RECONFIGURATION 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 reconfiguration management framelet. This framelet proposes an architecture to handle reconfigurations of AOCS functionalities. The framelet enhances reusability because it decouples the task of managing reconfigurations from the reconfiguration algorithm and from the use of the functionality by other objects.

Written By: A. Pasetti

Date: 30 April 2002

Issue: 2.2

Reference: SWE/99/AOCS/015

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 (2000), *AOCS Framework – Concept Level Description*, AOCS Framework Document ref. SWE/99/AOCS/004

RD3 A. Pasetti (2001), *Software Frameworks and Embedded Control Systems*, LNCS Series, Springer-Verlag, To appear in Dec. 2001

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
OBDH	On-Board Data Handling system (aka as OBDS)
NM	Normal Mode
NTT	Non-Time-Tagged
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
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 *AOCS reconfiguration 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 handle reconfiguration of AOCS functionalities. The framelet enhances reusability because it decouples the task of *managing reconfigurations* from the *reconfiguration algorithm* and from *the use of the functionality* by other objects.

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 reconfiguration management.

The architecture proposed here follows the general concept outlined in RD2.

In comparing the present document with RD2, readers 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.

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:

– Events in the Java framework are implemented using the Java event mechanism.

– Reconfiguration managers in the C++ framework expose their reconfiguration state as a monitorable property. This is not the case in the Java framework.

– As a consequence of the previous point, the change event repository hot-spot (section 11.3) does not exist in the Java framework.

3.3 Notation

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

UML class diagrams were obtained with the *Together* tool.

4 FRAMELET CONSTRUCTS

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

RECONFIGURATION MANAGEMENT FRAMELET
Design Patterns
Reconfiguration Design Pattern : pattern to make handling of reconfigurable objects independent of their reconfigurability
Framelet Interfaces
`Reconfigurable` : interface to be implemented by all reconfiguration managers.
Framelet Core Components
`ConfigurationState` : encapsulation of the state of a reconfiguration group
Framelet Default Components
`ReconfigurerHelper` : helper object to handle the management of a reconfiguration group `BasicUnitReconfigurer` : reconfiguration manager for a group of identical objects `RwSet` : reconfiguration manager for a set of 4 identical reaction wheels

The components listed above are those envisaged for the prototype version of the AOCS framework. Later versions may offer a richer set of default implementations of reconfiguration managers.

5 RECONFIGURATION MODEL

If the same functionality can be implemented in two or more independent ways, then the functionality is said to be *redundant*.

A redundant functionality can be *reconfigured.*

To reconfigure an object means to switch between different independent implementations of the same functionality offered by the object.

Reconfiguration usually occurs in response to detection of an error: if one implementation of a functionality is found to be faulty, reconfiguration makes the functionality available again by switching from a faulty to a (hopefully) correct implementation.

5.1 Reconfiguration Group

Functionalities in the AOCS software are implemented as services (method calls) provided by objects. The functionality with respect to which reconfiguration takes place is called the *reconfigurable functionality*. The reconfigurable functionality is often represented by an abstract interface.

A *reconfiguration group* is a set of objects that together offer a redundant functionality. A reconfiguration group can be the object of a reconfiguration.

A *redundant object* is an object belonging to a reconfiguration group.

The *order* of the reconfiguration group is the number of independent, functionally equivalent, configurations offered by the group.

Configurations in a reconfiguration group may be *ranked* according to the performance level with which they implement the group's functionality.

A configuration in a reconfiguration group is *marked* either "healthy" or "unhealthy". When a reconfiguration takes place, the configuration the group is configuring away from is marked as "unhealthy".

5.2 Triggering of Reconfigurations

The reconfiguration group manages the reconfiguration process but does *not* decide *when* a reconfiguration should take place. As discussed in section 6, a reconfiguration group is represented by a reconfiguration manager object. This object exposes, among others, a `reconfigure` method. A call to this method will cause a reconfiguration to take place (assuming that there are still healthy configuration available). Thus, the reconfiguration

manager offers reconfigurations as a service to other objects. Its task is to hide the details of *how* the reconfiguration takes place.

5.3 Representation of Configurations

Configurations within a reconfiguration group are represented by integers in the range [0,n-1] where n is the group's reconfiguration order.

The rank of a configuration is also represented by an integer in the range [1,m] where m is the highest configuration rank in the configuration group.

5.4 Management of Objects' Health Status

As mentioned above, when a reconfiguration takes place, the configuration the group is configuring away from is marked as "unhealthy". It must be stressed that the unhealthy marking applies exclusively to the abstract configuration and *not* to any concrete objects that may be implementing the configuration.

