AOCS FRAMEWORK – CONCEPT LEVEL 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 design principles and the overall architecture of the AOCS software. The architecture is described at the concept level. For each architectural issue, a baseline solution is identified. An overview of each framelet is also given.

Written By: A. Pasetti

Date: 30 April 2002

Issue: 3.3

Reference: SWE/99/AOCS/004

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), *Description of a Satellite Attitude and Orbit Control Subsystem*, AOCS Framework Document, SWE/99/AOCS/001

RD3 M. Fayad, D. Schmidt, R. Johnson, *Application Frameworks*, Wiley Computing Publishing, 1999

RD4 Society of Automotive Engineers, *Generic Open Architecture (GOA) Framework*, SAE Document AS4893, Jan. 1996

RD5 Deleted

RD6 R. Englander (1997), *Developing Java Beans*, O'Reilly

RD7 A. Burns, A. Wellings, *HRT-HOOD:A Structured Design Method for hard Real Time Systems*, Real-Time Systems, 6, p. 73-114, 1994

RD8 T. Vardanega, J. van Katwijk (1999), *A Software Process for the Construction of Predictable On-Board Embedded Real-Time Systems*, in *Software – Practice and Experience*, March 1999

RD9 C. Szyperski, (1998), *Component Software—Beyond Object-Oriented Programming*. Reading, Massachusetts: Addison-Wesley

RD10 A. Gosling, (1996), *The Java Programming Language*, Addison-Wesley

RD11 D. Herity (1998), *C++ in Embedded Systems: Myth and Reality*, Embedded Systems Programming, Feb. 1998

RD12 P. Plager, (1997), *Embedded C++: An Overview*, Embedded Systems Programming, Dec. 1997

RD13 T. Vardanega, *Tool Support for the Construction of Statically Analyzable Hard Real-Time Systems in Ada*, ftp://ftp.estec.esa.nl/pub/ws/wsd/hrt/

RD14 T. Vardanega, *Experience with the Development of Hard Real-Time Embedded Software in Ada*, ftp://ftp.estec.esa.nl/pub/ws/wsd/hrt/

RD15 T. Vardanega, *On the development of Hard Real-Time Software for On-Board Control Systems*, ftp://ftp.estec.esa.nl/pub/ws/wsd/hrt/

RD16 T. Vardanega, *On the Use of Ada Tasking in the Building of Satellite Control Software*, ftp://ftp.estec.esa.nl/pub/ws/wsd/hrt/

RD17 W. Codenie *et al* (1997), *From Custom Applications to Domain-Specific Frameworks*, Communications of the ACM, Oct. 1997, Vol. 40, N. 10, page 71-7.

RD18 M. Awad, J. Kuusela, J. Ziegler (1996), *Object-Oriented Technology for Real-Time Systems*, Prentice Hall PTR, Upper Saddle River, NJ 07458

RD19 A. Pasetti (2001), *Software Frameworks and Embedded Control Systems*, LNCS Series, Springer-Verlag, 2002

RD20 A. Pasetti (2000), *System Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/021

RD21 A. Pasetti (2000), *Inter-Component Communication Framelet*, AOCS Framework Document ref. SWE/99/AOCS/005

RD22 A. Pasetti (2000), *Object Monitoring Framelet*, AOCS Framework Document ref. SWE/99/AOCS/008

RD23 A. Pasetti (2000), *Sequential Data Processing Framelet*, AOCS Framework Document ref. SWE/99/AOCS/006

RD24 A. Pasetti (2000), *AOCS Unit Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/017

RD25 A. Pasetti (2000), *Reconfiguration Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/015

RD26 A. Pasetti (2000), *Operational Mode Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/009

RD27 A. Pasetti, T. Brown (2000), *Manoeuvre Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/012

RD28 A. Pasetti (2000), *Failure Detection Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/010

RD29 Deleted

RD30 A. Pasetti (2000), *Failure Recovery Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/011

RD31 A. Pasetti (2000), *Telemetry Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/003

RD32 A. Pasetti (2000), *Telecommand Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/014

RD33 T. Brown, A. Pasetti (2000), *Controller Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/016

RD34 J. Barnes, *Ada 95 rationale : the language, the standard libraries*, Heidelberg: Springer, 1997

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
MACS	Modular Attitude Control System
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 is written as part of the project *Design and Prototyping of a Software Framework for the AOCS* done for ESA-Estec under contract Estec / 13776 / NL / MV.

The objective of the project is to design a software framework for the AOCS software. Software frameworks are a form of software reuse that promotes the reuse of entire architectures optimized for a narrowly defined application domain.

The application domain targeted by the AOCS framework is described in RD2. Most recent ESA missions with a dedicated AOCS processor would fall within the framework domain.

The main features of the AOCS framework are:

- *Architectural reusability*

  At present, most AOCS systems are re-designed from scratch for each new mission. In some cases, code is re-used across missions (generally only if the same contractor is responsible for software development in different missions) but architecture – often the toughest part to design – cannot be ported at all.

  The AOCS framework encapsulates a generic architecture for an AOCS system allowing re-use of architecture or architectural features across missions.

- *Component reusability*

  AOCS software re-usability is currently possible at most at the subroutine level and even this is usually impossible because small functionality changes need code changes and hence complete re-testing of the subroutine.

  The AOCS framework is made up of components that can be re-used as binary units without changes while allowing functionality tuning through inheritance or object composition.

- *Architectural extensibility*

  At present, it is possible to extend the functionality of an AOCS system by modifying the algorithms coded in its subroutines. Changes with architectural impacts (eg adding a new mode or a new telecommand) are however much harder and more expensive to implement.

  While freezing certain architectural decisions, the AOCS framework leaves other explicitly open for modification.

- *Compatibility with simulation environment*

  Simulation of an AOCS software currently implies either the development of code that simulates the AOCS software or the use of (software or hardware) emulators for the AOCS processor.

  The AOCS framework allows an AOCS system to be built as a collection of components that can be used without change either in the AOCS system itself or in an off-line simulator during testing.

- *Compatibility with autocode tools*

  The AOCS framework allows embedding of code generated by th Xmath autocode tool.

- *Object-oriented design*

  Current AOCS software systems follow the modular paradigm.

  The AOCS framework conforms to object-oriented principles which are regarded as the soundest paradigm in software design available at the moment.

3.1 Purpose of this Document

Using the terminology of RD19, the present document covers the framework concept definition. More specifically, its objectives are:

- *To define the general design principles for the AOCS framework*

  The general design principles are the overall constraints on the framework design. They define the policy to be adopted in regard of general issues like: use of multiple inheritance, use of dynamic memory and task allocation, use of non-standard language constructs, use of templates, specialization mechanisms, general class structure, allowed implementation languages, use of component standards, etc.

- *To define the high-level abstractions in the framework domain*

  The domain abstractions capture the commonalities of the AOCS applications in the framework domain. They describe concepts and functional features that span application boundaries and are found in all domain applications.

- *To identify the main framework hot-spots*

  A framework is a tool to help generate concrete applications within its domain. The hot-spots are the points where the framework is customized during the application instantiation process to adapt it to the needs of a specific concrete application.

- *To identify the main framework design patterns*

  The design patterns are one of the basic constructs offered by a framework to application developers. Design patterns provide architectural solutions to the adaptation problems of the framework.

- *To identify the framelets*

  In order to simplify its design, the framework was subdivided into framelets. Framelets are self-standing units of design that address specific design problems within the framework.

The difference between *identification* of a feature and *definition* should be noted. Identification requires simply that the presence and need for the feature be recognized and that its function be informally and perhaps incompletely described. Definition instead implies full formal description of the feature. Features that are only identified in this document are defined in the lower level documents describing the framelet architectures (see RD20 to RD33).

3.2 Design Methodology

The design and development of the AOCS framework did not follow any formal methodology. This was partly because of manpower and schedule constraints, and partly because of the lack of mature methodological tools targeted at framework (as opposed to application) development. Use was, however, made of many of the methodological concepts described in RD19.

3.3 Boundary Conditions

Broadly speaking, the ultimate objective of the AOCS framework project is to improve the re-usability – and hence lower the cost and enhance the reliability – of AOCS software. This objective is achieved by exploiting recent advances in software engineering (the transition to the OO paradigm) and in microprocessor technology (the space-qualification of the ERC32 SPARC processor).

In general, technological advances could be used either to expand the functionality of a piece of software or to improve its quality[1]. The intention of this project was to concentrate on the

[1] Historically, the emphasis has been on the former task. This is why successive revolutions in software engineering – the transition to compiled languages, then to procedural languages, and still later to modular languages – have not had a major impact on software quality (programs are as error-prone and as poorly readable / reusable today as they were in the sixties!).

latter task. Hence, the framework to be designed in the present study cover only the present functionalities of AOCS systems. By constraining functionality to its present level, technological advances are leveraged to improve quality and reduce development costs.

3.4 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. However, since it deals with the framework at design level it remains largely applicable for the Java version. In particular, the description of the framework design principles and framework design patterns that forms the core of this document is entirely language-independent.

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: http://www.aut.ee.ethz.ch/~pasetti/RealTimeJavaFramework/index.html.

3.5 Notation

Architectural solutions are presented using UML notation. The diagrams included in this document were obtained with the *Together* tool (version 4.0). Note that this is only a concept-level document and therefore the class and object diagrams may be incomplete as their purpose is merely illustrative.

Where appropriate, implementation outlines are given in this document using pseudo-code. The notation and pseudo-code examples are not always consistent. Their purpose is not to represent a design but to convey ideas and concepts that will then be more formally developed in an architectural design document.

The following naming conventions are adopted in the pseudo-code examples:

- Type, class and interface names begin with a capital (as in '`Runnable`' which denotes an interface)
- Variable names begin with a small letter (as in: '`modeManager`' which denotes the name of an object.)
- Variable names are sometimes derived from their class name by prefixing 'a' or 'the' (as in: '`theModeManager`' which denotes an object of class '`ModeManager`' or '`aAocsController`' which denotes an object of class '`AocsController`').

