Keck Adaptive Optics Note 671

Keck Next Generation Adaptive Optics Container Component Model

Jimmy Johnson, Douglas Morrison, Erik Johansson
Revision 2: June 3, 2009
Revision 1: April 30, 2009

1.0 Introduction

Software development and design have evolved significantly over the last few decades. They have evolved from Structured Development to Object-Oriented Development to Distributed Development where applications no longer exist in a single process space on a single node, but can be distributed over many machines throughout a network. Many solutions to distributed application development grew from this trend, including CORBA, .Net, and Java RMI, each providing their own services and features to manage communication for distributed computing.

Distributed Development has been a very successful design method for large software projects and growing businesses, however, it still suffers 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 compilation and deployment. This results in reduced service to clients while the system is in flux. 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 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. ESO has realized a component based development model using CORBA for their Atacama Large Millimeter/submillimeter Array (ALMA) Common Software, and component based development solutions are in development for the Advanced Technology Solar Telescope (ATST) and the European Extremely Large Telescope (E-ELT) among others.

The following sections will detail the CBD model, and discuss an implementation that will be used in the NGAO Software Architecture design. In this document component based development is described as a platform independent model (PIM). The PIM is independent of the component architecture, assuming only a generic notion of components, with ports and attributes. Later documents will describe the NGAO CBD in a platform specific way.

2.0 Component-Based Development

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. CBD is intended to be applied to all aspects and phases of a project lifecycle, and is based around the concept of component-oriented design.

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

A component is a piece of software that 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 CBD and non-CBD system.

In a system built with a traditional development model it is possible that a modification to a module or class will precipitate through to the entire system. Although conceptually the modification is isolated to a single unit, other modules that rely on its functionality and interface may also be affected. This can result in the recompilation and deployment of many modules, or the system as a whole.

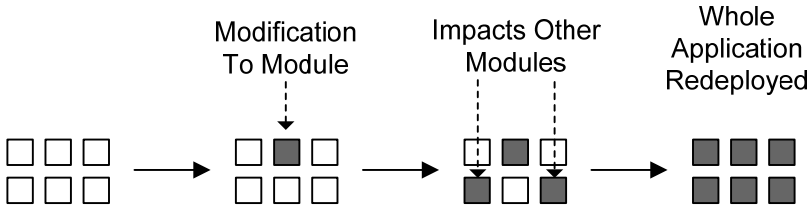

Figure 1: Traditional System Maintenance

In a CBD system a modification to a component implementation is truly self-contained, and impacts only the module affected. In theory only the module that has been changed will need to be redeployed.

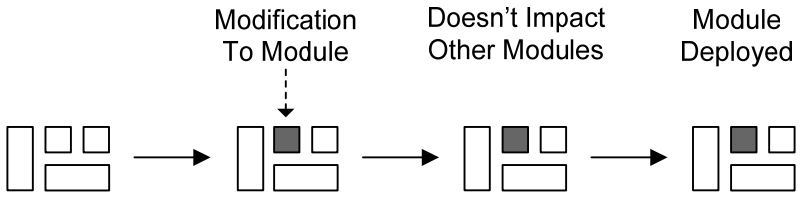

Figure 2: CBD System Maintenance

Although the above example specifically addresses the maintenance phase of a project, the same concept of component based design can be applied to the earlier stages as well. With its architecture and module centric approach to development, system design is partitioned into discrete components where integration between components is well defined, and solution implementation can be considered independently. Since CBD separates the system design from the component design, developers experienced in one area do not need to be trained to understand the entire architecture. Component developers can focus on the implementation details of specific solutions, while the system designers can be focused on the technical requirements, concurrently.

3.0 CBD Example

This section details a simple example using the CBD approach to create an application capable of reading configuration data from an external source. We start by defining an interface that our application will use to obtain configuration data.

