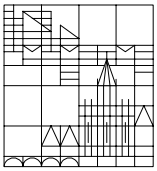


University of Constance
Dept. of Computer Science

CONTROLLER MANAGEMENT FRAMELET

Concept And Architecture Description

Abstract

This document was written as part of the study "Design and Prototyping of a Software Framework for the AOCS" done under contract Estec/13776/99/NL/MV for ESA-Estec. The purpose of the study is the development of a software framework for the Attitude and Orbit Control Subsystem (AOCS) of a satellite. The framework will be built as a collection of framelets. This document describes the controller framelet. This framelet proposes an architecture to handle closed-loop controllers. The framelet enhances reusability because it decouples the task of managing the controllers from the implementation of the control algorithms.

Written By: A. Pasetti / T. Brown

Date: 30 April 2002

Issue: 2.1

Reference: SWE / 99 / AOCS / 016

TABLE OF CONTENTS

1 REFERENCES

RD1 E. Gamma *et al.* (1995), *Design Patterns – Elements of Reusable Object Oriented Software*, Reading, Massachusetts: Addison-Wesley

RD2 A. Pasetti (1999), *AOCS Framework – Concept Level Description*, AOCS Framework Document ref. SWE/99/AOCS/004

RD3 Deleted

RD4 A. Pasetti (2001), *Software Frameworks and Embedded Control Systems*, LNCS Series, Springer-Verlag, To appear in Dec. 2001

RD5 A. Pasetti (2000), *Operational Mode Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/009

2 ACRONYMS

AAD	Attitude Anomaly Detection
AOCS	Attitude and Orbit Control Subsystem
AST	Autonomous Star Tracker
CSS	Coarse Sun Sensor
ES	Earth Sensor
FDIR	Failure Detection, Isolation and Recovery
FPM	Fine Pointing Mode
FSS	Fine Sun Sensor
GYR	Gyroscope
KF	Kalman Filter
IAM	Initial Acquisition Mode
MIMO	Multi-Input-Multi-Output
NM	Normal Mode
NTT	Non-Time-Tagged
OBDH	On-Board Data Handling system (aka as OBDS)
OCM	Orbit Control Mode
OO	Object-Oriented
PD	Proportional-Derivative controller
PI	Proportional-Integral controller
PID	Proportional-Integral-Derivative controller
RRM	Rate Reaction Mode
RTOS	Real-Time Operating System
RW	Reaction Wheel
SAS	Sun Attitude Sensor
SBM	Stand-By Mode
SISO	Single-Input-Single-Output
SPS	Sun Presence Sensor
STR	Star Tracker
SLM	Slewing Mode
SM	Safe Mode
TC	Telecommand
THU	Thruster
TM	Telemetry
TT	Time-Tagged

3 INTRODUCTION

This document describes the *controller management framelet* for the AOCS framework. The framelet is described at both the framelet concept level and at the framelet architectural level.

This framelet proposes an architecture to implement closed-loop controllers. The framelet enhances reusability because it decouples the task of *managing the controllers* from the *implementation of the control algorithms.*

3.1 Context

The context for the design of the framelet is described in RD2. The present document assumes that the reader is familiar with RD2 and in particular with the section dealing with the controller management.

The architecture proposed here follows the concept outlined in RD2.

In comparing the present document with RD2, the reader should bear in mind that the class definitions presented in the latter document are not necessarily entirely consistent with the class definitions presented here. This is because the main purpose of RD2 was to introduce an architectural *concept* whereas the main purpose of the present document is to describe an architecture. The design presented here therefore should be regarded as an evolution of the design presented in RD2.

Also note that MIMO controllers are not implemented in the framework prototype.

3.2 Applicability to Java Version

The AOCS Framework was first implemented in C++ and then ported to Java. This document was originally written for the C++ version and is only partially applicable to the Java version. Generally speaking, the description of the framelet at design level – in particular its design patterns – is language-independent and is equally applicable to both the C++ and Java versions whereas the architectural-level description is more tied to the C++ version. For a detailed description of the architecture of the Java framework, readers should refer to the JavaDoc documentation generated from it.

The porting of the AOCS Framework to Java was done in the "Real Time Java Project". The issues that should be borne in mind when using this document for the Java version of the AOCS framework are presented in the project web site currently located at the following address: www.aut.ee.ethz.ch/~pasetti/RealTimeJavaFramework/index.html. Some specific points to note are:

– The mechanism to link controllers to their inputs and outputs is different. It is no longer based on the data item concept that was not carried over to the Java version of the framework. It is instead based on the *data sink* and *data source* concept.

3.3 Notation

The pseudo-code examples in this document use a C++ notation.

The class diagrams use UML notation generated with the reverse engineering tool of the *Together* tool.