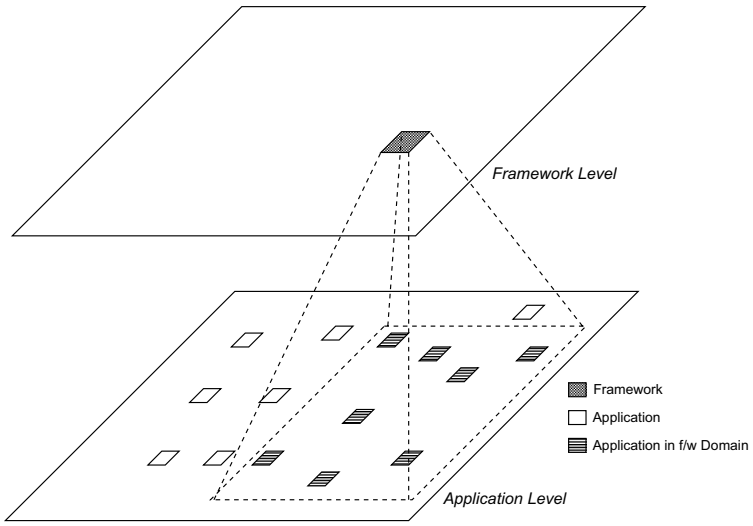
3.6 Appendices

Sections 29 to 31 are reference appendices that give a list of the framework abstractions, the framework design patterns and the framework hot-spots. Readers may want to refer to them as an aid to their understanding of this document.

4 SOFTWARE FRAMEWORKS

Software frameworks are a form of software reuse that primarily promotes the reuse of entire architectures within a narrowly defined application domain. They propose an architecture that is optimized for all applications within this domain and make its reuse across applications in the domain possible. See RD3 for a general introduction to software frameworks.

Experienced software engineers who develop several related applications reuse architectures as a matter of course. Software frameworks are intended to formalize and make explicit this form of architectural reuse. They allow the investment that is made into designing the architecture of an application to be made available across projects and across design teams.



Framework and Application Levels

A framework therefore serves as the basis for the generation of a whole family of applications (see previous figure). The process whereby a concrete application is derived from the framework is called *framework instantiation*. The two key problems in framework design are to identify the points at which the framework behaviour is adapted to the requirements of a specific applications, the so-called *hot-spots*, and to devise ways to implement this adaptation. In this paper, frameworks are assumed to be object-oriented and the adaptation mechanisms are therefore based on abstract coupling (run-time adaptation) and inheritance (static adaptation).

The term “framework”, like so many others in computer science, is heavily overloaded and although most experts will agree with the definitions given above of what frameworks are for, there is as yet no consensus as to what exactly constitutes a framework. In the AOCS project, a framework was defined as consisting of:

- abstract interfaces
- design patterns
- concrete components

These are the constructs that the framework offers to application developers and that the latter use to build a particular application in the framework domain.

The *abstract interfaces* are declarations of sets of related operations for which no implementation is provided. Abstract interfaces capture behavioural signatures that are common to all applications in the framework domain. In a language like C++, they are implemented as pure virtual classes.

The *design patterns* are optimized solutions to recurring architectural problems in the framework domain (see RD1). Since they encapsulate reusable architectural solutions, and since frameworks are intended as vehicles for architectural reuse, design patterns are normally the heart of a framework.

Additionally, design patterns promote architectural uniformity by ensuring that similar problems in different parts of the same application receive similar solutions. This endows the application architecture with a single “look & feel” that makes using and expanding it considerably easier. Design uniformity is an important aspect of architectural reusability and a second important contribution of design patterns to frameworks.

The *concrete components* are binary units of reuse that implement one or more interfaces and that can be configured for use in a specific application at run-time.

Framework components can be of two types: core components and default components. *Core components* encapsulate behaviours that are common to all applications in the framework domain. They are therefore used by all applications instantiated from the framework.

Default components represent default implementations for some of the abstract interfaces in the framework. They encapsulate behaviours that are found in many applications in the framework domain but that are not intrinsic to the framework domain.

A framework will in general provide default components to implement the most common behaviours found in its domain. When very specific behaviours are required that are not catered for by the default components, application-specific components need to be created.

These will have to conform to the framework interfaces and may often be obtained by specializing the default components either through inheritance.

5 GENERAL STRUCTURE OF THE AOCS FRAMEWORK

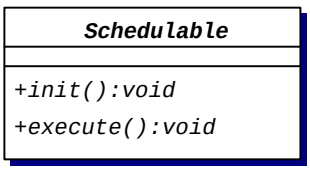
The primary objective of the AOCS framework project was to define a framework architecture for the AOCS domain. The resulting architecture is described in the remainder of this document. This section concentrates on presenting the general structure of the framework. In particular, an analogy with Real Time Operating Systems (RTOS) is developed that is interesting both because it provided the inspiration for the AOCS framework and, from a more theoretical point of view, because it points to an interesting parallel between frameworks and operating systems.

5.1 The RTOS Example

One of the challenges of the AOCS framework project lay in the real-time and embedded character of this type of applications which have seldom been the targets of frameworks. In the initial phase of the project, however, it was realized that there is at least one highly successful example of frameworks for real-time systems. As will be shown in a moment, RTOS's fully fit the definition of framework given in section 4 and, because of their record in providing reusable software solutions, it was felt that they could serve as conceptual models for the AOCS framework. Accordingly, the AOCS framework project started with an attempt to answer the question: "How does an RTOS achieve reusability?". Three complementary answers were identified. They are illustrated below for the case of task scheduling which is a typical functionality offered by RTOS's.

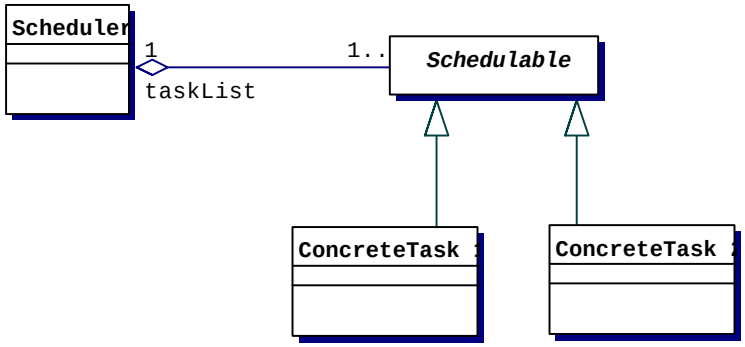
Firstly, an RTOS enforces an *architectural infrastructure.* Namely, it assumes an application to be made up of tasks with certain well-defined characteristics (single entry point, single thread of execution, etc). Applications that wish to use the RTOS must conform to this architecture. Using the terminology of section 4, the RTOS proposes a design pattern.

Secondly, the RTOS relies on the *separation of the management of a functionality from its implementation* and achieves this separation through an abstract interface. In the case of task scheduling, for instance, and using UML notation, the RTOS sees a task as a component implementing the following abstract class:



where a call to `init()` causes the task to initialize itself and a call to `run()` causes it to start executing. Separation through an abstract interface fosters reusability because it decouples the implementation of the task, which is necessarily application-specific, from the scheduling policy which is used to activate it and which is application-independent.

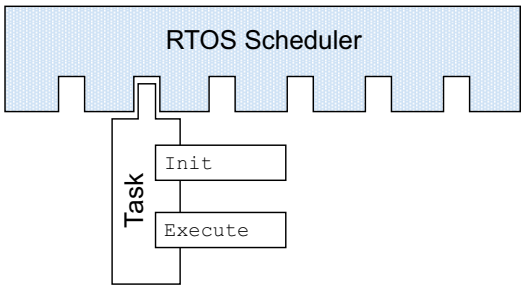
Thirdly, the RTOS provides an application-independent and hence reusable component that encapsulates the management of the RTOS functionalities. The RTOS executable, for instance, will normally include a scheduler component that can be made application-independent because it sees the tasks it manages only through the abstract `schedulable` interface:



In the terminology of section 4, the scheduler is a core component.

Thus, the RTOS offers all three frameworks constructs identified in section 4and qualifies as a framework because the design pattern and the abstract interfaces are introduced to model adaptability and the component packages the invariant part of RTOS-based applications. It may be noted in passing that, conceptually, the primary entities are the design pattern and the schedulable abstract interface. The scheduler component is secondary as its function is essentially defined once the design pattern and abstract interface have been defined.

The next figure gives an alternative view of an RTOS-based system. The RTOS module is shown as a component exposing slots where application-dependent components can be plugged. The application-dependent components represent the tasks to be managed by the RTOS. The slots in the RTOS are therefore the hot-spots where the RTOS customisation takes place. The “shape” of the slots that determines whether a certain component can be plugged in is defined by the abstract interfaces that separate task management from task implementation.



Plug-In View of an RTOS Scheduler

5.2 The Lesson for the AOCS

One conclusion of the analysis in the previous section is that in an AOCS there is at least one functionality – task scheduling – where a framework approach can improve reusability. Task scheduling is of course only one of many functionalities implemented by an AOCS and it is therefore natural to ask whether the same principles that make scheduling management reusable could be applied to the other AOCS functionalities.

A large part of the design of the AOCS framework can be seen as an attempt to provide a positive answer to this question. This was done by first isolating the functionalities present in an AOCS and then systematically applying to each the reusability model of the RTOS.

The functionalities that were identified in an AOCS are:

- The *Telemetry Functionality* covering the formatting and despatching of housekeeping information to the ground station.
- The *Telecommand Functionality* covering the processing and execution of commands received from the ground station.
- The *Failure Detection Functionality* covering the implementation of checks to autonomously detect failures in the AOCS software.
- The *Failure Recovery Functionality* covering the implementation of the corrective measures to counteract the failures reported by the failure detection functionality.
- The *Controller Functionality* covering the management and implementation of the control algorithms for the control loops operated by the AOCS.
- The *Reset Functionality* covering the control of the reset functions that bring the AOCS components to some initial default state.

- The *Configuration Functionality* covering the control of the configuration functions to clear all configuration information in the AOCS components, and to check that all components are correctly configured and ready to start normal operation.
- The *Unit Functionality* covering the acquisition of data from and the forwarding of commands to the external units managed by the AOCS (the sensors and the actuators).

From the point of view of the framework architecture designer, these functionalities can be treated as independent of each other. This is obvious in all cases with the possible exception of the failure detection and failure recovery functionalities that in some systems are merged together. In the AOCS framework, however, the failure detection manager constructs failure events encapsulating the description of any failures it has encountered and deposits them in shared repositories. In a separate activity, the failure recovery manager inspects the failure event repository and processes the failures according to their description.

The mutual independence of the functionalities was crucial in simplifying the framework design as it allowed architectural solutions to be developed in isolation for each functionality. Architectural independence, however, did not degenerate into arbitrariness of solutions. As argued in section 5.4, design unity was maintained by imposing the same meta-pattern on all functionality management problems.