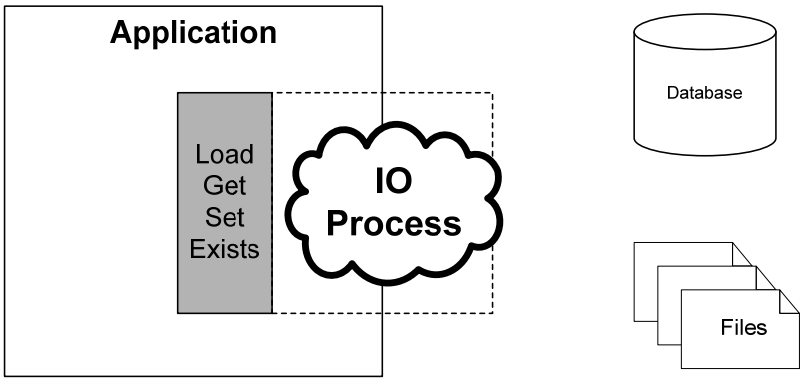

Figure 3: Simple I/O Component

Without concern for how configuration data is stored or even where it is located, the developers can plan out what capabilities an I/O configuration component will need.

- The application will need to be able to load a specific configuration based on a unique identifier.
- An application will need to be able to resolve configuration values by using some form of a get method.
- An application may need to be able to update a configuration value, requiring a set implementation.
- Finally, there should be a way to test if a configuration item actually exists.

These requirements can be translated into the following class definition:

```
class IOComponent {
      public IOComponent();
      public boolean Load(String Name);
      public String Get(String Item);
      public boolean Set(String Item, String Value);
      public boolean Exists(String Item);
}
```

The developer has successfully defined an external interface for I/O components. Specific components can now be implemented against this interface to perform I/O from any available source.

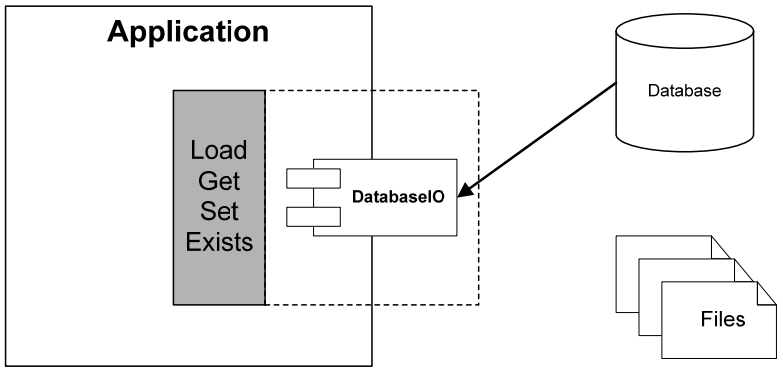

Figure 4: I/O Database Component

An I/O Component can be created to access configuration information from a database, parameter files, over the network, interactive command line, etc. If a business upgrades their configuration

storage, a new component can be created to handle the IO process and it can be dropped into the system to replace the old implementation. As long as both the component and application honor the interface, old implementations can be swapped out with new implementation, and from the perspective of the system, nothing has changed.

4.0 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. It provides for separation of concerns between the functional (business logic) and the technical architecture. The pattern leads to easier development, deployment and reuse. Object Management Group (OMG) standards exist for a 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.