4 FRAMELET CONSTRUCTS

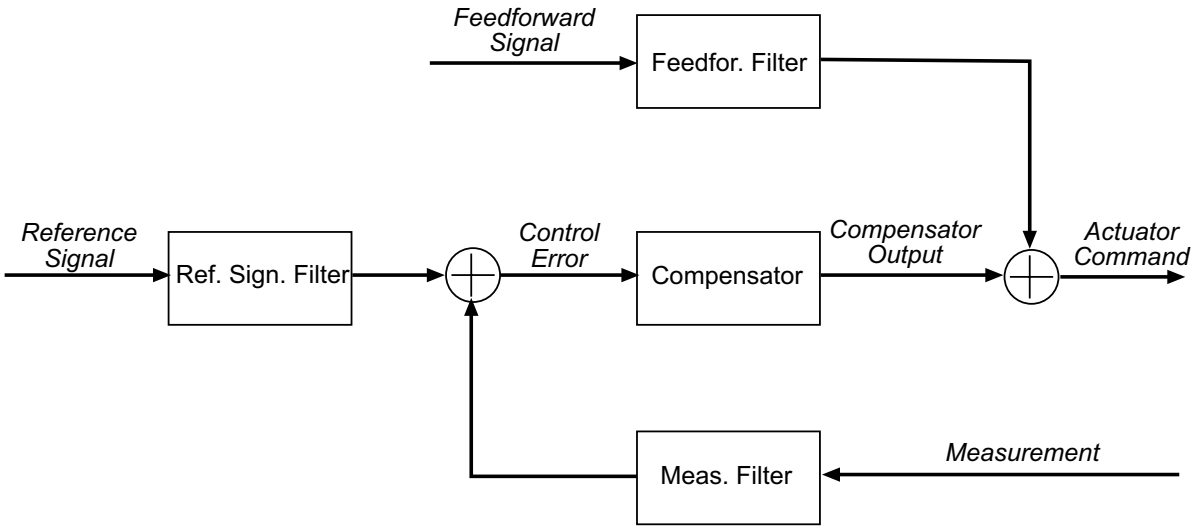
The architectural constructs exported by this framelet are listed in the following table:

CONTROLLER MANAGEMENT FRAMELET
Design Pattern
Controller Design Pattern: design pattern to separate the management of controllers from their implementation
Framelet Interfaces
`ControllerModeManager` : interface for controller mode managers
Framelet Core Components
`Controller` : class encapsulating a SISO controller `MimoController` : class encapsulating a MIMO controller (not provided by framework prototype) `ControllerManager` : class encapsulating a controller manager
Framelet Default Components
`FollowerControllerModeManager` : implementation of the controller mode manager interface based on the follower mechanism

5 CONTROLLER MODEL

Closed-loop controllers are widely used inside AOCS applications. They are primarily used to control the satellite attitude but can also be used to control the satellite orbit or the reaction wheel speeds.

The controller model assumed by the framelet is shown in the figure:



The figure shows a classical closed loop control system. The arrows represent signal flows and each arrow can represent several parallel flows. The boxes represent multi-input- multi-output transfer functions.

5.1 Controller Inputs

The inputs to a controller are:

- The *reference signals* that must be tracked by the controller
- The *measurements* from a sensor
- The *feedforward compensation signals* that are added to the control output and typically model known disturbances

The sensor measurement inputs are mandatory. The reference signals and feedforward compensation inputs are optional. If they are absent, the controller assumes that they are equal to zero.

5.2 Controller Outputs

A controller has three outputs:

- The *actuator commands* that represent the commands for the actuator
- The *compensator outputs* that represent the inputs to the compensator block
- The *control errors* that are obtained as the difference between filtered measurements and reference signals

Generation of the actuator command outputs is mandatory. Generation of the other two outputs is optional.

5.3 Controller Transfer Function Blocks

A controller includes four transfer function blocks:

- The *compensator* which implements the control law
- The *measurement filter* that filters the measurement signals
- The *feedforward filter* that filters the feedforward signals
- The *reference signal filter* that filters the reference signals

Explicit specification of the compensator transfer function is mandatory. The other transfer function blocks need not be specified by the user, in which case they are implicitly assumed to be equal to 1.

The measurement filter block often represents a state estimator.

5.4 Controller Configuration

A controller is implemented as a configurable component. At configuration time, the user links the component to the input and output signals and loads the transfer function blocks.

The links to the input signals are made through `readDataItem` objects. The links to the output signals are made through `writeDataItem` objects. The transfer function blocks are implemented as abstract control channels.

5.5 Closed and Open Loop Operation

A controller can operate in one of two modes:

- *Closed Loop*

  This is the nominal operating mode. The controller processes all the inputs and uses them to compute the input to the compensator.

- *Open Loop*

  In this mode, the feedback signals from the measurement sensors are ignored. The inputs to the compensator are computed exclusively from the feedforward and reference signals.

5.6 Operational Modes

The control and filtering algorithms used by a satellite controller are normally operational mode dependent. The framelet models this dependency by endowing the controller manager with mode-dependent behaviour.

5.7 Stability Checks

Closed loop controllers can become unstable. It is often possible to devise checks that the controller can perform upon itself to verify whether instability has set in. The framelet accordingly endows controller objects with the ability to perform a stability check upon themselves and report its result.

