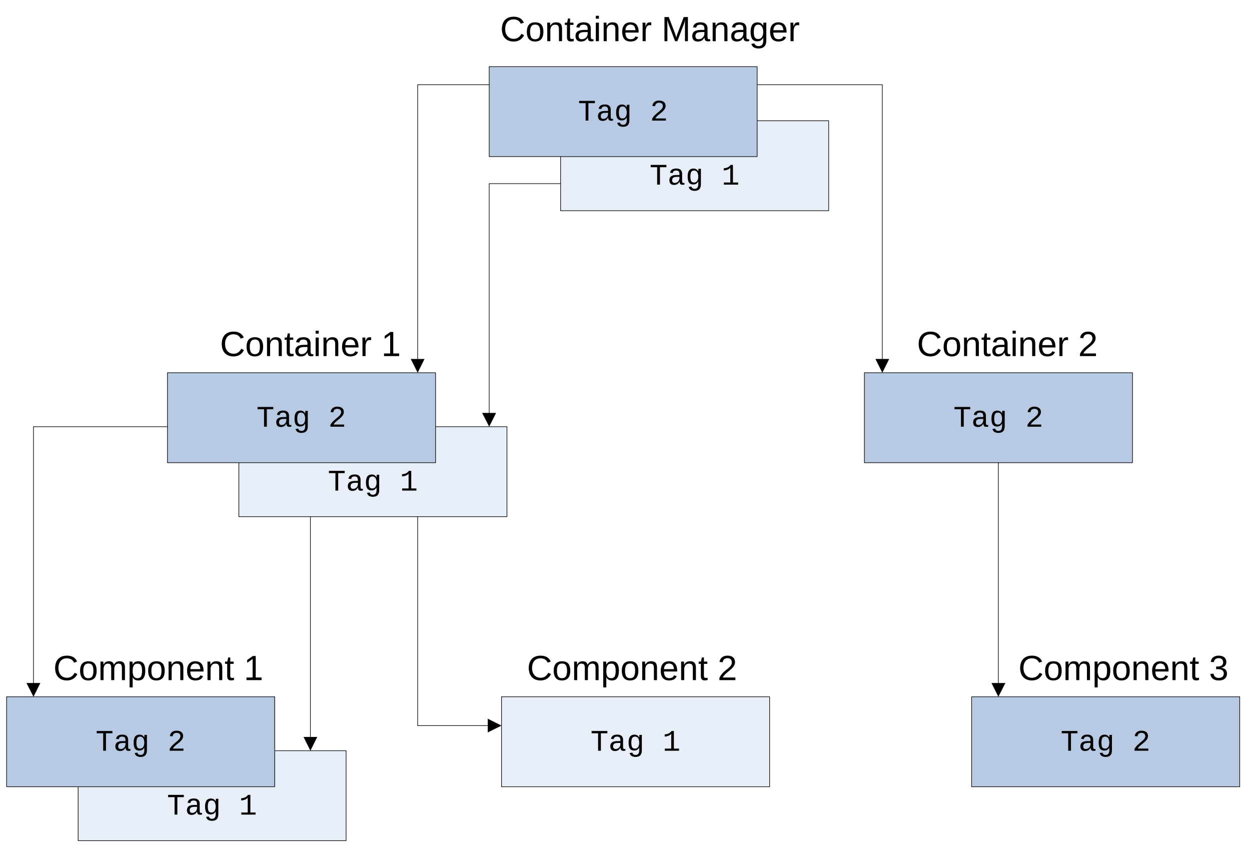


Figure 5: Data Versioning

Figure 3 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.

2.2.1 Tag

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.

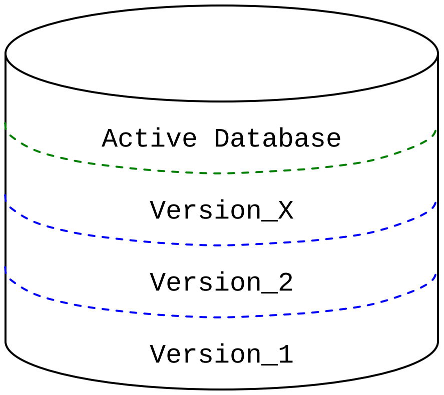


Figure 6: Database Versions

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.

2.2.2 Checkout

The process of restoring a specific version of the database is called *checkout*. When a checkout is performed the current state of the active database is completely overwritten by the values and relationships defined in the tagged version. Modifications to the active database will not impact the saved version. To update an existing version you must perform a tag with the associated version ID.