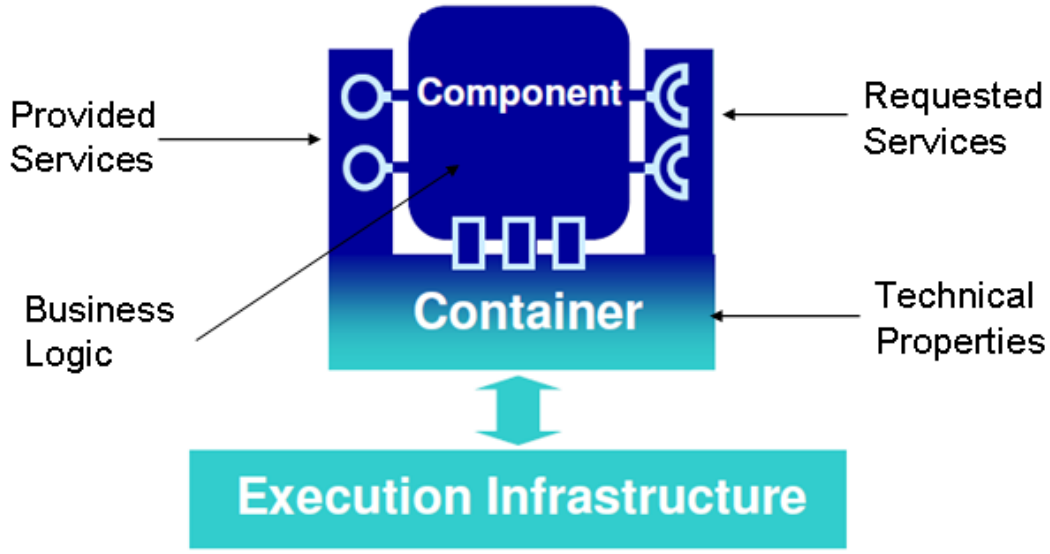

Figure 5: Container Component Model

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. A container will create the components, start it running, shut it down, and remove it 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.

5.0 CCM Ecosystem

CCM includes a number of key entities:

- Container Manager
  - A component responsible for managing containers. This is capable of deploying containers on different nodes, attaching to already running containers, stopping and removing containers.

- Container Server
  - A process in which one or more containers actually execute.
- Container
  - Manages components and provides services and connections.
- Component
  - A standalone reusable functional entity.
- Services
  - Services are typically shared by all components in a container. Examples include logging, alarming etc.

A container server is a process that provides an arbitrary number of containers during runtime. A container is created as a result of deployment of a component. A container manager determines the appropriate set of policies and service bindings to be used by the container, and then acts as a factory to create containers.

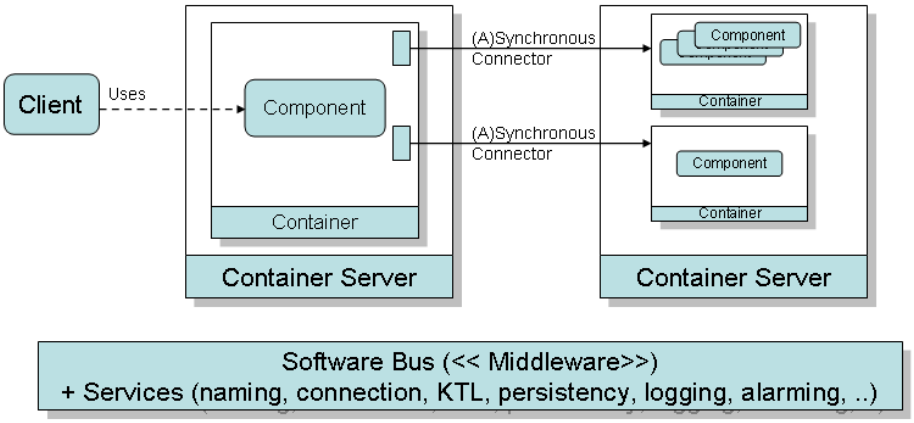

Figure 6: General Model for container and servers

Figure 6 shows a general model for container and server configuration. A container server instantiates one or more containers to host several software components. A client interacts with components by sending requests to the component via the software bus (Note: if components are on the same machine the communication is via IPC not RPC). The container intercepts each request from the client to ensure that transaction, security, and persistence are properly applied before the actual operation is performed on the component. If a component is connected to another component, the container provides synchronous and asynchronous connectors to access remotely hosted components.

As containers provide the creation and life-cycle management of the system, components are designed to implement the functional requirements of the system. 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, and shutdown), a component will provide 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.

By utilizing an architectural pattern called Inversion of Control (IOC) we can gain further advantages from the CCM pattern. IOC is a key part of what makes a framework different than a library. A library is essentially a set of functions that you can call, typically organized into classes. Each call does some work and returns control to the client. A framework embodies some abstract design, with more behavior built in. To use the pattern, behavior is inserted into various places in the framework. In this case the framework can define an interface that a client code must implement for the relevant calls.