Thus, for instance, if a reconfiguration occurs from a primary unit to its redundant back-up, the unit that is being configured out is *not* marked unhealthy. It is not the job of the reconfiguration manager to set the health status of AOCS objects in general and of unit objects in particular. However, it may happen that the reconfiguration is commanded *as a consequence* of some other entity (such as the failure detection manager) having marked a unit or an object unhealthy.

5.5 Notification of Reconfigurations

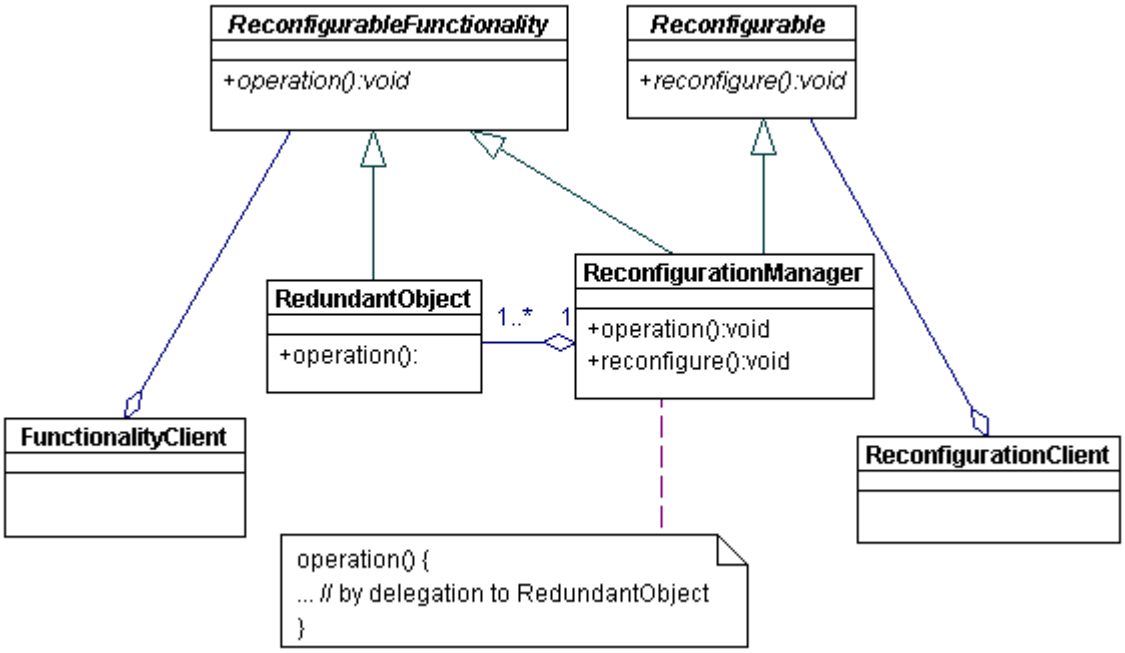
RD2 proposed a mechanism whereby objects that are excluded from the currently active configuration can advertise the fact through a change notification. The notification might allow a reconfiguration manager to autonomously start a reconfiguration in response to reconfigurations occurring in other, intersecting, reconfiguration groups. This mechanism has been dropped from the framelet since the responsibility of the reconfiguration managers should be limited to the *management* of the reconfigurations and not to the decision as to *when* a reconfiguration should take place (see also section 5.4).

5.6 Reconfiguration Examples

See RD2 for some examples of reconfiguration groups.

6 THE RECONFIGURATION DESIGN PATTERN

This design pattern is introduced to address the problem of separating the management of a reconfiguration group from the provision of the reconfigurable functionality. This pattern is illustrated by the following class diagram:



The redundant objects are instantiated from class `ReconfigurableObject`. The reconfigurable functionality they offer is encapsulated in the abstract interface `ReconfigurableFunctionality`. The reconfigurations are managed by *Reconfiguration Manager*. This component is characterized by interface `Reconfigurable` whose key method is `reconfigure`. A call to `reconfigure` triggers a reconfiguration: the reconfiguration manager chooses the highest-ranking configuration among the alternative configurations that are still marked "healthy". The configuration that is abandoned is automatically marked "unhealthy". Components that are responsible for performing reconfiguration (class `ReconfigurationClient` in the diagram) thus see the reconfiguration as an instance of type `Reconfigurable`.