3.0 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 upon that by providing functionality to modify the contents and structure of the database.

3.1 Design

The Configuration Library is a KCSF Java module that wraps JDBC, acting as a translation layer between SQL queries and framework compatible types (Attribute Lists). Database schemas and relationships are defined within the library allowing the module to expose simple methods for otherwise complex SQL statements. Changes to the database schema or table relationships will need to be updated in the library to maintain compatibility.

3.2 Client Interface

The client interface exposes read only capabilities to users. Within the Keck Common Services 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);
};
```

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.

3.3 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 poperties).
- *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.

4.0 Configuration Service API

The Configuration Service is the primary means of access of configuration information for software components. The service acts as the bridge between the KCSF and the Configuration Database, and provides software with read-only access to configuration information. The following pseudo-code details the Configuration Service interface.

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

Configuration Service implementations provide the capabilities to acquire configuration data for each of the base KCSF types. 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 specific format of the attribute lists varies for each method and is detailed in the following sections.

4.1 getContainerManagerConfiguration

This method returns an attribute list containing all of the Manager's configuration information. Specifically, this is a set of symmetric lists defining all of the containers the manager will be responsible for deploying.

IAttributeList:

- *ContainerName* : String []
- *ContainerClass* : String []
- *ContainerType* : String []
- *ContainerHost* : String []

A container is defined by a name, Java class, type, and host machine. With this information the Manager will be able to find, load, create, and deploy any number of Container instances.

4.2 getContainerConfiguration

This method returns an attribute list containing all of the Container's configuration information. Specifically, the configuration defines the name and Java class for all of the KCSF services, and a set of symmetric lists defining all of the components that will be maintained by the Container.

IAttributeList:

- *ConnectionServiceName* : String
- *ConnectionServiceClass* : String
- *EventServiceName* : String
- *EventServiceClass* : String
- *LogServiceName* : String
- *LogServiceClass* : String

- *HealthServiceName* : String
- *HealthServiceClass* : String
- *ComponentName* : String []
- *ComponentClass* : String []
- *ComponentType* : String []

Each service will be defined in the attribute list by type. The Container only needs to know the name and specific class to load and create a service. Similarly, components are defined by a name, Java class, and a type. The container will use this information to load, create, and initialize each of its child components.

4.3 getComponentConfiguration

The *getComponentConfiguration* method returns an attribute list containing the configuration information for a component.

IAttributeList:
- *AttributeName* : *<Type>*