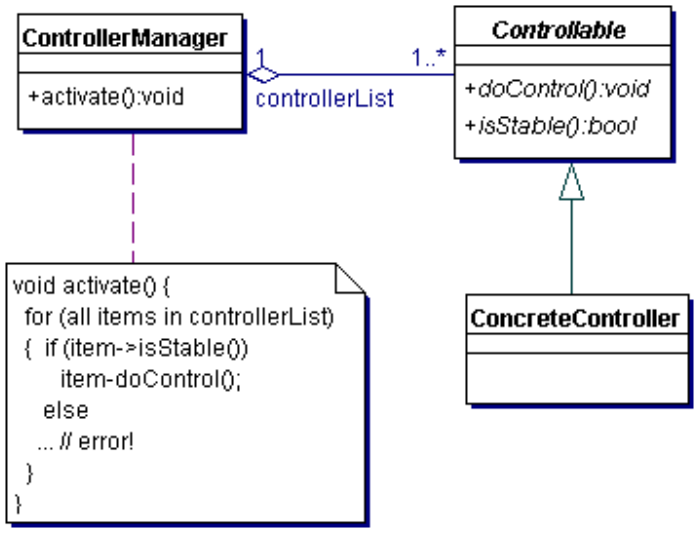
5.8 Limits on Actuator Commands

In order to prevent damage to actuators or to contain the effects of temporary instabilities, it is often useful to be able to constrain the commands to the actuator to remain within a pre-defined threshold. For this purpose, the framelet endows controller objects with methods to cap the commands sent to actuators.

6 THE CONTROLLER DESIGN PATTERN

This design pattern is introduced to address the problem of separating the management of closed-loop controllers from their implementations. It is based on the manager meta-pattern of RD2.

The pattern is illustrated in the following class diagram:



The abstract base class or interface `Controllable` represents a generic closed-loop control system of the kind discussed in the first part of this section. Its key method is `doControl` that directs the controller to acquire the sensor measurements, derive discrepancies with the current set-point, and compute and apply the commands for the actuators. Since closed-loop controllers can become unstable, a second key method `isStable` which is provided to ask a controller to check its own stability.

The controller manager component is responsible for maintaining a list of objects of type `Controllable` and for asking them to check their stability and, if the stability is confirmed, to perform their allotted control action.

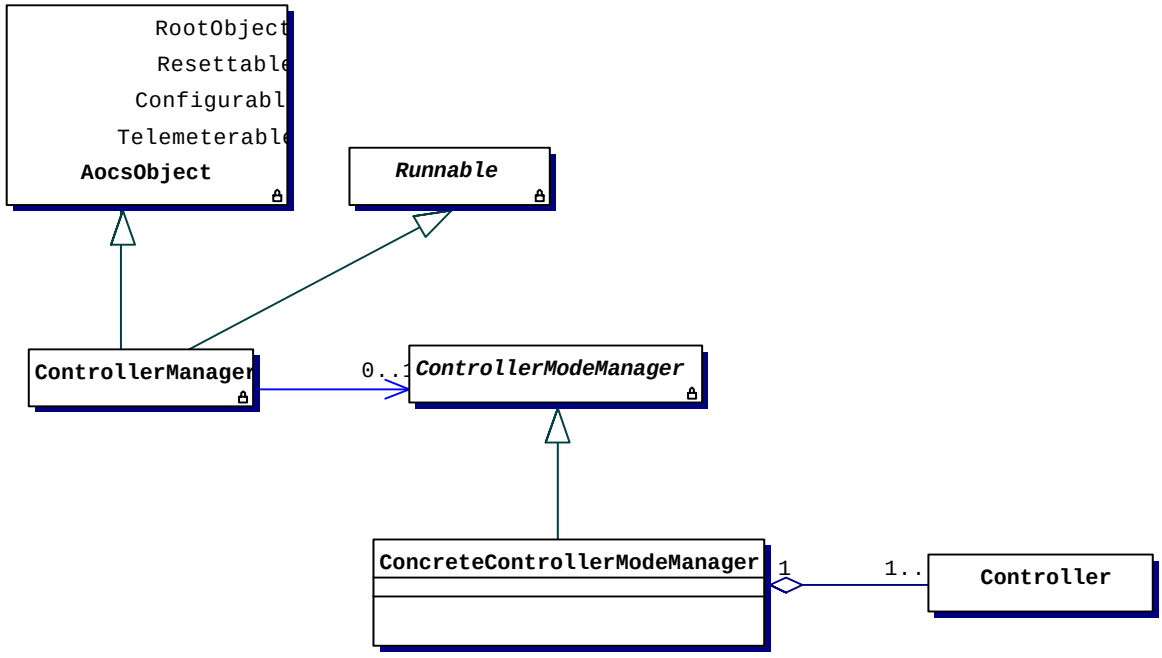
6.1 Instantiation of Controller Design Pattern

This controller design pattern is instantiated in the AOCS framework as follows:

- The controller manager is implemented as an active object and its `activate` method is the `run` method declared by interface `Runnable`.