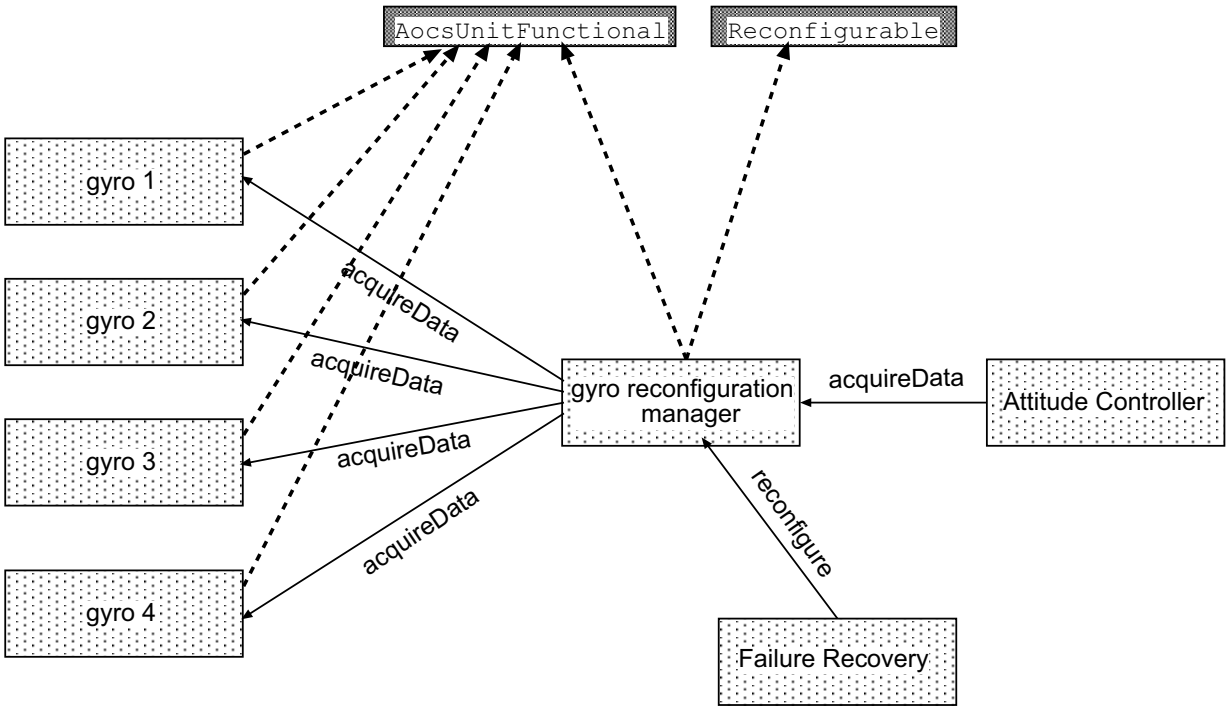
The reconfiguration manager also implements interface `ReconfigurableFunctionality` which makes it "look like" a redundant object. The reconfiguration manager implements the methods declared by `ReconfigurableFunctionality` through delegation to the redundant objects in the reconfiguration group. The reconfiguration manager must be able to

act as the sole functional interface between the reconfiguration group and the users of the reconfigurable functionality. Components that use the reconfiguration functionality (class `FunctionalityClient` in the diagram) thus see the reconfiguration manager as an instance of type `ReconfigurableFunctionality`.

The implementation of method reconfigure in interface `Reconfigurable` is one of the framework hot-spot (*reconfigurable hot-spot*) as it is here that AOCS applications define their application-specific reconfiguration logic.

As a concrete example, consider again the gyro reconfiguration group example described in See RD2.. The gyros – as AOCS units – would be characterized by interface `AocsFunctional` which then becomes the reconfigurable functionality. A typical user of this functionality could be an attitude controller that needs 3-axis rate information. A typical component responsible for performing reconfiguration could be a failure recovery component that would trigger a reconfiguration in response to the detection of a gyro failure.

The architecture for this example is shown in the figure using an informal notation:



The lightly shaded boxes represent objects. The darker boxes are abstract interfaces. The dashed arrows are implementation links (thus, gyro 1 is an object that implements interface `AocsUnitFunctional`). The solid arrows represent association links.