The framework's code then calls your code at these points. We plan to realize a specific style of IOC using dependency injection.

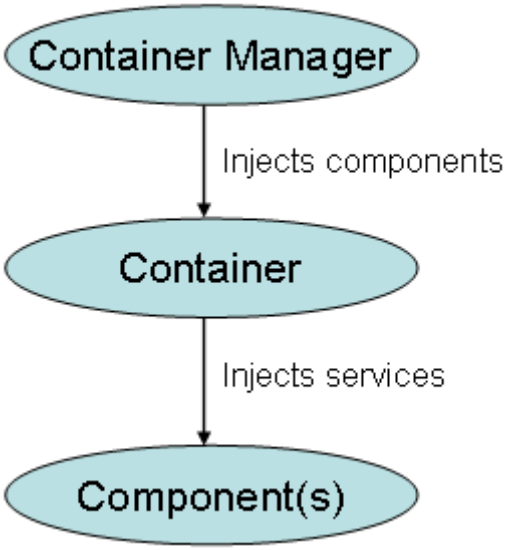

Figure 7: Inversion of Control pattern for CCM

Dependency injection allows the system to build a container-component relationship during system initialization, instead of explicitly defining relationships programmatically in source-code or at compile time. To effectively execute dependency injection a third component is added to the model, a Container Manager. As the name suggests, the Container Manager is responsible for the creation, initialization, and management of individual containers. Based on customizable configuration properties the manager will create instances of containers, and "inject" component information into the containers. The containers will use this information at run-time to load and instantiate the component instances which they will be responsible for managing. The container will in turn inject the common services it supports into the components. This design allows services to be shared or made private as necessary for each component.

This design pattern provides great flexibility for developing, maintaining, and upgrading software systems because it allows developers to design container-component relationships outside of the system, without requiring any modifications to the existing code base.

5.1 Components

The basic entity in component-based software is a component. "A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. . . Larger pieces of a system's functionality may be assembled by reusing components as parts in an encompassing component . . ., wiring together their required and provided interfaces." *Object Management Group, "Unified Modeling Language, Version 2.0, Superstructure."*[1]

[1] *Document number ptc/04-05-02, May 2004. http://www.omg.org/cgi-bin/doc?ptc/04-05-02*

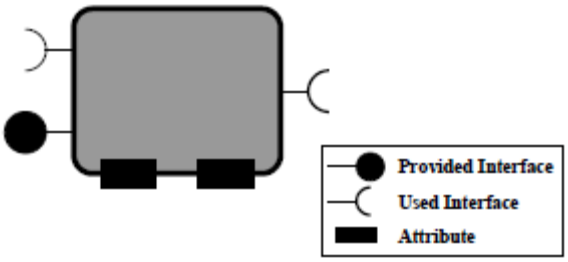

Figure 8: Component

Figure 8 shows an illustration of a component that has one provided interface port, two used interface ports, and two attributes. A provided interface port represents the set of methods and features made available to the system by the component for commanding and interaction. Used interface ports represent the interfaces of other components in the system that the modeled component relies on. The component is shown as a black box, entirely defined by its set of ports and attributes, to imply that the component's implementation is replaceable, as long as each implementation conforms to the same interface. Once a component interface is specified, one or more component implementations of that interface can be created. Based on the same set of syntactic and semantic requirements, they are functionally identical.

Components implement the functional requirements of the system, and provide the health, status, and state information for the devices they control. When a component is created by its parent container it will be in an uninitialized state. The container will provide the component with all of the communication interfaces and services that it requires, along with any other general startup properties. The container will then tell the component to commence its initialization and startup. Here a component will perform its custom startup tasks that can include,

- Accessing configuration information
- Creating sub-components
- Connecting to drivers
- Initializing hardware and software

When the component initialization completes, the container will tell it to enter its run state where it will be readied for control mode. At this point the component and its devices are ready for operations.