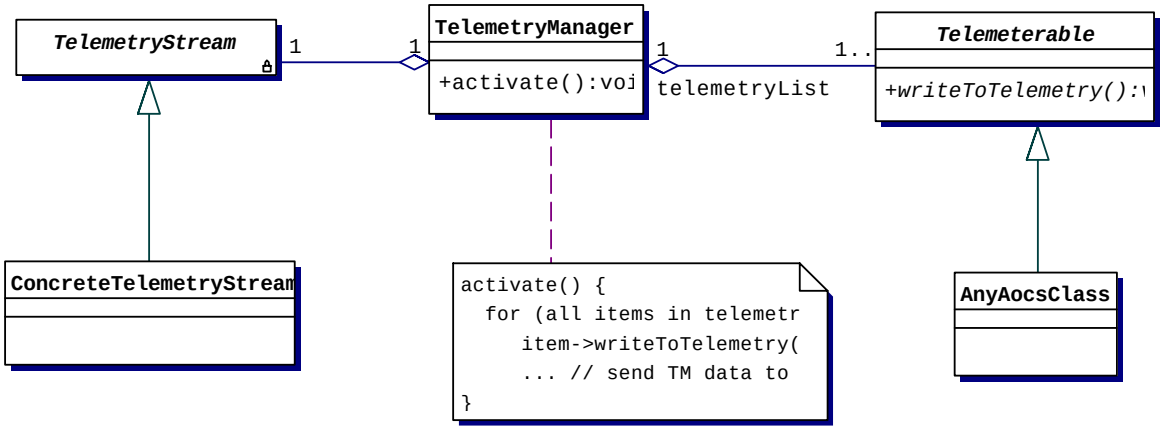
The influence of the RTOS example on the AOCS framework design will now be illustrated by retracing the steps that led to the definition of an architectural solution for the telemetry management problem. This description anticipates that of section 27 documenting the telemetry framelet.

5.3 The Telemetry Functionality in the AOCS Framework

In current AOCS systems, telemetry processing is controlled by a so-called telemetry handler that directly collects telemetry data, formats and stores them in a dedicated buffer, and then has them transferred to the ground. To accomplish its task, the telemetry handler needs an intimate knowledge of the type and format of the telemetry data: it has to know which data, in which format, and from which objects have to be collected. It is this coupling between telemetry handler and telemetry data that makes the former application-specific and hinders its re-use.

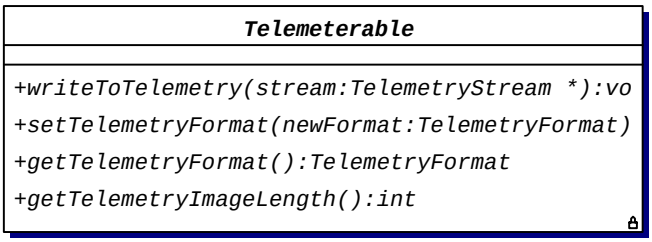
In keeping with the RTOS reuse model, the approach taken in the AOCS project is based on a design pattern – the *telemetry design pattern* – that calls for a separation of telemetry management from the implementation of telemetry data collection. This is achieved by endowing selected components in the AOCS software with the capacity of writing themselves

to the telemetry stream. Telemetry processing is then seen as the forwarding to the ground of an image of the internal state of some of the AOCS components. The resulting architecture is:



The ability of a component to write its own state to the telemetry stream is encapsulated in the abstract interface `Telemeterable` which must be implemented by all components intended for inclusion in telemetry. Its basic method is `writeToTelemetry`. Calling it causes a component to write its internal state to the telemetry stream. The telemetry manager is responsible for keeping track of the components whose state should be sent to the telemetry stream and for calling their `writeToTelemetry` methods to start the forwarding of telemetry data to the ground. The telemetry manager also needs to be aware of the data stream to which the telemetry data should be written. Since the hardware characteristics of the channel over which telemetry data are transmitted to the ground differs across AOCS systems, the telemetry stream is identified by an abstract interface, `TelemetryStream`. This interface defines the generic operations that can be performed on a data sink representing an ideal telemetry channel.

The second step in the development of an architectural solution for AOCS telemetry management was the instantiation of the telemetry pattern for the AOCS domain. This chiefly involved providing exact definitions for the `Telemeterable` and `TelemetryStream` interfaces. The former, for instance, was defined as follows:



In the selected implementation, the telemetry stream is passed as an argument to method `writeToTelemetry` and methods are provided to check the size and set the format of the telemetry image generated by each component.

In the third and final step of the architecture design process, the telemetry manager component was characterized. The semantics of the telemetry pattern and of its associated abstract interfaces make its definition straightforward. The core of the telemetry manager component can be represented in pseudo-code as follows:

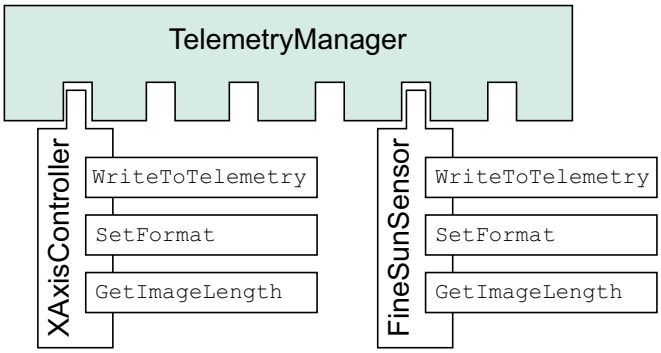
```
Telemeterable*     tmList[N_TM_OBJECTS];
TmStream*          tmStream;
. . .
void activate()         {
  for (all TM objects ‘t’ in tmList)
  {  t->setTelemetryFormat(tmFormat);
         tmDataSize=t->getTelemetryImageLength();
         if (object image fits in TM buffer)
              t->writeToTelemetry(tmStream);
         else
              . . .         // error!
  }
}
```

Thus, the telemetry manager maintains a list of telemeterable components and, when it is activated, it calls their `writeToTelemetry` method. It uses the other operations exposed by `Telemeterable` to control the format of the telemetry images and to verify that the size of their image is consistent with the capacity of the telemetry channel. Additionally, and not shown in the pseudo code segment, the telemetry manager will offer operations to dynamically load and unload items from its internal list of telemeterable components and to set the telemetry stream at initialisation. Clearly, all these operations are application-independent and hence the telemetry manager is fully reusable or, in other words, it becomes a core component in the framework. The AOCS framework also provides a default

component implementing the `TelemetryStream` interface for the case of a DMA-based telemetry channels that happens to be very common in AOCS systems.

The architectural solution to the telemetry management problem therefore exhibits the typical features of a framework being based on a design pattern that is specialized by abstract interfaces and a core component (the telemetry managers) and complemented by a default component (the default implementation of interface `TelemetryStream`).

Figure 4 shows a plug-in view of telemetry management. It again draws attention to the similarity between the telemetry management and the RTOS scheduling architectures and on how in both cases reusability originates in the introduction of an abstract interface to separate the management of a functionality from its implementation.



Plug-In View of a Telemetry Manager

5.4 The Manager Meta-Pattern

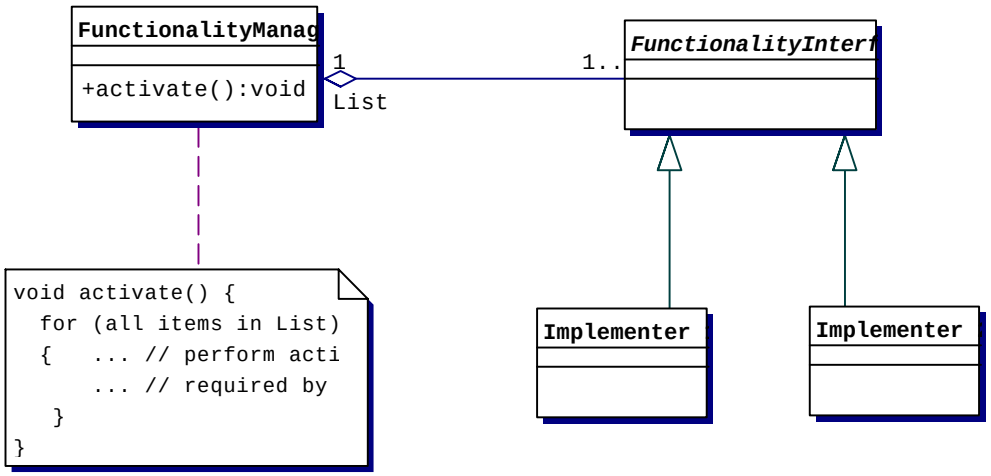
The procedure described above for the telemetry functionality was followed for all other functionalities identified in the AOCS. In all cases, architectural solutions were produced that were conceptually similar to that for the telemetry functionality and that were inspired by the RTOS model of achieve reusability. This commonality of design solution is so significant that it deserves to be captured in a new concept: the *manager meta-pattern*.

Design patterns define solutions for design problems at application level: they prescribe recipes that allow a class of related design problems confronting application developers to be solved in a uniform manner. The term meta-pattern is instead proposed to describe commonalities in different design patterns used to address different design problems in the same framework. Just as design patterns promote consistency at application level by ensuring that similar problems arising in different locations receive similar solutions, meta-patterns support design consistency at the framework level.

The manager meta-pattern addresses the problem of managing a functionality that requires the same actions to be repeatedly performed on a class of objects providing different implementations of the functionality itself. The solution it proposes can be summarized as follows:

- Define an abstract interface capturing the behavioural signature of the functionality. This interface separates the management of the functionality from its implementation.
- Build an application-independent and reusable functionality manager component (a core component). The functionality manager maintains a list of objects seen as instances of the functionality interface and it exposes an activation method that systematically performs all the actions required by the functionality management on all objects in the list
- Where appropriate, provide default implementations of the functionality interface (default components)

The class diagram for the manager meta-pattern is:



The manager meta-pattern was applied to all the AOCS functionalities identified above. This resulted in a framework architecture that is both elegant and homogeneous. Extensions and upgrades are facilitated by the mutual independence of the functionality managers that makes addition of facilities to handle new functionalities easy.

5.5 Overall Structure

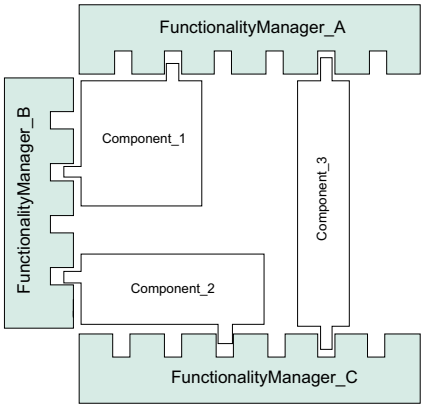
Systematic application of the manager meta-pattern leads to a framework that will generate AOCS applications with an architecture as shown in the next figure. A collection of functionality managers represent the reusable architectural skeleton of the application. This is

customized for a specific application by composition with components encapsulating the application-dependent functionalities implementations. To each functionality manager an abstract interface is associated.