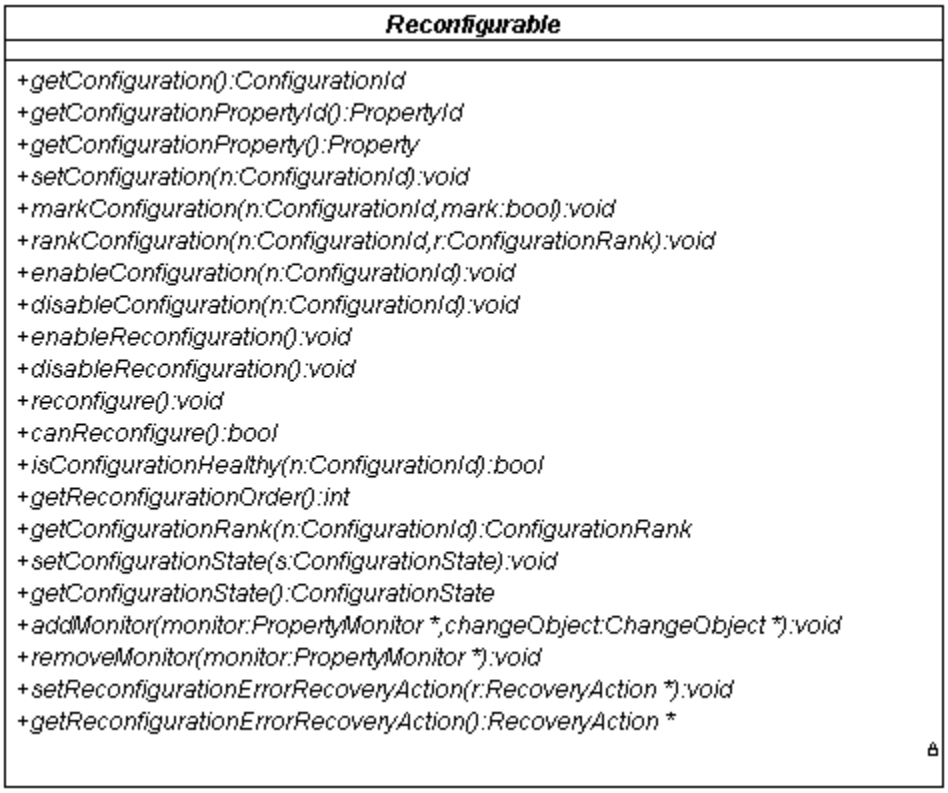
The figure shows that reconfiguration manager has two faces: it has an `AocsUnitFunctional` face that it exposes towards the attitude controller (to which it supplies the rate estimate obtained from merging the rate measurements from the three active gyros) and it has a `Reconfigurable` face that it exposes towards the failure recovery manager (to which it supplies a method to reconfigure the set of four gyros to exclude faulty units).

This example illustrates a very common case in which a reconfiguration manager handles reconfiguration across real units. In that case, the reconfigurable interface is `AocsUnitFunctional` and the reconfiguration manager thus becomes a fictitious unit.

Instantiation

6.1 Instantiation of Reconfiguration Pattern

The AOCS framework instantiates the reconfiguration design pattern by specifying the `Reconfigurable` interface as follows:



The semantics of the methods defined by this interface are summarized in the table below:

`getConfiguration()`
Returns the current configuration. The configuration is indicated by an number in the range [0,n-1] where n is the order of the configuration group.
`getConfigurationPropertyId(), getConfigurationProperty()`
The current configuration is a monitorable property. The first method returns its property identifier and the second returns the current configuration as a property object.
`setConfiguration(n)`
This method forces the configuration to n. The reconfiguration is performed even if the target configuration had previously been marked “unhealthy” or if reconfiguration are currently disabled.
`markConfiguration(n, mark), isConfigurationHealthy(n)`
The first method marks configuration n as healthy if `mark` is `true` and as unhealthy

if `mark` is `false`. The second method can be used to verify the health status of the n-th configuration.
`rankConfiguration(n, rark), getConfigurationRank(n)`
The first method sets the rank of configuration `n` to `rank`. The second method returns the rank of the n-th configuration.
`enableConfiguration(n), disableConfiguration(n)`
Enable or disable reconfiguration to the n-th configuration. The effect of this method is the same as marking the n-th configuration as healthy or unhealthy.
`enableReconfigurations(), disableReconfigurations()`
Enable or disable reconfigurations. A call to method `reconfigure` has no effect when reconfigurations are disabled. However, it does not interfere with calls to method `setConfiguration`.
`reconfigure()`
Perform a reconfiguration to the highest-ranking healthy configuration that has not been disabled. Reconfigurations are performed only if they are enabled. If no reconfiguration is possible, the method returns without taking any action. The generation of a failure event should be done by the caller of the `reconfigure` method.
`canReconfigure()`
Returns `true` if further reconfigurations are possible. Calls to `reconfigure` should normally be preceded by calls to `canReconfigure` to verify that the reconfiguration is possible.
`getConfigurationState(),setConfigurationState()`
Getter and setter methods for the configuration state. See section 9.
`getConfigurationOrder()`
Return the order of the reconfiguration group.
`addMonitor(), removeMonitor()`
The current configuration is a monitorable property. These methods allow monitors to register their interest in the current configuration and to be automatically notified in case of configuration changes.
`setIllegalConfigurationRecoveryAction(),getIllegalConfigurationRecove`

ryAction()
Attempts to operate on non-existent configurations give rise to the generation of a failure event. These are the getter and setter methods for the associated recovery action.

The standard procedure for commanding a reconfiguration is as follows:

- a call to method `canReconfigure` is performed to verify that the reconfiguration is possible
- if `canReconfigure` reports that no reconfigurations are possible, then contingency action is taken. This could for instance consist of generating a failure event
- if `canReconfigure` reports that further reconfigurations are possible, a call to `reconfigure` is issued to perform the reconfiguration.