5.2 Containers

Containers are responsible for managing the deployment, initialization, and life-cycle of one or more components. They also provide a consolidated health status based on the each of the components, and their own overall status. Containers (and the components they manage) are designed to be run as separate processes or within dedicated threads. On Solaris or Linux hosts each container will be launched as an executable: for every container implementation an accompanying *main* routine will be defined and built into the object code. On a VxWorks host, or other task based operating system, an instance of the container class will be created and assigned to a thread. The method of deployment will be defined in configuration and will be used by the Container Manager to determine how and where to create and start a container.

After a container process has been created, the container instance will open connections to the NGAO communication framework and wait for commands from its Container Manager. At this point the container has not yet created any components or initialized any services. The manager will gather the information it needs from the configuration database for each of its containers, and will invoke the appropriate routines on the container to load and create components and connect to the system services. The container will be responsible for using this information to load and incarnate the

components and services, and setting the services for each component. The container will then wait for a command to initialize its members, before putting the components into their run state for operational control.

5.3 Container Manager

The Container Manager is responsible for deploying and initializing containers and their components through dynamic process creation and dependency injection. Container managers will have access to configuration information that defines the number and types of containers and their specific components and services.

The manager will iterate through each object that it needs to create, and collect the configuration information that will be used to determine the type and properties of each container. The container class will be developed to run within a dedicated thread or as its own process, and it will be the manager's responsibility to start the container on the specified host machine. The manager will tell the containers which services and components to load by passing them the formatted configuration information. The manager will then issue an *initialization* command to each container that will cause it to create the services and load the components. After the containers have been initialized the manager can tell them to put their components into their run state for operational system control through the *startup* routine.

When the Container Manager is brought up (either as an executable or as part of a hierarchical system startup) it will perform the container creation and deployment automatically. The manager interface also provides methods to explicitly create, deploy, and destroy containers and their components during run-time. This will give operators the power to swap out components dynamically, or restart components if there is a problem with the system. Since the Container Manager will be developed with the NGAO framework we will be able to create a system-wide user interface that can allow operators to easily bring up, modify, and shutdown portions of the system with a few mouse clicks.

6.0 Distributed Architecture

For NGAO we aim to develop a highly distributed system for improved load balancing and domain partitioning. There is also a strong desire for rapid fault recovery if any part of the system should go down, without requiring a full system reboot. For these reasons the NGAO CCM design will focus on the creation and management of many independent containers instead of a single large-scale hierarchy.