Component configuration is simply a name-value pair for each of the object's configurable members. The value returned for each attribute is the default as defined in the Configuration Database. The type of the attribute can be any of the KCSF enumerated types. Meta-data is also associated with each Component attribute (the default value being one of those meta-data items). After the component has obtained it's list of attributes it will query the Configuration Service for each attributes meta-data through the *getMetaData* method.

4.4 getMetaData

This method returns an attribute list containing all of an attribute's metadata. *AttributeName* must be a fully qualified unique name.

IAttributeList:
- *DefaultValue* : *<Type>*
- *MinValue* : *<Type>*
- *MaxValue* : *<Type>*
- *LOLOAlarm* : *<Type>*
- *LOAlarm* : *<Type>*
- *HIAlarm* : *<Type>*
- *HIHIAlarm* : *<Type>*
- *Type* : String

Every attribute will have associated with it a set of meta-data properties. These include the default value, type, and if applicable, min /max and alarm values. Components and their subclasses can use this information to determine acceptable value ranges and trigger automatic alarms if any of the thresholds are crossed.

5.0 CCM Configuration Model

The configuration service is used by the majority of the KCSF application objects. At the highest level is the Container Manager. This object is responsible for deploying and initializing containers on specific hosts. However, since Container Managers are themselves a type of Component, and can not be created on their own, a Container is used to instantiate the Manager. From the Container's perspective the Manager will be treated like any other component, and as such the Container will behave in the standard way: configuration will be read from the database, the Manager will be created, and services injected.

5.1 Container Managers

After the Manager has been created and is put into its initialization phase it will be ready to create and deploy child Containers. The Manager will execute the Configuration Service's *getContainerManagerConfiguration*, passing in its fully qualified name. The service will return an Attribute List defining all of the containers and their properties in symmetric sequences (index *N* in each sequence pertains to container *N*). The Manager will use this information to create each container, and start the process or task on the appropriate host. The Manager will obtain communication proxies with each of the Containers before entering an idle state. The Manager will remain in this state until it is commanded by a user, or is shutdown.

5.2 Containers

Containers are responsible for initializing services and creating and deploying components. A Container will obtain its configuration information from the Configuration Service through the *getContainerConfiguration* method. The Configuration Service will return an Attribute List defining all of the additional services and components the Container will need to create.

Services are defined by a name and Java class type, and the Container will use this information to load the associated class code and create service instances. After the services are instantiated they will be initialized and ready for use by components.

Components are defined by a name, class, and type identifier. The Container will once again use a Java loader to create instances of each component, followed by injection of services, and object initialization. Once the objects have completed their startup phase the Container will deploy them for use.