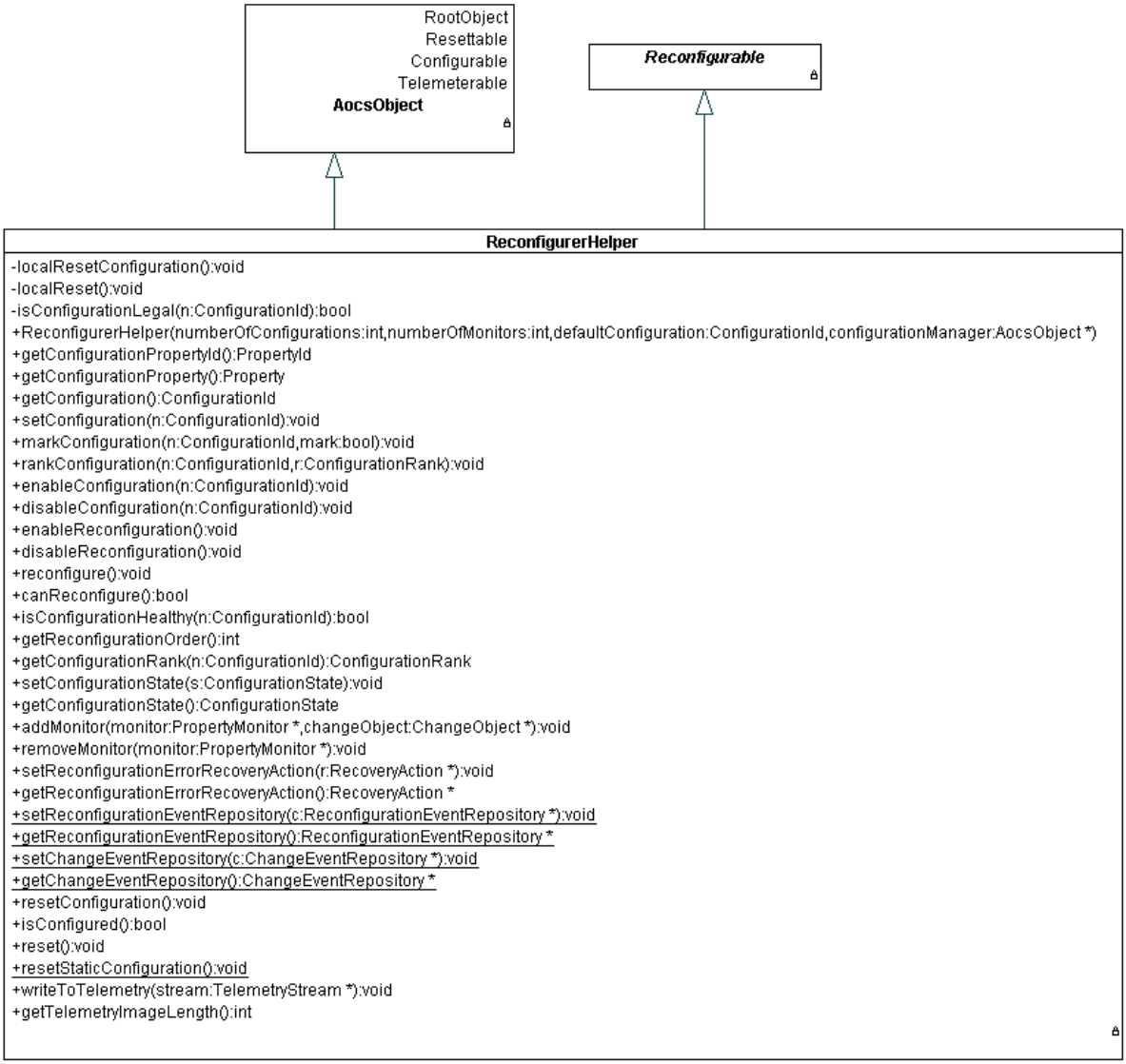
7 THE RECONFIGURATION MANAGER HELPER

The reconfigurable functionality is usually represented by an abstract interface (as shown in the first figure of this section) but might also be represented by a concrete or abstract class. Since the design rules for the AOCS framework forbid the use of multiple inheritance of implementation, `Reconfigurable` was implemented as a pure interface so as to make it possible for reconfiguration managers to inherit both from it and from any class representing the reconfigurable functionality.