The RTOS could be one of the functionality managers and the framework can therefore be seen as offering a *domain-specific extension to the operating system*.

It should be stressed that the same component can be used to customize several functionality managers. A closed-loop controller, for instance, would obviously be plugged into the controller manager but might also be plugged into the telemetry manager if it is desired to include its state in the telemetry stream. This situation is handled through multiple inheritance with a component implementing all the interfaces associated to the functionality managers it customizes. Component_1 in the figure, for instance, would have to implement (at least) the abstract interfaces corresponding to the functionality managers A and B.

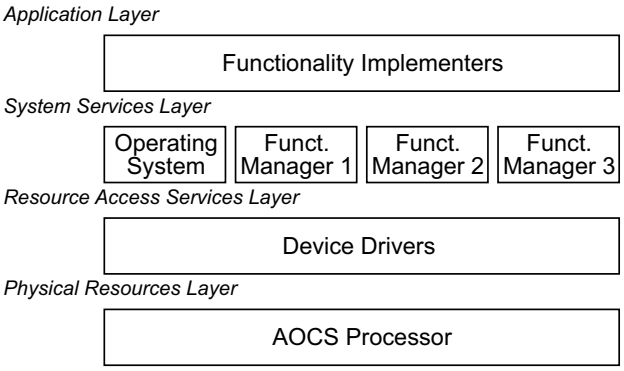
Use of multiple inheritance is often resisted because it can lead to ambiguities but the AOCS framework only requires multiple inheritance of abstract interfaces which is known to be safe. This kind of multiple inheritance is used extensively throughout the AOCS framework with typical AOCS components normally implementing 5 or 6 different abstract interfaces each representing a different functionality.



Structure of Framework-Generated AOCS

The next figure shows how the application architecture implied by the AOCS framework can be recast according to the Generic Open Architecture (GOA) model (see RD4) . GOA is a reference architecture for embedded software to provide a foundation for effective open systems. It distinguishes four layers in a software system: application software, system services, resource access services, and physical resources. In a GOA perspective, an AOCS application is most naturally decomposed by placing the functionality managers in the

system services layer alongside the operating system. This choice is dictated by the application-invariant character of the functionality managers and by the fact that they – unlike the components implementing the functionalities – need access to the hardware drivers in the resource access layer. The telemetry manager for instance has an interface to the hardware channel through which telemetry data are routed to the ground.



GOA Model of the AOCS Architecture

The previous figure illustrates again the affinity of functionality managers and operating systems and shows that the framework factors out of an application OS-like functionalities encapsulating them into application-independent modules.

5.6 Architectural Infrastructure

The functionality managers address localized design issues in the AOCS domain but do not exhaust the framework since they cannot handle non-localized infrastructure problems.

One first example of infrastructural problem concerns inter-component communication. The architecture promoted by the AOCS framework is component-based. Some mechanism is required to allow these components to exchange information.

A second example of infrastructure problem arises from the need of components to monitor each other both in order to synchronize their behaviour and for purposes of failure detection.

Other infrastructural problems in AOCS applications concern the handling of unit reconfiguration, the handling of sequential data processing chains, and the implementation of mode-dependent behaviour.

As will be seen in the remainder of this document, the AOCS framework provides solutions to all of the above infrastructure problems. These solutions are qualitatively different from

those proposed for the AOCS functionalities but they resemble them in being based on the definition of domain-specific design patterns and of the abstract interfaces required to specialize them. In many cases, core and default components were also developed but in the framework design process they always came second to the design patterns and abstract interfaces.

6 GENERAL DESIGN PRINCIPLES

This section describes the general principles adopted in the design of the AOCS framework. The general design principles are the overall constraints on the framework design which define the policy to be adopted in regard of general design issues that affect the entire framework.

6.1 An Object-Oriented Framework

Many design methodologies carrying the label "object-oriented" (eg. HOOD) are more properly described as "object-based". Such methodologies aim to build a software system as a collection of hierarchical objects that encapsulate data while exposing operations to handle them. The design process they prescribe typically starts with an analysis of the problem domain to identify *concrete* classes and *concrete* objects. They often result in designs that are hierarchical in the sense that objects and classes are nested within each other.

The truly object-oriented design process is radically different as it puts the emphasis not on objects but on abstract interfaces. An abstract interface is a set of operations with no or incomplete implementation (abstract interfaces are represented in C++ by pure virtual classes). In this kind of object-oriented approach, the design begins by identifying abstract functionalities that are then encapsulated in abstract interfaces. These interfaces will only include high-level behaviour and will leave most of the functionality implementation open. Objects are derived at a later stage of the design process as instances of classes that implement one or more abstract interfaces. Object-oriented architectures are hierarchical in the sense that they are based on hierarchical class trees with abstract classes at the top of the tree and concrete classes at the bottom.

The AOCS framework is based on true object-orientation. It adds a further dimension – specific to frameworks – to the typical object-oriented design process because it places additional emphasis on identifying points of behaviour adaptation where the framework can be tailored to the need of a particular application. At the architectural level, adaptation is based on the use of design patterns. At the implementation level, it is based on abstract coupling and dynamic binding.

See RD19 for an example of the difference between object-based and object oriented design.

6.2 A Component-Based Framework

The framework is intended to be component-based. By this, it is meant that all the modules defined in the framework should be re-usable *without changes to the source code*. Behaviour adaptation is to be achieved entirely through inheritance and / or object composition.

At this stage, the term *component* is used to designate a binary unit of reuse that implements one or more framework interfaces and that can be configured for use in a specific mission at run-time. At a later stage – and in a different project – the option will be studied of implementing the framework upon a component infrastructure (eg DCOM or CORBA) that allows inter-operability across language, compiler and processor barriers while remaining compatible with the real-time and criticality requirements of the AOCS software.

In order to understand the benefits of implementing the AOCS framework upon a component infrastructure consider again the example of telemetry management presented in section 5.3. The proposed solution is built on a telemetry manager component that, when it is activated, calls the operations defined by interface `Telemeterable` on the objects whose state should be included in telemetry.

Telemetry management is a function that is usually performed by every subsystem on a satellite. If the subsystems are physically located on different processors, or even only in different partitions on the same processor, then each subsystem will require its own telemetry manager component. This is clearly wasteful. A more efficient architecture has a single telemetry manager that is able to see all its telemetry objects as if they resided in the same address space. A component infrastructure provides precisely the mechanisms to make this possible.

6.3 Use of Design Patterns

As explained in section 4, design patterns are one the fundamental constructs of software frameworks. Wherever possible, the AOCS frameworks makes use of the design patterns of RD1 which are taken as model of good object oriented-design. Where these standard patterns are not adequate, AOCS-specific patterns are developed.

6.4 Component Behaviour Adaptation

The derivation of an AOCS software from the framework requires the adaptation of its behaviour to the specific needs of the AOCS software. The adaptation of the behaviour of a component is generally based on dynamic binding as achieved either through *inheritance* or through *object composition*. The former mechanism is used for static behaviour adaptation, the

latter for dynamic behaviour adaptation. In the present design, there no effort is made to favour one mechanism over the other.

6.5 Delegation of Responsibility

If a certain task has to be performed by or upon an object, the responsibility for performing the task is as far as possible delegated to the object itself. Thus, for example, rather than having a procedure that writes the state of an object to telemetry, the object itself will be endowed with the ability to write itself to telemetry (ie. it will be given a method that, when called by an external telemetry manager, causes the object to write its state to the telemetry stream).

Delegation of responsibility to objects results in a “bottom heavy” implementation where many functionalities are implemented in the objects at the bottom of the object hierarchy. This approach ensures that the higher level objects that embody the AOCS structure are mission-independent and can be reused without changes.

6.6 Avoidance of Multiple Implementation Inheritance

As discussed in section 5.5, the selected framework architecture needs multiple inheritance. On the other hand, it is well known that multiple implementation inheritance presents semantic problems that advise against its use in critical applications (see RD9) and that its over-use can sometimes make component re-use difficult (see RD17). No such problems,

however, attach to multiple inheritance of interfaces (in the Java sense of the word). Hence the solution adopted for this project is as follows:

- only single inheritance of implementation is allowed;
- multiple inheritance of interfaces is allowed.

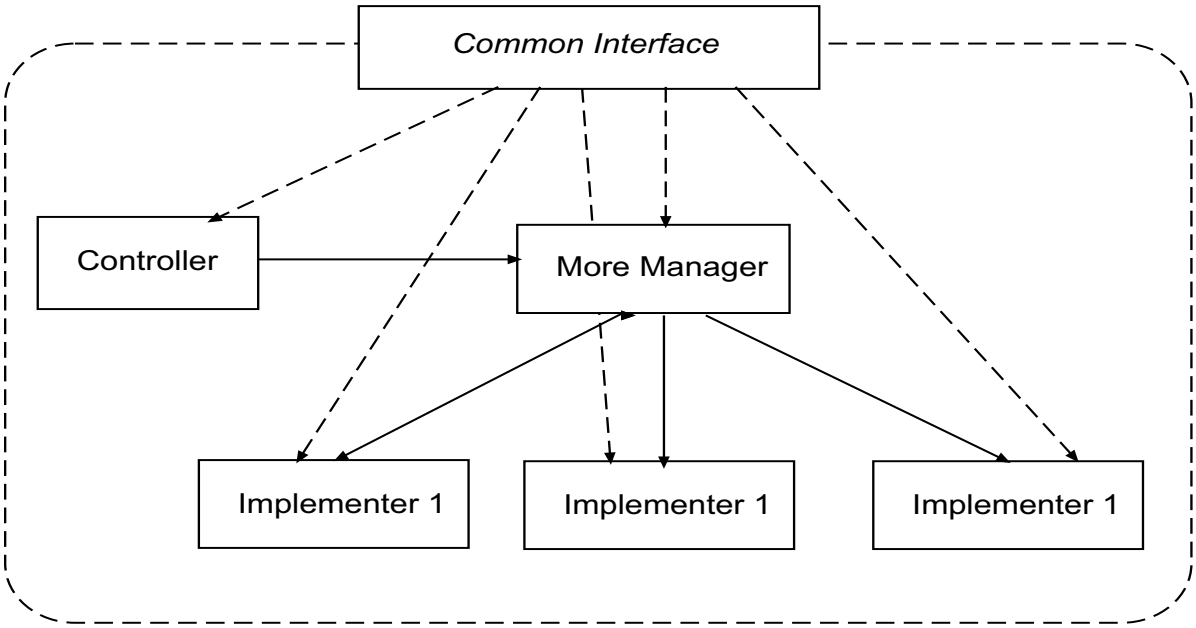
This solution is borrowed from the Java language and it is both safe and well-tested.