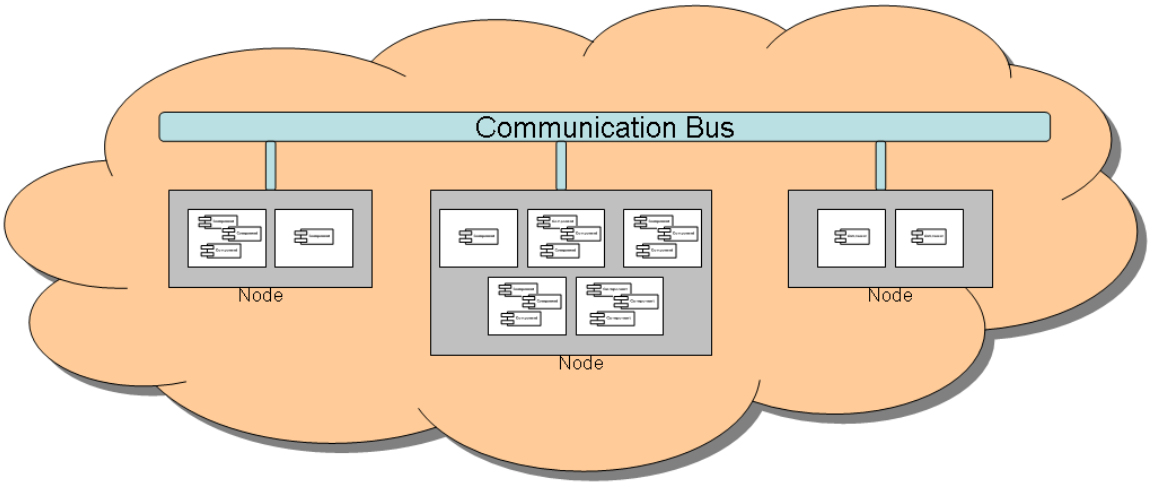

Figure 9: A distributed CCM view

In order to deploy containers and their components independently of others, we will design them so they can each be assigned to a dedicated process or thread. This will enable operators to start and stop specific NGAO tasks in order to balance resource usage over hosts or recover from a failure. Multiple containers can exist on a host at once and will be able to communicate with each other using the common middleware framework.

7.0 NGAO Specifics

Figure 10 shows a conceptual model of how NGAO will realize a CCM.

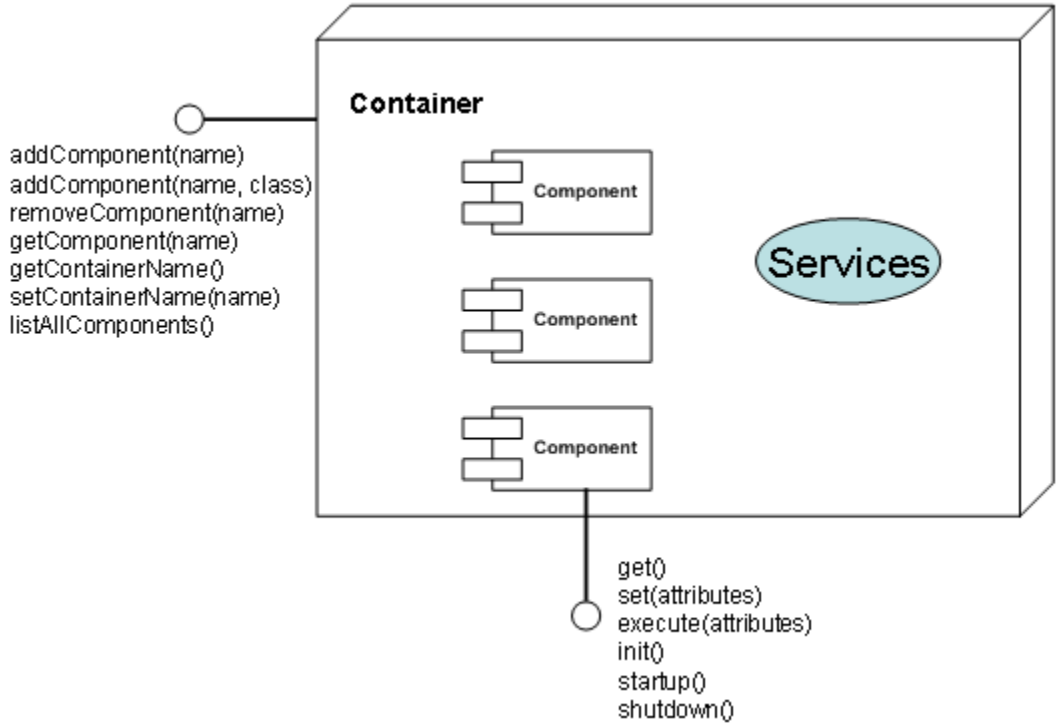

Figure 10: Proposed CCM implementation for NGAO

Key interfaces for each of the main entities are shown. The specifics of the interfaces, methods, attributes and services will be discussed in a separate document.

Note: by treating a container manager as a component we can use the standard software infrastructure to bootstrap the entire system or subsystems. As Figure 11***Figure*** shows, the system can be started by creating a small main program that creates a container and assigns a container manager component to it. When the container manager is started it will deploy its containers as necessary and set them to a running state. Using the framework the status of the container manager can be monitored and it can be controlled like any other entity.