There are however certain standard operations pertaining to the management of a reconfiguration group that apply in general to all reconfiguration groups and it would be convenient to capture their implementation and make it available to reconfiguration managers. For this purpose a reconfiguration manager helper – the `ReconfigurerHelper` object – was introduced. Its class definition is shown in the next page. The public methods that are specific to this class (ie. not inherited from other classes) are described in the table:

`ReconfigurerHelper(n, m, d, manager)`	
	Constructor that specifies the number of configuration n, the maximum number m of monitors for the configuration property, the default configuration d and the configuration manager `manager` to which this helper is attached.
`setReconfigurationEventRepository(),getReconfigurationEventRepository`	
	Reconfigurations are recorded as reconfiguration events (see section 8). The reconfigurer helper therefore needs access to the corresponding repository. These are the getter and setter methods to the reconfiguration event repository.
`setChangeEventRepository(),getChangeEventRepository()`	
	The current configuration is a monitorable property. If monitors have registered their interest in it, change events may have to be generated as a result of reconfigurations. These are the getter and setter methods for the associated change event repository.

Essentially, the reconfigurer helper manages a set of n identical configurations and implements all the methods defined by interface `Reconfigurable` for this situation. Specific reconfiguration managers would normally create an internal instance of the reconfigurer helper and delegate to it implementation of as many of the `Reconfigurable` operations as is possible. Classes `BasicUnitReconfigurer` and `RwSet` provide two examples of how this delegation is performed in practice.



7.1 The Telemetry Interface

The reconfigurer helper is a telemetry objects because it inherits from `AocsObject` the `telemeterable` interface. The data sent to the telemetry stream by it object in each telemetry mode are summarized in the table:

TM Format	TM Data
Short	none

Normal	current configuration
Long	normal TM + health status and rank of all configurations
Debug	long TM + instance ID of recovery action and repository events

7.2 The Reset and Configurable Interface

The reconfigurer helper object inherits from `AocsObject` the `Resettable` and `Configurable` interfaces and must therefore implement the corresponding method.

The reconfigurer helper defines a class-specific `Reset` method that:

- enables all reconfigurations,
- marks all configutations as healthy
- set the rank of all configurations to 1

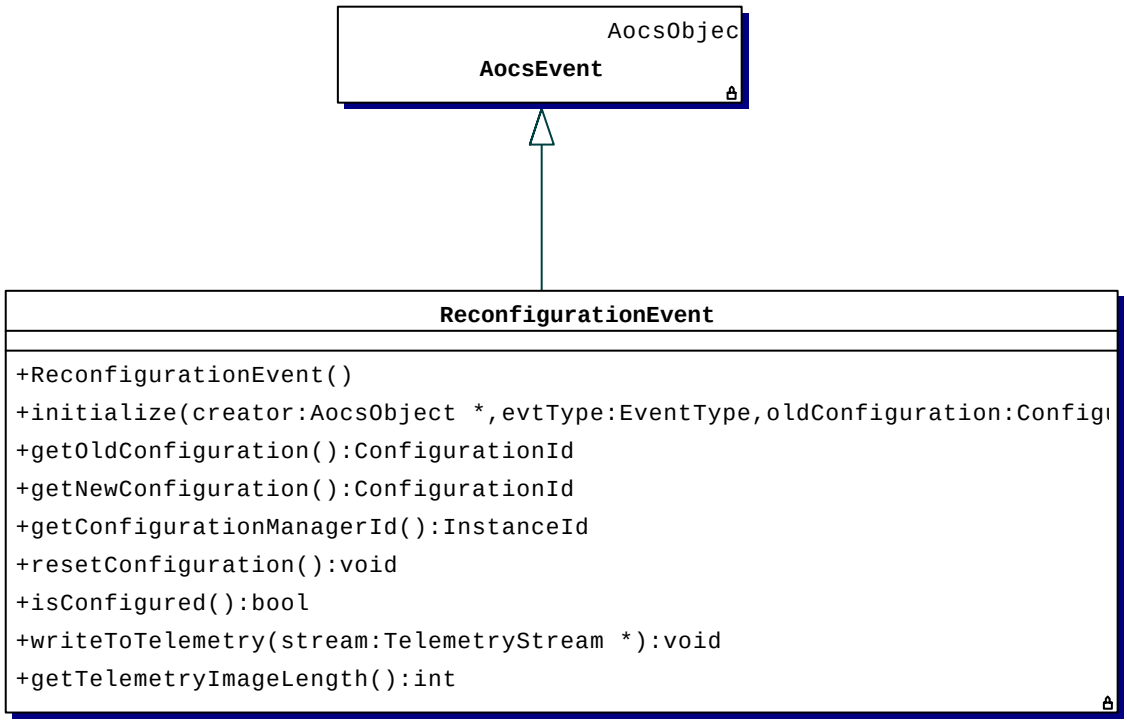
The reconfigurer helper defines a class-specific `resetConfiguration` method that unloads the recovery action objects and resets the configuration of the object lists associated to the monitoring of the configuration property.