6.7 External Interfaces

External interfaces – towards external devices and towards the processor – are encapsulated in objects that, within the AOCS software, act as proxies for the external objects. This is an instance of the *proxy* pattern from RD1.

6.8 Façade Encapsulation for Components

The framework design often results in components that are made up of several smaller components all of whom need to interact with external components. Consider for instance an attitude controller component. Its structure can be represented as follows:



The boxes within the dashed containers are the components required to implement controller management (the controller manaer, its mode manager and several controller implementers).

External entities wishing to interact with the controller functionality need to interact with the individual components inside it. Thus, for instance, if they wish to add or remove an implementer, they need to act on `ModeManager` whereas if they need to change the mode manager itself, they need to act on `Controller`. This makes interaction with the controller functionality difficult because it forces its partners to hold references to all its internal objects.

The *façade design pattern* from RD1 offers an alternative solution. With this design pattern, a wrapper component (dashed box in the figure) is used that encloses all components involved in controller management. The wrapper component is provided with an interface that gathers together all operations that internal components need to expose to the outside world. The implementation of these operations simply consists of a delegation to the corresponding operation of an enclosed object.

Use of a façade wrapper simplifies interactions among components although it adds a new level of indirection. Façade wrappers are not used in the present – prototype – version of the framework but may be added in future versions.

6.9 Basic Classes

A *basic class* is a class that serves as a base class to a large number of framework classes. Basic classes are introduced in the AOCS framework to allow uniform treatment of large numbers of objects.

Two basic classes are present in the AOCS framework: `RootObject` and `AocsObject`. Class `RootObject` is used as the base for all concrete (non-interface) classes in the AOCS framework and in the applications instantiated from it. This class defines two read-only attributes: the *instance and class identifiers*. Its interface is defined as follows:

```
class RootObject {
     IdType getInstanceIdentifier();
     IdType getClassIdentifier();
}
```

Class `RootObject` ensures that all objects and all classes in the framework and in AOCS applications can be uniquely identified.

The `AocsObject` class is introduced as a base class for all non-trivial objects in the AOCS framework and in the AOCS applications. Its purpose is to gather together and provide to its children classes some basic services that are likely to be required by all but the simplest

objects in the framework. Objects that are directly or indirectly instantiated from `AocsObjects` are called *AOCS Objects.*

Class `AocsObject` makes the following services available to its children through protected methods:

- Time recovery
- Failure reporting
- Configuration error reporting

Additionally, AOCS objects have the following properties that they inherit from abstract interfaces implemented by class `AocsObject`:

- Telemeterability (AOCS objects can be included in telemetry)
- Resettability (AOCS objects are capable of resetting their internal state)
- Configurabiity (AOCS objects are capable of checking that they are correctly configured)

AOCS objects in an application are intended to be created during initialization and never to be destroyed. To ensure that this is the case, the `AocsObject` destructor generates an error if it is called. Note that this means that AOCS objects cannot be passed as method parameters. Components must operate upon and exchange objects exclusively through pointers.

Since AOCS objects are created statically and never destroyed, it is safe to use pointers to them. A design rule in the AOCS framework dictates that pointers can only be used on objects of type `AocsObject`. Adherence to this rule could in principle be checked by automatic inspection of the framework source code.

6.10 Time Management

The AOCS framework assumes that two kinds of timing information are available: a clock time measurement from an *AOCS clock* object and a cycle number that indicates the number of AOCS cycles elapsed from some starting cycle.

An AOCS clock is an object implementing the following interface:

```
interface AocsClock {
     AocsTime getTime();     // get clock time
     int getCycle();         // get cycle number
     void synchronizeWithSystemTime();
}
```

The AOCS framework assumes that all applications will make available one AOCS clock object. AOCS object have access to timing information via the services offered by class `AocsObject`.

6.11 Language Selection

The design presented here is intended to be language independent. The assumption made during the architectural design phase is that it should be powerful enough to support full object-orientation. This in particular implies support for the safe version of multiple inheritance used in the framework design.

The framework prototype will be implemented in C++ but other language solutions are possible.

Ada95 is still the recommended language for ESA-initiated space projects and ensuring compatibility with it would be correspondingly desirable. However, Ada95 does not directly support multiple inheritance. The effects of multiple inheritance can be achieved using the *multiple view* mechanism of RD34 but this is very awkward to use as it essentially represents a manual implementation of a construct that in languages like C++ and Java is automatically coded by the compiler. It may be suitable where usage of multiple inheritance is incidental but not in a system like the AOCS framework where it plays a fundamental role.

There is one further point that deserves to be made on the contentious issue of language selection. The AOCS framework – like much modern design practice (see RD9) – is moving towards component-based architectures. Components are reused as binary units and, ideally, should offer language interoperability. If this vision is realized, then design attention will shift away from the internal architecture of the component and towards the interface that it exposes. At that point, language issues will become much less relevant at system level. This situation already exists today for operating systems – arguably the only case of commercially successful components – which are normally bought as binary entities and where the issue of the language in which they were originally written is of no interest.

6.12 Execution Time Predictability

AOCS systems must be *demonstrably schedulable*. An application is schedulable if it can meet all its deadlines under any operational conditions. The schedulability requirement derives from the hard real-time nature of the AOCS. Additionally, since AOCS systems are mission-critical, schedulability must be demonstrable in the sense that it must be possible to verify it statically by analysing the source code.

A necessary condition for the demonstrable schedulability of a software application is that the execution time of any of its code segments be statically predictable. This requirement has a repercussion at framework level insofar as some of the code in an AOCS application is inherited from the framework (core and default components).

The AOCS framework safeguards timing predictability primarily by avoiding constructs and operations that lead to non-predictable code. In practice, this led to the avoidance of exception handling and dynamic memory allocation. The former presented no particular problem as the framework infrastructure offers a mechanism for reporting events that can be used *in lieu* of exceptions. The latter was instead rather difficult to achieve. Most existing frameworks are targeted at non-real-time applications and make extensive use of dynamic object creation and destruction. Avoiding such operations required some rethinking of typical design patterns used in frameworks and sometimes resulted in slightly awkward constructs. On this point, see also the next sub-section.

As already mentioned, the AOCS framework relies extensively on dynamic binding to model application adaptability. At first sight dynamic binding might seem to be incompatible with code execution predictability because of the impossibility of statically associating a definite piece of code to a particular method call. However, in embedded systems there is no dynamic class loading and hence the number of methods that *might* be called is finite (and often small). It is therefore always possible to determine worst-case execution times and use this estimate in the schedulability analysis.

The heavy reliance of the framework on design patterns poses a more serious problem since some design patterns have a recursive structure that makes static timing analysis impossible. In such cases, timing predictability can only be retained by adding meta-information to the source code that specifies the maximum depth of the recursion. In order to make it easy to provide this information, this and lower level documents clearly identify all recursive design patterns they introduce and explain how upper bounds on the depth of recursion can be inferred from their semantics.

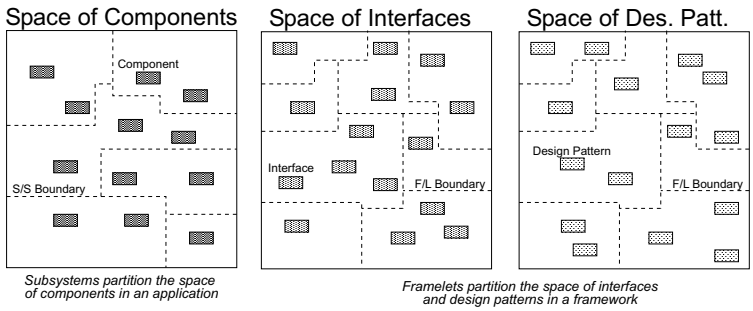
7 FRAMELET OVERVIEW

The AOCS framework is built as a collection of framelets. Framelets are introduced as subdivisions of a framework to simplify its design. They address the quantitative aspect of framework complexity that stems from the sheer size of frameworks: a single framework may encompass hundreds of constructs – classes, design patterns, interfaces, hot-spots, etc – embedded in an often tangled web of interconnections and semantic relationships.

The need to tackle quantitative complexity is of course not restricted to frameworks. It also arises in the case of conventional application design where the classical solution is to subdivide the application into *subsystems* that partition the space of components making up the application. The subsystems are designed to be as self-contained as possible and to have minimal coupling. This makes it possible, to some extent, to develop them independently of each other thus reducing design complexity.

This approach works well for individual applications that are just the sum of their components but is unsuitable for frameworks that consist of more than just components for they also include design patterns and abstract interfaces among their basic constructs. In fact, as should be clear from the discussion in section 5, in the case of frameworks, components take second place to design patterns and interfaces that are the true foundations upon which frameworks are built. The behaviours implemented by the core components are normally implied by the design patterns and are therefore conceptually subordinate to them while default components are simply implementations of abstract interfaces.

Given these relative priorities among the constructs making up a framework, the transposition of the subsystem approach to framework design requires a shift of focus from components to abstract interfaces and design patterns. Simplification in other words must be achieved by partitioning not the space of components, but that of design patterns and abstract interfaces. The framelet term was introduced to designate non-overlapping groups of logically related design patterns and interfaces. The two ways of partitioning the design space – through framelets and through subsystems – are schematically contrasted in the next figure.



Design Simplification in Frameworks and Applications

It is normally possible to partition the space of abstract interfaces and design patterns into homologous subsets because the abstract interfaces exist to support the variability and adaptability introduced by the design patterns and therefore abstract interfaces can normally be unambiguously associated to design patterns.

Hot-spots are not primitive constructs in a framework because they are reducible to design patterns and abstract interfaces but it is interesting to note that, precisely for this reason, hot-spots can be assigned to framelets. Framelets, therefore, can also be defined as a partitioning of a framework into subsets of logically related abstract interfaces, design patterns and hot-spots.

In practice, logical relationship means that each framelet addresses a specific design problem in the framework. Framelets in other words become *units of design* with the framework being obtained by combining the framelets.

Some of the characteristic features of framelets that follow from all the above are:

- *Small size:* since they are made up of closely related constructs, framelets will typically be small. In the AOCS framework, framelets consist of 2-3 design patterns and a handful of abstract interfaces.
- *No execution control assumptions*: since they are intended to be integrated with each other to form the framework, framelets must not assume that they have control of the application execution.
- *Self-standing*: since they address a specific design problem, framelets will normally be self-standing in the sense that they could be used in isolation from the other framelets (perhaps in a different framework).