- The separation between the controller management and the controller implementation is achieved not through an abstract interface (`Controllable` in the previous diagram) but through a configurable concrete class (class `Controller`, see section 7.1).
- In most cases, the control algorithms that are applied by each control loop depend on operational conditions. This is taken into account by making the controller manager mode-dependent. The controller mode manager then manages a single strategy represented by the list of controller objects that are managed by the controller manager.

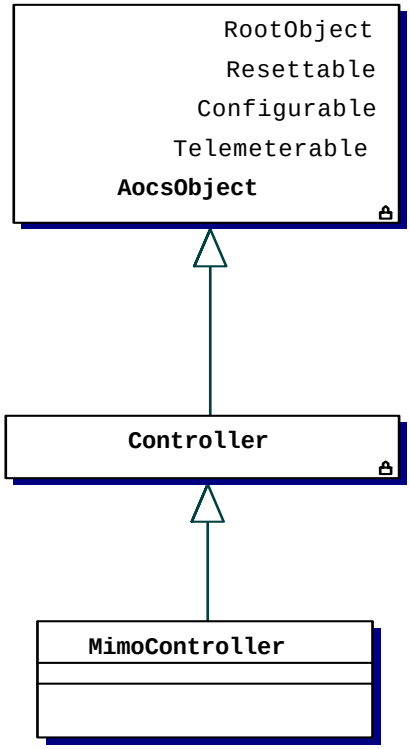
The resulting class diagram then becomes:



The mode manager is characterized by a dedicated abstract interface as discussed in section 8.1.

7 CONTROLLER OBJECTS

Objects that implement a controller are called *controller objects*. Their class structure is shown in the figure:

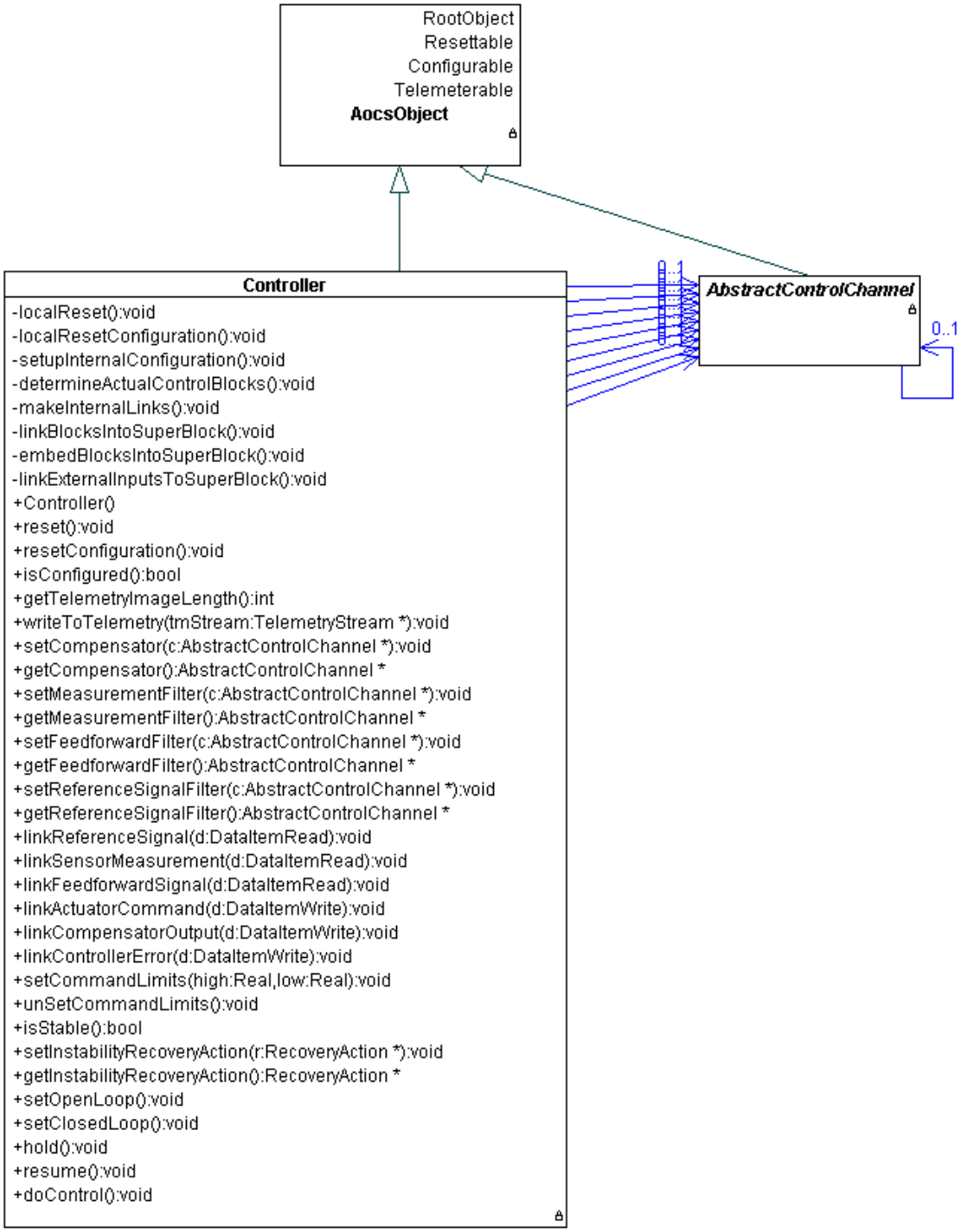


Two classes of controller objects are defined. Class `Controller` in the figure represents a SISO controller. Class `MimoController` instead represents a MIMO controller. SISO controllers are of course a special case of MIMO controllers and hence the same class might have served to represent both. However, given the wide prevalence of SISO controllers in AOCS applications, it was deemed appropriate to have a dedicated class to represent them so as to allow an efficient implementation.

Generally speaking, class `MimoController` offers the same functionalities as class `Controller` but allows specification of individual signals in the MIMO system.

7.1 The `Controller` Class

The `Controller` class is defined as follows:


RootObject
Resettable
Configurable
Telemeterable
AocsObject
Controller
-localReset():void
-localResetConfiguration():void
-setupInternalConfiguration():void
-determineActualControlBlocks():void
-makeInternalLinks():void
-linkBlocksIntoSuperBlock():void
-embedBlocksIntoSuperBlock():void
-linkExternalInputsToSuperBlock():void
+Controller()
+reset():void
+resetConfiguration():void
+isConfigured():bool
+getTelemetryImageLength():int
+writeToTelemetry(tmStream:TelemetryStream *):void
+setCompensator(c:AbstractControlChannel *):void
+getCompensator():AbstractControlChannel *
+setMeasurementFilter(c:AbstractControlChannel *):void
+getMeasurementFilter():AbstractControlChannel *
+setFeedforwardFilter(c:AbstractControlChannel *):void
+getFeedforwardFilter():AbstractControlChannel *
+setReferenceSignalFilter(c:AbstractControlChannel *):void
+getReferenceSignalFilter():AbstractControlChannel *
+linkReferenceSignal(d:DataItemRead):void
+linkSensorMeasurement(d:DataItemRead):void
+linkFeedforwardSignal(d:DataItemRead):void
+linkActuatorCommand(d:DataItemWrite):void
+linkCompensatorOutput(d:DataItemWrite):void
+linkControllerError(d:DataItemWrite):void
+setCommandLimits(high:Real,low:Real):void
+unSetCommandLimits():void
+isStable():bool
+setInstabilityRecoveryAction(r:RecoveryAction *):void
+getInstabilityRecoveryAction():RecoveryAction *
+setOpenLoop():void
+setClosedLoop():void
+hold():void
+resume():void
+doControl():void
0..1
AbstractControlChannel
0..1

The controller implements the structure shown in the figure of section 5. The controller transfer function blocks are the abstract control channels to which class controller holds multiple references.

The semantics of the methods specific to this class (i.e. not inherited from higher level base classes) are described in the table:

`doControl()`
A call to this method causes the current measurements to be propagated through the control system and the controller outputs to be accordingly updated. In implementation terms, a call to `doControl` translates into calls to the `propagate` methods on the abstract control channels representing the controller transfer function blocks.
`setOpenLoop(), setClosedLoop()`
Methods to switch between open and closed loop operational mode.
`isStable()`
A call to this method directs the controller object to perform an internal stability check. The method returns `true` if the check indicates that the controller is stable and returns `false` otherwise.
`hold(), resume()`
A call to `hold` puts the controller in hold mode. In this mode, the controller's inputs are no longer propagated and its internal state and external outputs are not updated. In implementation terms, a call to `hold` translates into a call to the method of the same name on the control channels representing the controller transfer function blocks. A call to `resume` brings the controller back to its normal operation. In implementation terms, a call to `resume` translates into a call to the method of the same name on the control channels representing the controller transfer function blocks.
`setCommandLimits(high, low), unSetCommandLimits()`
These methods set and remove limits to the maximum and minimum values of the actuator command. The specified actuator command is then constrained to the interval (low, high).