The reconfigurer helper defines a class-specific method `isConfigured` that returns `true` if:

- the recovery actions have been loaded
- the event repositories have been loaded
- if the object lists associated to the monitoring of the configuration property are configured

8 RECONFIGURATION EVENTS

Reconfigurations are recorded in events of type `ReconfigurationEvent`. The class diagram for this class is:



Thus, the reconfiguration event adds the following attributes to those defined by the base class `AocsEvent`:

- `oldConfiguration` : the configuration before the reconfiguration took place.
- `newConfiguration` : the configuration after the reconfiguration took place.
- the object identifier of the reconfiguration manager that performed the reconfiguration

Creation of reconfiguration events is the responsibility of the reconfiguration manager.

Reconfiguration events are created for reporting purposes only. They provide a vehicle through which reconfigurations can be recorded for possible reporting to the ground in the telemetry stream. Objects that need to observe reconfigurations should do so through by registering their interest in the configuration property exposed by each reconfiguration manager.

Reconfiguration events are generated in response to the occurrence of a reconfiguration. Sometimes, reconfiguration requests cannot be fulfilled because there are no other healthy configurations in a configuration group. This situation, too, is recorded as a reconfiguration event.

8.1 The Telemetry Interface

Reconfiguration events are telemetry objects because they (indirectly, through `AocsEvent`) inherit the `telemeterable` interface.

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

TM Format	TM Data
Short	the new configuration indicator + old configuration indicator
Normal	short TM + instance identifier of reconfiguration manager
Long	same as normal TM
Debug	same as long TM

8.2 The Reset and Configurable Interface

Reconfiguration event objects inherit from `AocsObject` the `Resettable` and `Configurable` interfaces and must therefore implement the corresponding method.

Reconfiguration events have no dynamic state associated to them and therefore they do not define a class-specific `reset` method.

Reconfiguration events define a class-specific `resetConfiguration` method that resets all event attributes to zero. Method `isConfigured` returns true if the new and old configuration indicators are equal or if the reconfiguration manager reference is NULL.

9 CONFIGURATION STATE OBJECTS

A reconfiguration is usually the result of a detected or suspected fault in an object that, through the reconfiguration, is excluded from the normal flow of AOCS data. Reconfiguration information should therefore be preserved across software and hardware resets in order to allow safe autonomous re-initialization of the AOCS software.

Configuration information is stored in an object of type `ConfigurationState`. This object encapsulates the configuration of a reconfiguration group by storing the following information:

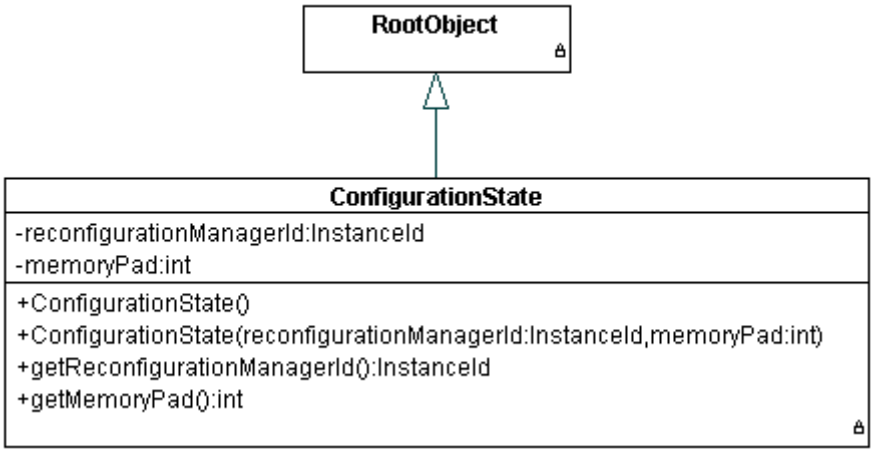
- Configurations that have been marked "unhealthy"
- The current configuration

Individual configuration managers may add more specific information.

Storage of configuration information is done using the *memento design pattern* from RD1.

Since the exact format of the configuration state data cannot be specified in advance, the internal structure of the `ConfigurationState` type remains correspondingly open. Essentially, configuration state object reserve a fixed amount of memory that reconfiguration managers can use to store class-specific information. In the framework prototype implementation, this "memory pad" area consists of a single integer.

The configuration state class diagram is:



The public methods that are specific to this class (ie. not inherited from other classes) are described in the table:

`getReconfigurationManagerId()`
Returns the instance identifier of the reconfiguration manager whose state is recorded in the object.
`getMemoryPad()`
Return the integer encoding the configuration state of the reconfiguration manager. The latter is the only object that can decode the information in the memory pad .

Configuration state objects are created by reconfiguration managers. The system manager object holds a list of all the reconfiguration managers and of their configuration state objects. Configuration managers register with the system manager – by calling its method `addReconfigurationManager` – as part of their initialization process.

The system manager exposes a method – `configurationStateChange` – that can be invoked by a reconfiguration manager to notify the system manager that a change in configuration state has occurred.