It is noteworthy that all the above features contribute to simplifying design and justify the claim that framelets help reduce the complexity of the framework design process.

It is important to stress that there is no guarantee that framework components can be mapped to single framelets. Indeed, in most cases they will implement several interfaces possibly belonging to different framelets and will participate in several design patterns possibly provided by different framelets. The framelet structure cannot therefore be imposed upon the space of components each of which will normally straddle framelet boundaries. However, specific components are normally introduced in response to the design need of a particular framelet and in this sense they will be said to be *exported by or to belong to the framelet*. This will normally imply that the component in question implements some of the interfaces defined by the framelet or that it serves as an implementation vehicle for some of its design patterns but it does *not* exclude implementation of interfaces or design patterns exported by other framelets.

7.1 AOCS Framework Framelets

The AOCS framework is made up of the following framelets:

- *System Management Framelet*

  This framelet proposes an architectural solution to the problem of performing the same action – such as a software reset – on all objects in the AOCS software. The system actions covered by the system management framelets are software reset, configuration check and configuration reset.

  The framelet enhances re-usability because it decouples the task of *management of the system actions* from their *implementation*.

  This framelet is covered in section 9.

- *Object Monitoring Framelet*

  This framelet proposes an architectural solution to the problem of monitoring an object and its properties. Object monitoring is useful for FDIR purposes and for managing operational mode changes.

  The framelet enhances re-usability because it decouples the task of *managing the monitoring process* from that of *performing the monitoring checks*.

  This framelet is covered in section 12.

- *Inter-Component Communication Framelet*

This framelet proposes an architectural solution to the problem of managing the data exchanges among framework components and it defines a standard interface for the data representing AOCS-specific quantities exchanged among these components.

This framelet enhances reusability in two ways. Firstly, through the use of shareable data areas for inter-component communications, it decouples the *production* of data and events from their *consumption*. Secondly, by allowing uniform treatment of all data types, it makes component interfaces independent of the type of data they process.

This framelet is covered in section 13.

- *Sequential Data Processing Framelet*

  This framelet defines a pattern for the sequential processing of AOCS data.

  The framelet enhances reusability because it provides a standard interface for components that must perform sequential processing on AOCS data and because it allows easy combination of data processing blocks.

  This framelet is introduced in section 14. Three architectural options are discussed in sections 15 to 17.

- *Aocs Unit Framelet*

  This framelet defines an architecture to manage external AOCS units.

  The framelet enhances reusability because it provides a standard interface for AOCS units that decouples the managers and users of unit data, from the units themselves.

  This framelet is covered in section 18.

- *Unit Reconfiguration Framelet*

  This framelet proposes an architecture to handle reconfiguration of AOCS units.

  The framelet enhances reusability because it decouples the task of *managing unit reconfigurations* from the task of *processing of unit data* and from the reconfiguration algorithms.

  This framelet is covered in section 20.

- *Operational Mode Management Framelet*

  This framelet proposes an architectural solution to the problem of managing operational mode changes in AOCS objects.

The framelet enhances reusability because it separates the implementation of mode-specific algorithms from the mode switching logic.

This framelet is covered in section 21.

- *Manoeuvre Management Framelet*

  This framelet proposes an architectural solution to the problem of managing manoeuvres such as wheel unloading, slews, delta-V, etc.

  The framelet enhances reusability because it separates the task of *managing* the manoeuvres from the task of *carrying them out*.

  This framelet is covered in section 22.

- *Failure Detection Framelet*

  This framelet defines an architecture to handle failure detection tasks.

  The framelet enhances reusability because it decouples the task of *managing the failure detection function* from the task of *carrying out failure detection tests.*

  This framelet is covered in section 24.

- *Failure Recovery Framelet*

  This framelet defines an architecture to handle failure recovery tasks.

  The framelet enhances reusability because it decouples the task of *managing the failure recovery function* from the task of *carrying out the failure recovery actions.*

  This framelet is covered in section 25.

- *Telecommand Framelet*

  This framelet defines an interface to which telecommands must conform and defines an architecture for the telecommand manager.

  The framelet enhances reusability because it decouples the task of *managing the telecommands* from the task of *executing them*.

  This framelet is covered in section 26.

- *Telemetry Framelet*

  This framelet defines an interface to which objects that can be stored in telemetry must conform and defines an architecture for the telemetry manager.

The framelet enhances reusability because it decouples the task of *managing the telemetry* from the task of *writing objects' state to telemetry*.

This framelet is covered in section 27.

- *Controller Management*

  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.*

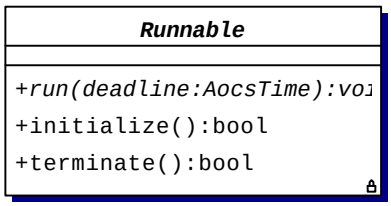
  This framelet is covered in section 28.

8 SCHEDULING ISSUES

The AOCS software is usually organized as a multi-tasking system. In the overwhelming majority of cases, cyclic and non-preemptive scheduling is used. For reasons of efficiency and simplicity, scheduling is done by dedicated code (as opposed to using a standard real-time system or ADA's built-in support for tasking). In the future, more sophisticated scheduling policies may be used and then the AOCS will have to be implemented upon an RTOS.

The AOCS framework assumes that the AOCS is periodic but does not otherwise make any other assumptions about the scheduling policy implemented by the AOCS applications to be instantiated from it.

The framework introduces the *active object* abstraction. An active object is characterized by the following abstract interface:



The key method is `run`. A call to `run` will cause a component to perform all the actions associated to the current period.

Method `run` takes one single parameter representing the maximum time available for the execution of the method. This parameter is not used in the current version of the framework but could be used in the future to implement some kind of cooperative scheduling policy.

Consider for instance a telecommand manager that maintains a queue of telecommands due for execution. As will be seen in section 26, the telecommand manager is an active component. If it possessed an estimate of the time it takes to execute a telecommand (or at least has an upper bound for this quantity), it could check how many telecommands it can execute in the current AOCS period.

In another example, consider a component that performs a background check on program integrity by computing the checksum on the code memory. Such a component would be likely to implement interface Runnable and if it knew how long it takes to process each memory location, it could determine how far it can go in carrying out the checksum check before having to return from method `run`.

Methods `initialize` and `terminate` are available to encapsulate actions that have to be performed when a component is first scheduled and when it is descheduled.

The decoupling of the scheduling policy from the framework architecture is obtained by stipulating that method `run` can only be called from outside the framework (normally by an external scheduler or by an interrupt servicing routine) and that active components that are also core components do not to make any assumptions about the frequency with which their method `run` is called or about the source of the call. Whenever they are activated by a call to method `run`, they take whatever action is appropriate at the time they are called without regard to how recently they were last called in the past or to how soon they may expect to be called again in the future. This insulates them – and hence the framework – from the AOCS scheduling policy.

As discussed in section 5.5, the framework conceptualizes an AOCS application as a bundle of functionalities. As will be seen in the remainder of this document, the functionality managers are active components and must therefore implement interface `Runnable`. Functionality managers are obvious examples of core components since they are application-independent and hence their `run` method are implement not to make any assumptions about how frequently they are called.

Obviously, the components that implement the AOCS functionalities and that customize the functionality managers will rely on their methods being invoked according to some timing pattern but these components are application-specific and are defined and configured by the application developer at application-level. Their dependency on a particular scheduling policy therefore does not carry over to the framework.

As an example, consider the controller functionality described in section 28. A typical concrete controller component will normally implement a digital filter and therefore, in order to work properly, it will need to have its internal state propagated at regular intervals. Such a component will therefore have a built-in assumption about the frequency with which some of its methods must be called. No such assumption, however, applies to the controller manager component that sees the concrete controllers through an abstract interface whose operations do not imply any timing requirements.

The framework architecture therefore achieves independence from the scheduling policy of the AOCS application by confining scheduling assumptions to the application-specific components.

It should finally be noted that some scheduling policies require the implementation of synchronization mechanisms to coordinate access to shared resources. The framework

regards any public method as potentially giving access to a shared resource and therefore as potentially in need of an access synchronization mechanism. The type of mechanism to be used, however, is treated as an implementation rather than an architectural issue and is therefore neither provided nor dictated by the framework.

8.1 The AOCS Framework and HRT-HOOD

The previous section argues that the AOCS framework architecture is independent from the scheduling policy. In one notable case, this independence was concretely verified by demonstrating that the AOCS framework is compatible with HRT-HOOD.

HRT-HOOD (see RD7-8 and RD13-16) is a modification of the HOOD methodology that was specifically designed to produce systems that can be statically analyzed for their timing properties. In other words, a software system obtained following the rules prescribed by HRT-HOOD is guaranteed to be amenable to static schedulability analysis.

The HRT-HOOD method constrains a software system to be made up of objects of four types:

- *Passive Objects*

  These are objects that have no control over when invocations of their operations are executed and do not spontaneously invoke operations in other objects.

  A passive object is either used by only one other object or it can be used concurrently without error.

- *Protected Objects*

  These are objects that may control when invocations of their operations are executed (through monitor or other synchronization mechanisms) but do not spontaneously invoke operations in other objects.

  Protected objects model access to resources that are shared across threads.

  Protected objects must be analyzable for the blocking time they impose upon their callers.

- *Cyclic Objects*

  These are objects that represent periodic activities and they may therefore spontaneously invoke operations in other objects. The only operations they expose are requests which demand immediate attention (ie. they represent asynchronous transfer of control requests).

- *Sporadic Objects*

These are objects that represent sporadic activities and they may therefore spontaneously invoke operations in other objects. A sporadic object has a single operation which is called to invoke the sporadic and one or more operations which are requests that demand immediate attention (ie. they represent asynchronous transfer of control requests).

In order to ensure static analyzability, HRT-HOOD imposes some constraints on how objects *use* each other's operations and on how they can be *included* within each other. Loosely speaking, cyclic and sporadic objects can call any operation on any other object whereas protected objects cannot block once they start executing. Passive objects can only call operations on other passive objects. Similarly, cyclic and sporadic objects can include objects of any type whereas protected objects can only include protected or passive objects. Passive objects can only include other passive objects. Finally, deadlines should be assigned to cyclic and sporadic objects and worst-case execution times should be assigned to all externally visible operations.

When the above restrictions – on object type, on their use and include relationships and on their attributes – are respected, then HRT-HOOD ensures that the resulting system can be statically analyzed for its schedulability.