`setInstabilityRecoveryAction(r), getInstabilityRecoveryAction(r)`
Setter and getter methods for the recovery action associated to the controller instability condition. See section 8.
`setCompensator(abstractControlChannel), getCompensator()`
Getter and setter methods for the abstract control channel implementing the compensator transfer function (see section 5.3).
`setMeasurementFilter(abstractControlChannel), getMeasurementFilter()`
Getter and setter methods for the abstract control channel implementing the measurement filter transfer function (see section 5.3).
`setFeedforwardFilter(abstractControlChannel), getFeedforwardFilter()`
Getter and setter methods for the abstract control channel implementing the feedforward filter transfer function (see section 5.3).
`setReferenceSignalFilter(abstractControlChannel), getReferenceSignalFilter()`
Getter and setter methods for the abstract control channel implementing the reference signal filter transfer function (see section 5.3).
`linkReferenceSignal(dataItemRead)`
Set up the link to the data pool location from which the reference signal (see section 5.1) is to be retrieved.
`linkSensorMeasurement(dataItemRead)`
Set up the link to the data pool location from which the sensor measurement (see section 5.1) is to be retrieved.
`linkActuatorCommand(dataItemWrite)`
Set up the link to the data pool location to which the actuator command (see section 5.2) is to be written.
`linkControllerError(dataItemWrite)`
Set up the link to the data pool location to which the controller error (see section 5.2) is to be written.
`linkCompensatorOutput(dataItemWrite)`

	Set up the link to the data pool location to which the compensator output (see section 5.2) is to be written.

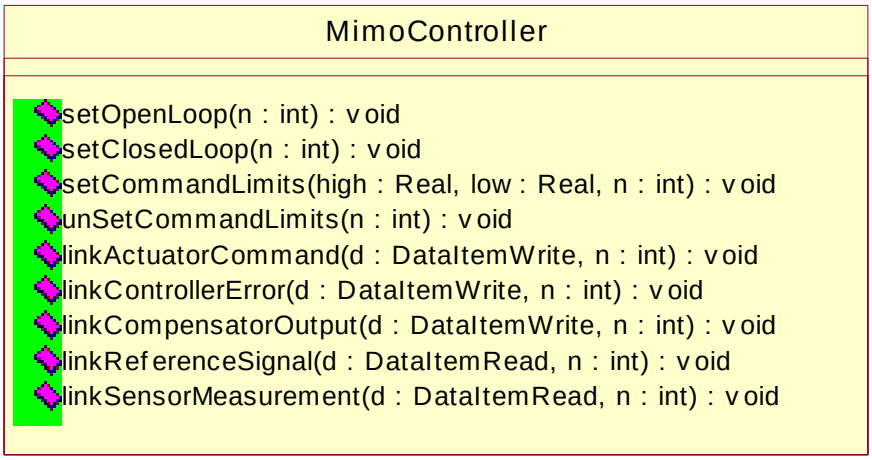
All transfer functions are implemented as abstract control channels. This incidentally allows the use of autocode modules from Xmath.

Recall that only the compensator transfer function must be defined by the user. For the other transfer functions, default unitary transfer functions are assumed if none are explicitly defined by the user.

Recall also that the only mandatory links are those to the actuator command and the sensor measurements. All other links are optional.

7.2 The `MimoController` Class

The `MimoController` class definition is shown in the figure:



Its methods are intended to mirror those of class `Controller` suitably modified to allow operations to be performed on individual channels. The semantics of the methods with class-specific implementation is as follows:

`setOpenLoop(n), setClosedLoop(n)`
These methods open or close the feedback loop from the n-th sensor input.
`setCommandLimits(high, low, n), unSetCommandLimits(n)`
These methods set and remove limits to the maximum and minimum values of the n-th actuator command. The specified actuator command is then constrained to the interval (low,high).
`linkReferenceSignal(dataItemRead, n)`
Set up the link to the data pool location from which the n-th reference signal (see section 5.1) is to be retrieved.
`linkSensorMeasurement(dataItemRead, n)`
Set up the link to the data pool location from which the n-th sensor measurement (see section 5.1) is to be retrieved.
`linkActuatorCommand(dataItemWrite, n)`
Set up the link to the data pool location to which the n-th actuator command (see section 5.2) is to be written.
`linkControllerError(dataItemWrite, n)`
Set up the link to the data pool location to which the n-th controller error (see section 5.2) is to be written.
`linkCompensatorOutput(dataItemWrite, n)`
Set up the link to the data pool location to which the compensator output (see section 5.2) is to be written.

Note that the MimoController is not implemented in the AOCS framework prototype.

7.3 Merging `MimoController` and `Controller`

In C++, the two classes `MimoController` and `Controller` could be merged into one by using the default argument mechanism offered by the language. The `Controller` class can be obtained from the `MimoController` interface by setting the signal specifier argument to 1.

7.4 The Telemetry Interface

Controller objects are telemetry objects because they inherit from `AocsData` the `telemeterable` interface.

The data sent to the telemetry stream by a controller object in each telemetry mode are summarized in the table:

TM Format	TM Data
Short	None
Normal	controller status (open/closed loop, held/not held)
Long	Normal TM + identification of all links to external signals
Debug	Same as Long TM

Controller objects are in a relationship of aggregation with the control channels implementing their transfer function blocks and hence calls to their telemetry methods are propagated to the control channels.

Note that the various telemetry formats are not implemented in the AOCS framework prototype.

7.5 The Reset Interface

Controller objects inherit from `AocsData` the `resettable` interface and must therefore implement the corresponding methods.

Method `reset` does not do any action on controller objects themselves but is propagated to the control channels implementing their transfer function blocks.