In case of system reset, the system manager will then pass its latest copy of configuration state object to each reconfiguration manager thus reestablishing the configurations that prevailed before the system reset was commanded. If the system reset is to preserve configuration information across complete reboots or other destructive events, it could be endowed with the capacity to store configuration information in some kind of permanent memory area external to the AOCS computer.

10 DEFAULT RECONFIGURATION MANAGERS

The framework prototype offers two default reconfiguration managers as described in the next two subsections.

10.1 The Basic Unit Reconfigurer Object

In a common situation a primary AOCS unit is backed up by one or more identical redundant units. The `BasicUnitReconfigurer` can be used as a reconfiguration manager for this situation. It handles reconfigurations across a set of n identical units. The reconfiguration management is delegated to a reconfigurer helper object (see section 7).

The basic unit reconfigurer internally treats units as objects of type `AocsUnit`. This is necessary because it needs housekeeping access to them to switch them on and off and to initialize them. Its reconfigurable functionality, however, is represented by the `AocsUnitFunctional` interface.

When a reconfiguration is due to take place the basic unit reconfigurer performs the following actions:

- switch off the unit being configured out
- update the configuration indicator
- switch on the unit being configured in
- initialize the unit being configured in

Thus the model assumed by the reconfigurer is one of so-called *cold redundancy*.

10.2 The Reaction Wheel Set Reconfiguration Manager

In another common situation, four reaction wheels are flown on a satellite in a skewed configuration that allows any given three to be used to control the satellite attitude. The `RwSet` reconfiguration manager is provided to manage the reconfigurations across the four redundant sets of wheels.

Configuration i – with i ranging from 1 to 4 – corresponds to the situation where the i-th wheel is inactive.

The functional inputs and outputs to reaction wheels are the wheel speeds and the wheel torques. The functional inputs and outputs to object `RwSet` are the wheel angular momentum vector expressed in spacecraft reference frame and the wheel torque vector also expressed in spacecraft reference frame.

The `RwSet` object maintains a set of matrices that allow it to map the wheel speed to the wheel angular moment and to convert the torque requests around spacecraft axes to torque requests around the axes of the currently active reaction wheels.

When a reconfiguration is due to take place `RwSet` performs the following actions:

- switch off the wheel being configured out
- update the configuration indicator
- switch on the wheel being configured in
- initialize the wheel being configured in

11 FRAMELET HOT-SPOTS

This section classifies the framelet hot-spots defined in the previous sections of this document. The classification is as described in RD3.

11.1 Reconfigurable Hot-Spot

Name: Reconfigurable Hot-Spot
Visibility Level: framework –level
Adaptation Time: compile-time
Adaptation Method: implementation of interface `Reconfigurable`
Pre-defined Options: `ReconfigurerHelper` component and default reconfiguration manager components (see section 10) exported by this framelet
Related Hot-Spots: none
Description Reconfiguration managers are characterized by the `Reconfigurable` interface. This is the hot-spot where specific reconfiguration managers are defined.

11.2 Recovery Action Plug-In for Illegal Configurations

Name: Recovery Action Plug-In for Illegal Configurations
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in objects implementing interface `Reconfigurable` (method `setIllegalConfigurationRecoveryAction`)
Pre-defined Options: no recovery action is defined by default
Related Hot-Spots: none
Description When an operation is attempted on a reconfiguration manager using an illegal value of

configuration indicator, then a failure event is generated. A recovery action should be associated to this failure. This hot-spot allows this recovery action to be loaded.

11.3 Change Event Repository Plug-In

Name: Change Event Repository Plug-In
Visibility Level: framelet-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `ReconfigurerHelper` class (method `setChangeEventRepository`)
Pre-defined Options: `ChangeEventRepository` component exported by inter-component communication framelet.
Related Hot-Spots: none
Description Reconfigurer managers whose configuration is being monitored by other components log changes in their configuration property as events stored in the change event repository. This hot-spot allows the change event repository component to be loaded. Note that this component is loaded as a `static` reference. The event repository is loaded by the reconfigurer helper object (see section 7) to which management of reconfigurations is delegated by reconfiguration managers.

11.4 Reconfiguration Event Repository Plug-In

Name: Reconfiguration Event Repository Plug-In
Visibility Level: framelet-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `ReconfigurerHelper` class (method `setReconfigurationEventRepository`)
Pre-defined Options: `ReconfigurationEventRepository` component exported by inter-component communication framelet.
Related Hot-Spots: none

Description

Reconfigurer managers log reconfigurations as reconfiguration events stored in the reconfiguration event repository. This hot-spot allows the event repository component to be loaded. Note that this component is loaded as a `static` reference. The event repository is loaded by the reconfigurer helper object (see section 7) to which management of reconfigurations is delegated by reconfiguration managers.