The HRT-HOOD method was conceived as a *design* tool. Owing to its HOOD heritage, HRT-HOOD is an *object-based* design methodology (see section 6.1). The AOCS frameworks however is *object-oriented*. Thus, HRT-HOOD would appear not to be directly applicable to its design.

Both an object-based and an object-oriented design process result in a final software system that is made up of objects that expose certain operations. Given such a system, it is possible to ask whether its architecture is of a kind that would be generated by the HRT-HOOD design process. In order to assess whether this is the case, the following questions must be answered:

- Can all the objects in the system be represented by objects of the types allowed by HRT-HOOD for terminal objects, namely *sporadic, cyclic, protected* and *passive*?
- Are the ways in which the objects *use* each other's operations compatible with HRT-HOOD rules?

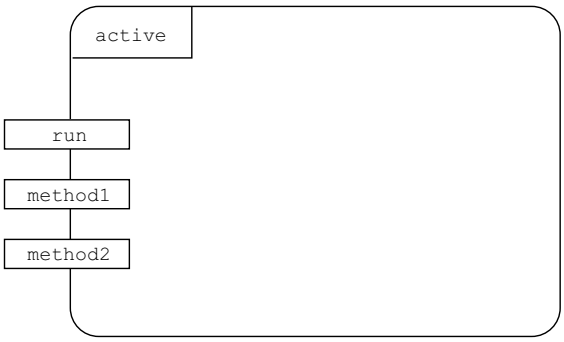
Used in this way, HRT-HOOD becomes an *analysis* tool to investigate whether the schedulability of a given software system can in principle be statically analysed.

Software systems for which the two above questions can be answered in the affirmative will be said to be *HRT-HOOD compatible*. Such systems *look like* they were designed with HRT-HOOD even if they were not actually designed using an HRT-HOOD methodology.

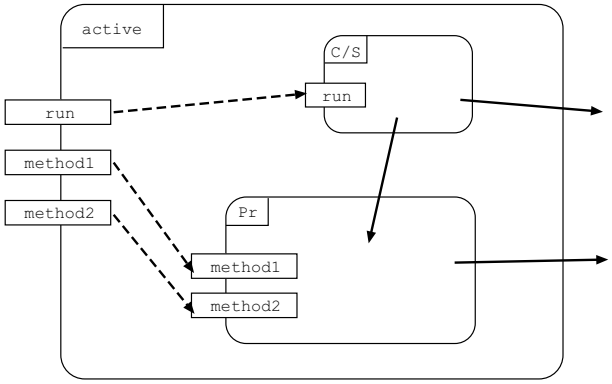
In the context of a particular framework project, one can ask the following question: “are the concrete applications that can potentially be derived from the framework HRT-HOOD compatible?”. This is an interesting question to ask because an affirmative answer indicates that the framework generates applications whose schedulability is statically analysable. In the case of the AOCS framework project the answer to this question is affirmative.

The first step to achieving HRT-HOOD compatibility for the AOCS framework is to recast the AOCS objects in terms of the canonical object types admitted by HRT-HOOD. Objects other than those implementing the `Runnable` interface should be mapped to HRT-HOOD protected objects. This is because the operations they expose can in principle be called by any component from any thread of execution and the framework gives no guarantee that public methods can be safely accessed by concurrently executing components.

Objects implementing the Runnable interface instead correspond to HOOD’s *active* objects which are not permitted by HRT-HOOD. The next figure shows a runnable object using HRT-HOOD notation:



This object exposes the `run` method and two other methods. It can be decomposed as follows:



Dashed arrows represent the way a method in a higher level object is mapped to a method in an included object and solid arrows representing *use* relationships.

The runnable object can therefore be decomposed into a protected object and a second object that presents a single entry point – method `run` – that is activated either by a scheduler or by some asynchronous event. Such an object can be mapped to either an HRT-HOOD *sporadic* or to a *cyclic* object. The AOCS framework does not make any assumptions about the scheduling policy and therefore it does not dictate whether runnable objects should be activated on a cyclical or on a sporadic basis. This is left as a decision for the application implementer.

This decomposition shows that runnable objects can be decomposed into HRT-HOOD terminal objects. It can therefore be concluded that all the objects directly defined by the AOCS framework can be mapped to HRT-HOOD objects.

The next step in attaining HRT-HOOD compatibility is to ensure that the constraints introduced by HRT-HOOD on the way objects use each other's operations are respected. This can be simply achieved in the AOCS framework by implementing method calls into shared objects as *protected synchronous execution requests*. This ensures that they are compatible with HRT-HOOD rules.

Thus, it can be concluded that applications generated from the AOCS framework are HRT-HOOD compatible and could therefore be subjected to static schedulability analysis.

9 SYSTEM MANAGEMENT FUNCTIONALITIES

A *system management function* is a function that is performed systematically on all AOCS objects present in the AOCS software at a given time. The *system manager* is the component responsible for performing system management functions.

Two system management functions are foreseen by the AOCS prototype:

- *System Reset*

  A system reset causes the internal state of all AOCS objects to be reset to a default state.

- *System Configuration Check*

  A configuration check causes the configuration of all AOCS objects to be checked.

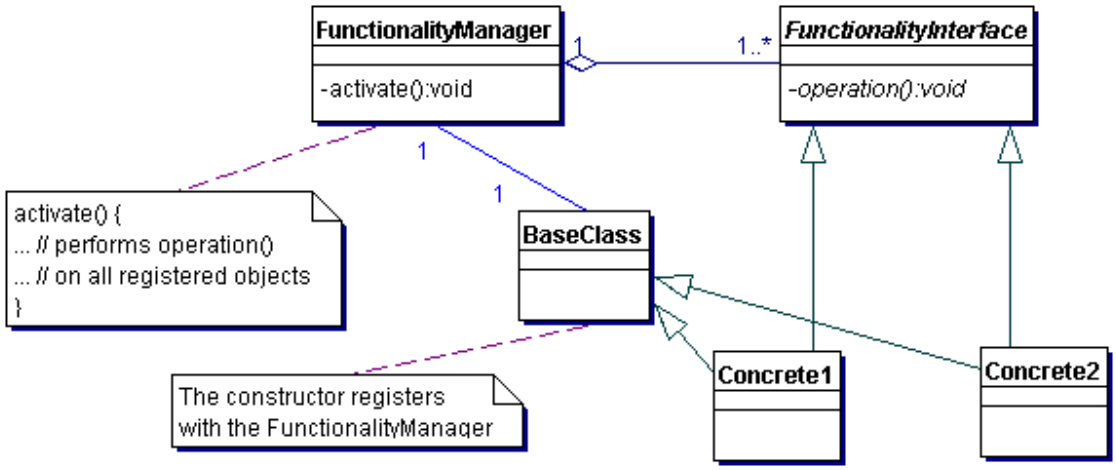
These two functions are described in greater detail in sections 9.2 and 9.3.

The system management functionalities are the subject of a dedicated framelet described in RD20.

9.1 The System Management Design Pattern

The system manager design pattern is introduced to address the problem of systematically performing the same set of operations on a target set of objects. The system manager design pattern is obtained by instantiating the manager meta-pattern.

The structure of the system manager design pattern is shown in the following UML diagram:



The functionality interface encapsulates the operations to be performed on the target set of objects. This interface is to be implemented by all objects in the target set. Additionally, all objects in the target set are to be derived from a single base class whose constructor registers with the functionality manager. Thus, the functionality manager is automatically provided with a list of all the objects in the target set. Deregistration is not required because AOCS objects are assumed never to be destroyed (see section 6.9).

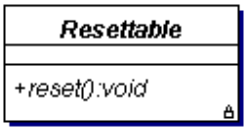
In the AOCS framework, the system manager design pattern is instantiated twice, once for the system reset function and once for the system configuration check function, as follows:

- The functionality manager is the `SystemManager` component
- The base class is `AocsObject`. This means that the set of objects upon which the system manager acts is the set of AOCS objects.
- The abstract interfaces associated to the system reset and system configuration checks functions are `Resettable` and `Configurable`, respectively

The functionality interfaces `Resettable` and `Configurable` are implemented directly by `AocsObject`. This ensures that they are implemented by all objects upon which the system manager is required to act (ie by all AOCS objects).

9.2 The System Reset Functionality

The system reset functionality is defined by the `Resettable` interface:



Its key method is `reset`. A call to method `reset` causes an object's internal state to be brought back to some initial default value. The term 'state' is used here to designate the attributes of an object that are updated as part of the object's performing its allotted task. The state of an object therefore does not cover those of its attributes that are normally set during the application initialization to configure it.

Thus, in the AOCS framework a reset operation can be performed at two levels. At o*bject's level* an individual object is reset by calling its `reset` method. At *system level*, a reset is performed by asking the system manager to reset all the AOCS objects in the AOCS application.

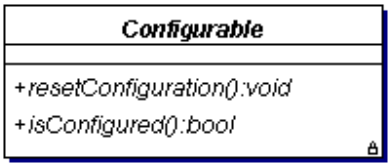
It should be stressed that in the AOCS framework a system reset should be distinguished from a system reboot. The system reboot will cause the AOCS application to be reloaded and restarted. A system reset will simply reset the state of all objects. The system reset therefore does not destroy the configuration of the AOCS application.

The system reset functionality is provided mainly for failure handling as it represents a less disruptive way of handling failures than a complete system reboot.

The `reset` method is a hot-spot because it has to be defined for each specific objects in an AOCS application.

9.3 The System Configuration Check Functionality

AOCS objects are configured during initialization for use in a specific application. Configuration will often imply loading plug-in components. In order to verify that an application is correctly configured, configurable objects expose a method that reports whether they are configured or not. This functionality is defined by interface `Configurable`:



The key method is `isConfigured` that returns false if the object is not properly configured. Often, this will simply mean that there are undefined references.

The system manager can call `isConfigured` on all AOCS objects in an application and check whether any of them reports `false`. In this way, the configuration of the entire application can be verified at any time.

The `isConfigured` method is a hot-spot because it has to be defined for each specific objects in an AOCS application.

9.4 Storage of Configuration Data

There is some information that should be preserved across resets. This in particular concerns the state of the *reconfiguration managers* (see section 0). This information should be preserved because it defines the health status of redundant functionalities or units. The system manager is responsible for maintaining and restoring the state of the reconfiguration managers.

10 OBJECT PROPERTIES

A *property* is an attribute of an object that describes one aspect of its behaviour or of its internal state and that is accessible to external objects.

A property can be wrapped into an object. The object is called a *property object*.

Properties are useful for failure detection purposes: the mode of change of an object's properties may indicate that a fault has arisen. Objects also monitor each other's properties when they want to coordinate their behaviour.

10.1 Properties

The property model proposed here is derived from the JavaBeans component architecture (see RD6).