Method `resetConfiguration` unloads all the control channel blocks and cancels the links to the external signals.

Method `isConfigured` returns true if: the compensator transfer function has been defined, and the links to the actuator commands and sensor measurements have been set up.

8 THE CONTROLLER MANAGER

Controllers are passive objects that model a closed-loop control system. The object that is responsible for their activation and for controlling their operations is the *controller manager*. Its class structure is shown in the figure of section 6.1.

A control manager is an active object. It maintains a list of controllers that are represented as objects of type `Controller`. When the controller manager is activated by the scheduler, it goes through the list, checks whether the controllers are stable and, if they are, it calls their `doControl` method. Thus, the basic implementation of method `run` in class `ControllerManager` looks like this:

```
class ControllerManager : public AocsObject, public Runnable {

  ObjectListTemplate<Controller>*  cList;

public:
  . . .

  void run(AocsTime t){
  {

    Controller* c;
    for (t=cList->first(); ! cList ->isLast(); c= cList ->next())
    {
         if (c->isStable())
              c->doControl()
         else
         {    . . . // error! raise failure event
         }
    }
  }
  . . .
}
```

The detection of an instability condition is treated as a failure condition that is reported using the failure event mechanism. The recovery action associated to this failure is retrieved from the controller itself using its `getInstabilityRecoveryAction` method.

8.1 Controller Mode Manager

In most cases, the control algorithm to be applied to control a given variable in a satellite (the attitude, a reaction wheel speed, etc) will be different for different satellite operational

conditions. This variation is modeled by making the controller manager mode-dependent. The mode management design pattern of RD5 is used for this purpose.

A controller mode manager is defined by the following abstract interface:

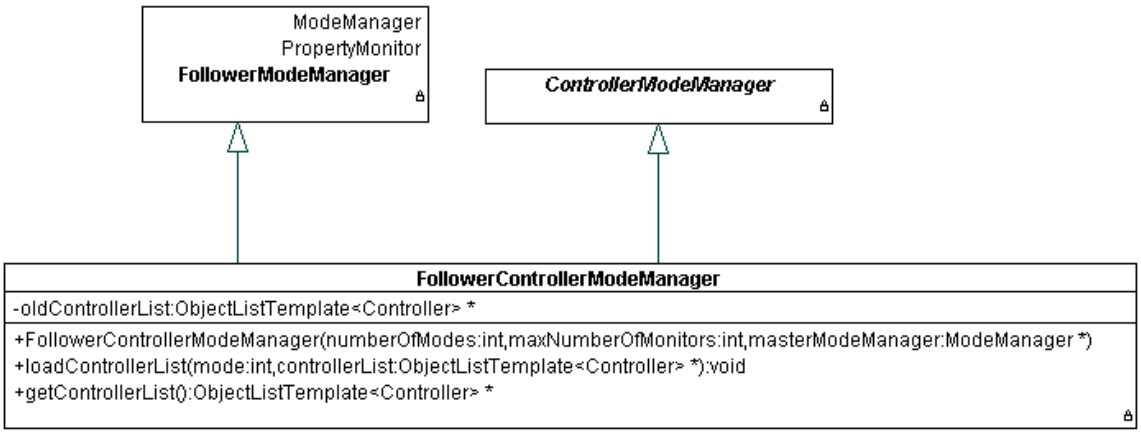
The semantics of the operations defined by this interface are summarized in the following table:

`getControllerList()`
This method is called by the controller manager to retrieve the currently valid controller list.
`loadControllerList(int n, ObjectListTemplate<Telemeterable>* c)`
This method is used to configure the controller mode manager. It associates controller list c to operational mode n.

Concrete controller mode managers are characterized by the mechanism that they use to decide which particular controller list should be returned by method `getControllerList` at any given point in time.

The prototype framework provides a default controller mode manager that is based on the follower mode manager. It maintains a set of controller lists, one for each operational mode, and changes operational mode in response to changes in a master mode manager.

The default controller mode manager is instantiated from the following class `FollowerControllerModeManager`:



Thus, the default controller mode manager uses the services offered by the generic follower mode manager component exported by the operational mode framelet.

The basic implementation of the `run` method of the controller manager now becomes:

```
class ControllerManager : public AocsObject, public Runnable {

  ControllerModeManager* modeManager;

public:
  . . .

  void run(AocsTime t){
  {
    ObjectListTemplate<Controller>*  cList;
    Controller* c;

    cList = modeManager->getControllerList();

    for (t=cList->first(); ! cList ->isLast(); c= cList ->next())
    {
         if (c->isStable())
              c->doControl()
         else
         {    . . . // error! raise failure event
         }
    }
  }
  . . .
}
```

8.2 The Telemetry Interface

Controller managers are telemetry objects because they inherit from `AocsData` the `telemeterable` interface.