5.3 Components

Components are responsible for implementing the functional requirements of a system, and are created, deployed, and managed by Containers. When a Component is created it will be provided with all of its services and dependencies. Components obtain their configuration during the startup phase. This is done by executing the *getComponentConfiguration* method on the Configuration Service. The service will return an attribute list containing all of the named attributes and properties of the Component and their default value. Java reflection can be used to automatically configure a component, or the attribute data can be passed up to user defined method for processing.

Once the Component's members have been initialized, the meta-data for the attributes will need to be collected. Meta-data exists for each configurable attribute, and defines properties for the attribute including its type and default value. The Component will iterate through each attribute obtained during the previous Configuration Service call, and pass the fully-qualified attribute name to the *getMetaData* method. This will return an attribute list containing all of the attribute's meta-data. A map will be maintained by the component to associate the read-only meta-data with each configuration attribute.

6.0 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.

6.1 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.

6.1.1 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 would 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 7: Class Definition Example

6.2 Configuration GUI

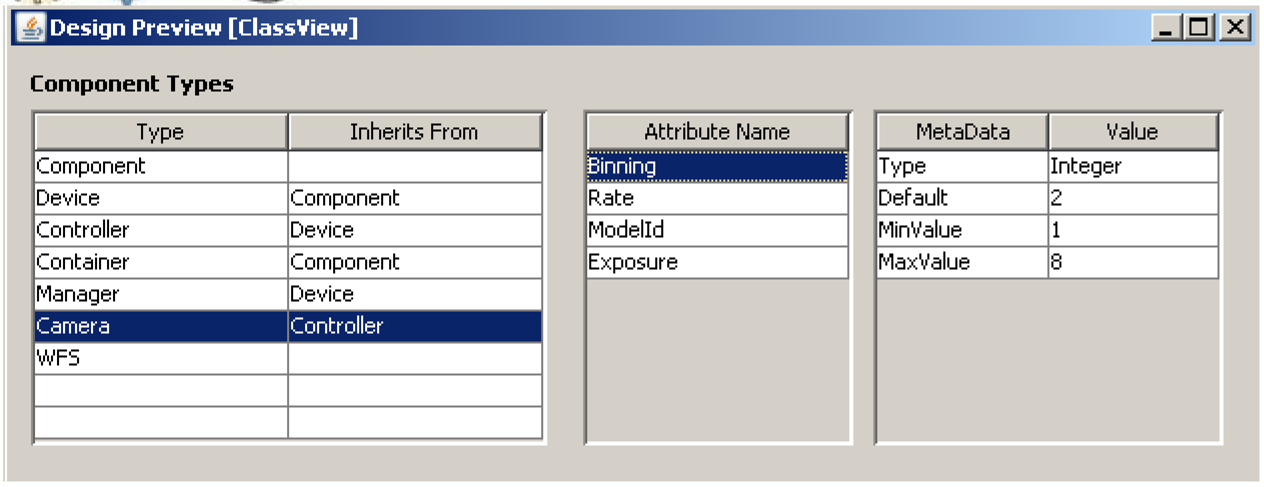
The Configuration GUI represents the primary means for full database control and manipulation. The GUI will provide all of the functionality available in the scripts, as well as utilize the full extent of the Configuration Library administrator interface:

- 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 creating a configuration for an application, the user would 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.

6.2.1 Class Definitions

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



Component 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 8: Example Class View

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.

6.2.2 Instances

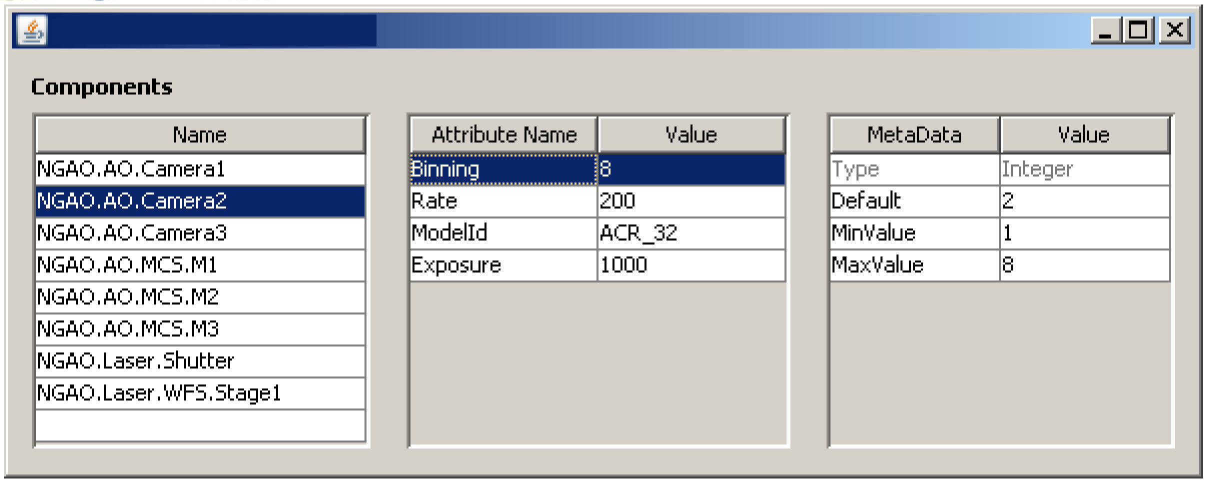
The Instances panel shows all of the object configurations for a project. From this panel a user can add and remove instances from a project, and modify their configuration values. When an instance is added it will automatically be populated with the default values specified in the class definition.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

6.2.3 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.)

6.2.3.1 Component View

The component view 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

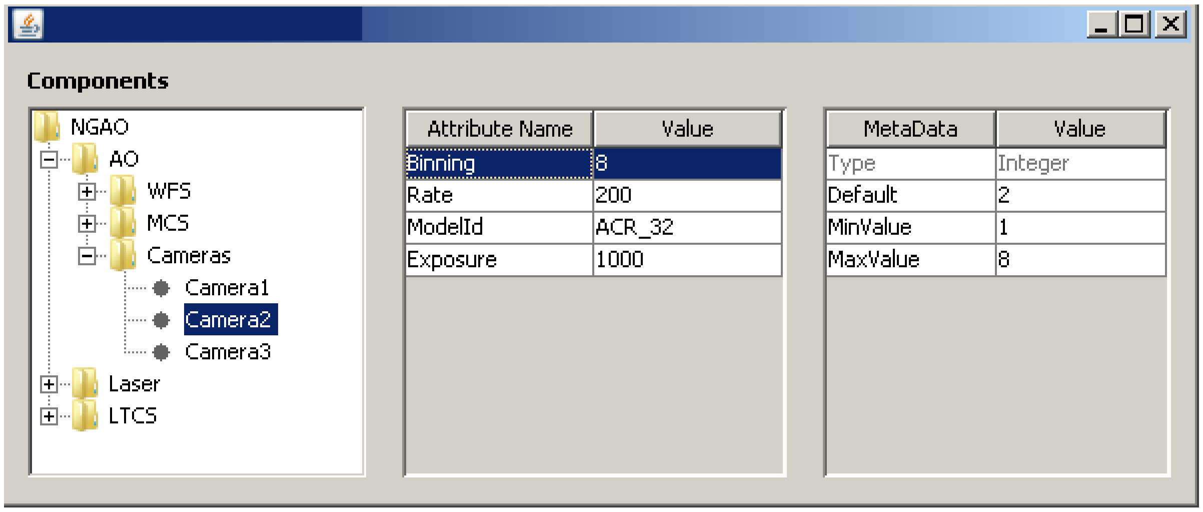
MetaData	Value
Type	Integer
Default	2
MinValue	1
MaxValue	8

Figure 9: Example Component View

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.

6.2.3.2 System View

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



Components

```
NGAO
  AO
    WFS
    MCS
    Cameras
      Camera1
      Camera2
      Camera3
  Laser
  LTCS
```

Attribute Name	Value
Binning	8
Rate	200
ModelId	ACR_32
Exposure	1000

MetaData	Value
Type	Integer
Default	2
MinValue	1
MaxValue	8

Figure 10: 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.

Appendix

Java JDBC:
http://java.sun.com/docs/books/tutorial/jdbc/index.html