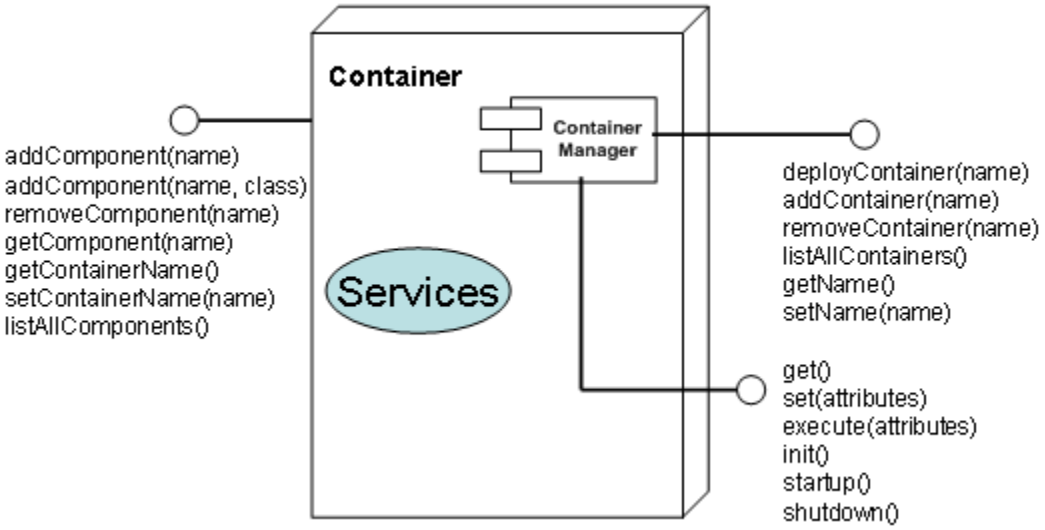

Figure 11: Bootstrapping the system

Figure 12***Figure*** provides a high level example of some domain specific controllers that will be developed for NGAO. The diagram is not meant to provide a complete overview of the NGAO control system (the controls architecture will cover this in more detail) and shows only a few main hardware devices. It is being used to illustrate the concept that there will be one or more base controllers provided as part of the software infrastructure. These will be then used to create more domain controllers such as linear stages, camera controllers etc. These in turn can be aggregated to create more complex multi-stage or coordinating controllers as shown in Figure 12. All of these will conform to the container component model and can be created, deployed and controlled in the same manner.