The data sent to the telemetry stream by a telemetry manager in each telemetry mode are summarized in the table:

TM Format	TM Data
Short	none
Normal	instance ID of current controller list
Long	same as normal TM
Debug	long TM + instance ID of controller mode manager

Controller manager objects are in a relationship of aggregation with their controller mode manager and hence calls to their telemetry methods are propagated to their mode managers.

8.3 The Reset and Configurable Interfaces

The controller manager inherits from `AocsObject` the `Resettable` and `Configurable` interfaces and must therefore implement the corresponding method.

Controller managers have no internal state and therefore they do not provided a class-specific implementation of method `reset`.

A call to method `resetConfiguration` unloads the controller mode manager.

Method `isConfigured` returns true if the controller mode manager has been loaded.

9 FRAMELET HOT-SPOTS

This section classifies the framelet hot-spots defined in the previous sections of this document. The classification is as described in RD4.

9.1 Controller Mode Manager Plug-In

Name: **Controller Mode Manager Plug-In**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `ControllerManager` class (method `setControllerModeManager`)
Pre-defined Options: `FollowerControllerModeManager` component exported by this framelet.
Related Hot-Spots: none
Description Controller managers need a mode manager to supply them with the list of controller objects. This hot-spot allows the mode manager to be loaded in the controller manager.

9.2 Instability Recovery Action Plug-In

Name: **Instability Recovery Action Plug-In**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `Controller` class (method `setInstabilityRecoveryAction`)
Pre-defined Options: no recovery action is defined by default
Related Hot-Spots: none
Description When the controller manager finds that a controller is unstable, it raises a failure event. The

recovery action associated to this failure event is stored in the controller object itself. This hot-spot allows this recovery action to be loaded.

9.3 Controller List Plug-In

Name: **Controller List Plug-In**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `ControllerModeManager` class (method `setControllerList`).
Pre-defined Options: none
Related Hot-Spots: none
Description The controller mode manager maintains a list of controller objects. This hot-spot defines the point where a new list is loaded into the controller mode manager.

9.4 Compensator Plug-In

Name: **Compensator Plug-In**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `Controller` class (method `setCompensator`).
Pre-defined Options: none
Related Hot-Spots: none
Description Hot-spot where the abstract control channel implementing the compensator transfer function can be loaded into a controller.

9.5 Measurement Filter Plug-In

Name: **Measurement Filter Plug-In**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `Controller` class (method `setMeasurementFilter`).
Pre-defined Options: none
Related Hot-Spots: none
Description Hot-spot where the abstract control channel implementing the measurement filter transfer function can be loaded into a controller.

9.6 Feedforward Filter Plug-In

Name: **Feedforward Filter Plug-In**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `Controller` class (method `setFeedforwardFilter`).
Pre-defined Options: none
Related Hot-Spots: none
Description Hot-spot where the abstract control channel implementing the feedforward filter transfer function can be loaded into a controller.

9.7 Reference Signal Filter Plug-In

Name: **Reference Signal Filter Plug-In**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `Controller` class (method `setReferenceSignalFilter`).
Pre-defined Options: none
Related Hot-Spots: none
Description Hot-spot where the abstract control channel implementing the reference signal filter transfer function can be loaded into a controller.

9.8 Actuator Command Link

Name: **Actuator Command Link**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: method call in `Controller` class (method `linkActuatorCommand`).
Pre-defined Options: none
Related Hot-Spots: none
Description Hot-spot to set up the link with the data pool location where the actuator command is to be written by the controller object.

9.9 Compensator Command Link

Name: **Compensator Command Link**
*Visibility Level**:*** framework-level
Adaptation Time: run-time
Adaptation Method: method call in `Controller` class (method `linkCompensatorCommand`).
Pre-defined Options: none
Related Hot-Spots: none
Description Hot-spot to set up the link with the data pool location where the compensator command is to be written by the controller object.

9.10 Control Error Link

Name: **Control Error Link**
*Visibility Level**:*** framework-level
Adaptation Time: run-time
Adaptation Method: method call in `Controller` class (method `linkControlError`).
Pre-defined Options: none
Related Hot-Spots: none
Description Hot-spot to set up the link with the data pool location where the control error is to be written by the controller object.

9.11 Reference Signal Link

Name: **Reference Signal Link**

Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: method call in `Controller` class (method `linkReferenceSignal`).
Pre-defined Options: none
Related Hot-Spots: none
Description Hot-spot to set up the link with the data pool location from which the reference signal is to be read by the controller object.

9.12 Sensor Measurement Link

Name: **Sensor Measurement Link**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: method call in `Controller` class (method `linkSensorMeasurement`).
Pre-defined Options: none
Related Hot-Spots: none
Description Hot-spot to set up the link with the data pool location from which the sensor measurement is to be read by the controller object.

9.13 Feedforward Signal Link

Name: **Feedforward Link**
Visibility Level: framework-level
Adaptation Time: run-time

Adaptation Method: method call in `Controller` class (method `linkFeedforward`).
Pre-defined Options: none
Related Hot-Spots: none
Description Hot-spot to set up the link with the data pool location from which the feedforward signal is to be read by the controller object.