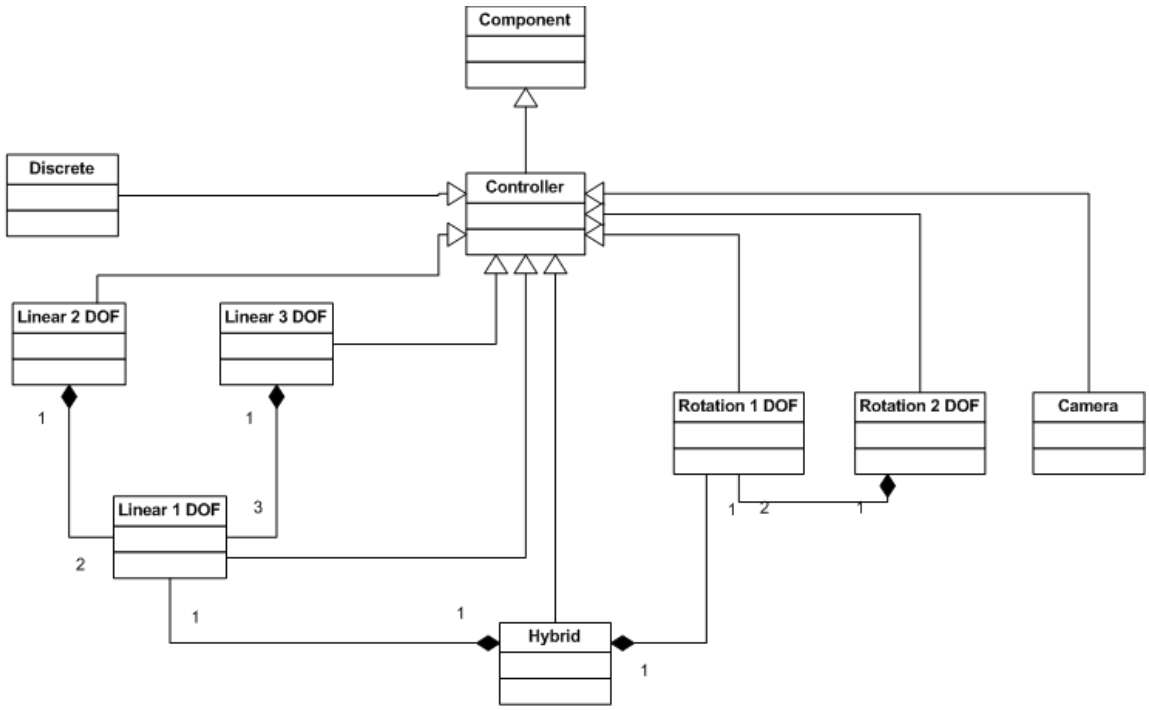

Figure 12: NGAO Controller subset

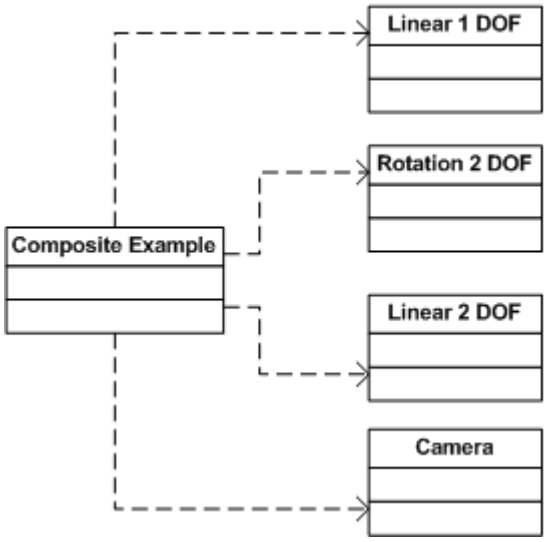

Figure 13: Composite Controller Example

8.0 Glossary of Terms

- *IPC (Inter-process Communication)*: The ability to exchange data and messages between multiple threads within one of more processes.
- *Persistence*: The ability for data to outlive the run-time execution of an application.
- *RPC (Remote Procedure Call)*: The ability to invoke commands and routines in another address space either on the same machine or across a network.
- *CORBA (Common Object Request Broker Architecture)*: A vendor and architecture independent object management and RPC invocation framework used for distributed applications and systems.
- *LWCCM (Lightweight CORBA Component Model)*: A component development model based around the lightweight / reduced functionality implementation of CORBA for embedded environments with strict resource allocation.
- *ATST (Advanced Technology Solar Telescope)*: A ground based solar telescope being developed through the collaboration of over 20 institutions, currently planned for deployment on the island of Hawaii.
- *ALMA (Atacama Large millimeter Array)*: A radio astronomy observatory currently under construction in the Chilean Andes.
- *E-ELT (European Extremely Large Telescope)*: A proposed design for a 42 meter telescope for the next generation European Southern Observatory optical telescope.
- *CBD (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.
- *CCM (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.
- *Inversion of control*: An abstract principle applied to software development that moves the flow control logic of an application to a separate entity.
- *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.
- *Java RMI:* Java Remote Method Invocation is an API provided by the Java SDK that performs the object-oriented equivalent of RPC. Distributed objects build upon the Remote base class, and leverage Java's language features to provide dynamic downloadable class definitions in remote virtual machines.
- *Component*: A distributed object that implements the control logic for a device or set of devices.
- *Container*: A class responsible for the dynamic loading and creation of components and for managing their life cycle for the duration of the application.
- *Container Manager*: A service that is responsible for creating, configuring, and managing containers and their processes.
- *Middleware*: An abstraction layer that hides the details, management, and specifics of a more complex underlying system.