Objects expose their properties through `get` and `set` methods. A property of name <`PropertyName`> and type <`PropertyType`> has getter and setter methods that conform to the following signature and naming pattern:

```
<PropertyType> get<PropertyName>();
void set<PropertyName>(<PropertyType> value);
```

Objects that wish to directly monitor the object's property call its `get` method. Objects that wish to set the property call its `set` method. Some properties are read-only and do not provide a set method. Properties of boolean type may have an `is<PropertyName>` method to check their value.

10.2 Property Objects

It is often useful to be able to treat properties as objects of the same type. Consider for instance the case of a failure detection manager. This object will need to keep a list of properties that must be monitored regularly. It would be convenient if an array of properties could be defined and if some kind of standard operations could be performed on all properties.

In order to make this possible, *property objects* are defined. These are objects of class `Property` that encapsulate a property and give it a standard interface and type.

To see the advantages of having property objects, consider the case of the failure detection manager that needs to perform systematic checks on several unrelated properties. It can do so by maintaining and inspecting a list of properties as follows:

```
Property properyList[20];
float value;
. . .
for (int k=1; k<20; k++)
{    value = property[k].get();
     . . . // process 'value'
}
```

where it is assumed that class `Property` exposes a `get`() method to retrieve the value of the property and that this value is (or can always be cast to) a float.

The next two sections will show how the processing of the property value can be made in a generic fashion (see in particular the last example in section 12.1).

10.3 Comparison with Data Items

Property objects encapsulate variables and provide read access to them. They are thus very similar to data items (see section 13.6). Indeed, one could think of using data items in lieu of property objects (or vice versa). This is not done here because the intended usage for the two types of objects is very different. Data items are meant to facilitate linking components to their data sources and will therefore be used extensively in any AOCS application. They must therefore be light-weight and efficient. By contrast, property objects are meant for monitoring purposes and may have to contain other in addition to the bare value of the property (eg. a property identifiers, an identifier of the property type, etc).

10.4 Reusability and Extensibility Issues

The encapsulation of properties in objects furthers software reusability because it helps decouple the task of *managing* the properties from that of *processing their values.* This decoupling is in particular crucial to building a reusable failure detection manager in section 24.

11 CHANGE OBJECTS

The next section deals with the monitoring of objects. The term monitoring refers to the observation of changes in property values. At the most basic level, monitors are interested in *any* change in the monitored property. More frequently, however, they are only interested in *certain types* of changes.

For instance, a failure detection manager may be interested in knowing when the output of a sun sensor exceeds certain pre-specified boundaries, as this might indicate that the spacecraft is going out of control or that the sensor itself is faulty. If the failure detection manager were notified every time the sun sensor output changes, it would have to implement a sensor-specific filtering mechanism to identify the out-of-range changes that might signal a failure. In order to avoid burdening monitors with the need to implement highly specific (and hence non-reusable) filters, the concept of change is encapsulated in objects which will be called *change objects*.

Thus, a change object is an object that encapsulates a specific time profile for a property value. The change is said to have occurred when the property value has violated the time profile.

11.1 Change Types

A change object may encapsulate any time profile. Typical change objects that are likely to be useful in an AOCS (and that are therefore provided as default components by the framework) include:

- *Simple change*: the change occurs when the monitored property changes its value.

  Example: the mode manager of an object registers with the mission manager to be notified whenever the mission mode changes.

- *Out-of-range change:* the change occurs whenever the monitored property moves outside a pre-defined range.

  Example: when a property drifts outside a range of values representing physically possible values, a fault is likely to have arisen. Hence, the failure detection manager registers with the objects representing sensors to be notified when their outputs leave the permitted range.

- *Delta change*: the change occurs whenever the monitored property changes by more than a pre-defined delta value.

Example: when the output of a sensor suddenly "jumps" by more than a certain threshold, a fault is likely to have arisen. Hence, the failure detection manager registers with the objects representing sensors to be notified when their outputs change too abruptly.

- *Filtered delta change*: monitors are notified whenever the filtered value of the monitored property changes by more than a pre-defined threshold.

  Example: the failure detection manager of the previous example may wish to ignore "spikes" in sensor outputs that might be due to freak conditions not indicative of a real fault. In that case, it registers with the sensor objects to be informed whether its output, after filtering for spikes, exhibits a suspicious change.

Other types of changes may be defined on an *ad hoc* basis.

11.2 Change Object Definition

A change object implements the following interface:

```
Interface ChangeObject {
     bool checkValue(float value);
}
```

`ChangeObjects` objects are responsible for carrying out the check on the property value and deciding whether a specified change criterion is met. It is assumed that the property value is always a float or can be meaningfully converted to a `float`.

Method `checkValue` takes as argument the current value of the property being monitored and it returns `true` if this value represents a violation of the change profile encapsulated by the change object. Note that change objects will in general encapsulate profiles that extend over time and hence change objects need to keep track of past value of a property.

As an illustration, consider the implementation of a class representing a simple change:

```
Class SimpleChange: public ChangeObject {

     float oldValue;

     bool checkValue(Real newValue) {
          if (newValue != oldValue) {
               return true;
          }
          oldValue := newValue;
```

```
            return false;
        }
    }
```

Here method `checkValue` checks whether the new value is different from the old one (which it remembers from the previous activation) and, if it is, it returns `true` to signal that the change has occurred.

As another example consider a change that must detect out-of-range changes in values. This could be built as follows:

```
Class OutOfRangeChange: public ChangeObject {

    float maxValue, minValue;

    bool checkValue(float value) {
        if ( (value>maxValue) || (value<minValue) ) {
            return true;
        }
        return false;
    }
}
```

In this case, `checkValue` return `true` if the value it inspects is outside a pre-defined range.

Section 12.1 shows an example of how the change object concept can be used to perform systematic checks on generic properties.

Change objects are framework hot-spots since applications must supply the exact definition of the change profile in which they are interested.

11.3 Reusability and Extensibility Issues

The encapsulation of change types in objects furthers software reusability because it helps decouple the task of *managing* the checks on property changes from that of performing the checks. This decoupling is in particular crucial to building a reusable failure detection manager in section 24.

Extensibility is furthered because it is easy to subclass `ChangeObject` to deal with new types of changes tailored to the needs of specific missions.

12 PROPERY MONITORING

The term *monitoring* or *monitoring action* refers to the observation of a change over time in the value of a property.

The object performing the monitoring action is called the *monitor*. The property being monitored is called the *monitored property*.

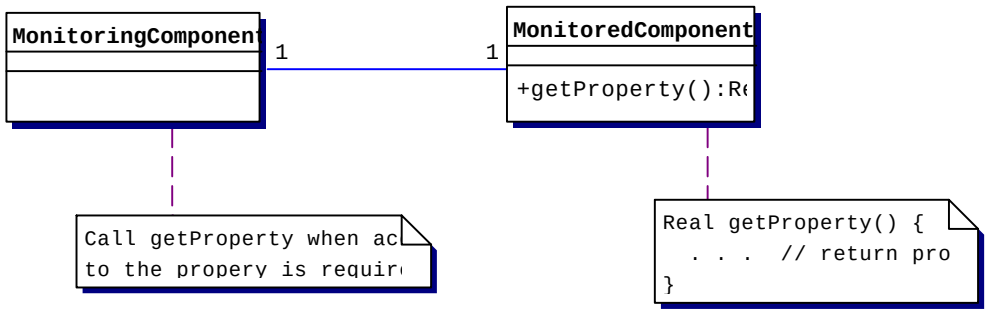
In general, objects need to monitor each other in order to coordinate their behaviour and for purposes of failure management.

The monitoring mechanism proposed here is loosely based on the JavaBeans property model (see RD6) and on the *observer design pattern* from RD1.

Two monitoring mechanisms are identified that are encapsulated in two design patterns: the *direct monitoring design pattern* and the *monitoring through change notification design pattern.*

12.1 The Direct Monitoring Design Pattern

The direct monitoring design pattern is introduced to address the problem of granting access to an internal property to a monitoring component. This pattern is very simple as it prescribes that the monitor directly accesses the monitored property through its getter methods:



Thus, for instance, an object that is interested in monitoring property <property> belonging to object <object> will periodically access it by calling method get<property> on <object>.

If the monitoring object is interested in detecting changes of a certain type, recourse can be made to a change object. Consider for instance the case of a monitored object of type SunSensor and suppose that the monitor is interested in detecting whether its X output is out of range. This can be done with the following statements:

```
output = aSunSensor.getXoutput();
if ( aOutOfRangeChange.checkValue(output) )
     . . .     // X output is out of range!
```

Object `aOutOfRangeChange` encapsulates an out-of-range change (see section 11.2 for a definition of its class).

In another example, consider a failure detection manager that keeps a list of property objects that must be checked periodically and systematically. To each property, a change object is associated that specifies the type of change that needs to be detected:

```
ProperyObject propertyList[20];
ChangeObject changeList[20];
float value;
ChangeEvent evt;
. . .
for (int k=1; k<20; k++)
{    value = propertyList[k].getValue();
     if ( changeList[20].checkValue(value) )
          . . . // the change has occured!
}
```

This example generalizes the example of section 10.2.

12.2 The Monitoring through Change Notification Design Pattern

This design pattern is introduced to address the problem of a monitoring component that wishes to be automatically notified when a change of a certain type occurs in a specific property in which it is interested.

The JavaBeans component architecture (see RD6) implements a simple mechanism of monitoring through change notification with the *bound property* mechanism: property monitors can register with the owner of a property to be notified whenever the value of that property changes.

The JavaBeans mechanism is rather limited because it does not discriminate between different types of change: any change results in a notification. In the case of the AOCS, the change notification design pattern uses change objects to ensure that notification of a monitor only takes place when a change of a pre-specified type has occured.

The property owner must allow monitors to register and unregister their interest in a property. It does so by exposing methods with signatures like:

```
void add<Property>Monitor(Monitor* monitor, ChangeObject* changeObject);
void remove<Property>Monitor(Monitor* monitor);
```

When a monitor `monitor` calls `add<Property>Monitor` it notifies the property owner that it is interested in property <`property`>. The second parameter in the method call indicates the type of change in which the monitor is interested.

When a monitor `monitor` calls `remove<Property>Monitor` it notifies the property owner that it is no longer interested in property <`property`>.

The property owner maintains a list of registered monitors together with their change objects and every time the property is changed, the new value is passed through its corresponding `checkValue` method.

Monitors are notified through a call to method `propertyChange`. They must therefore implement the following interface:

```
Class PropertyMonitor {
     void propertyChange(…)=0;
}
```

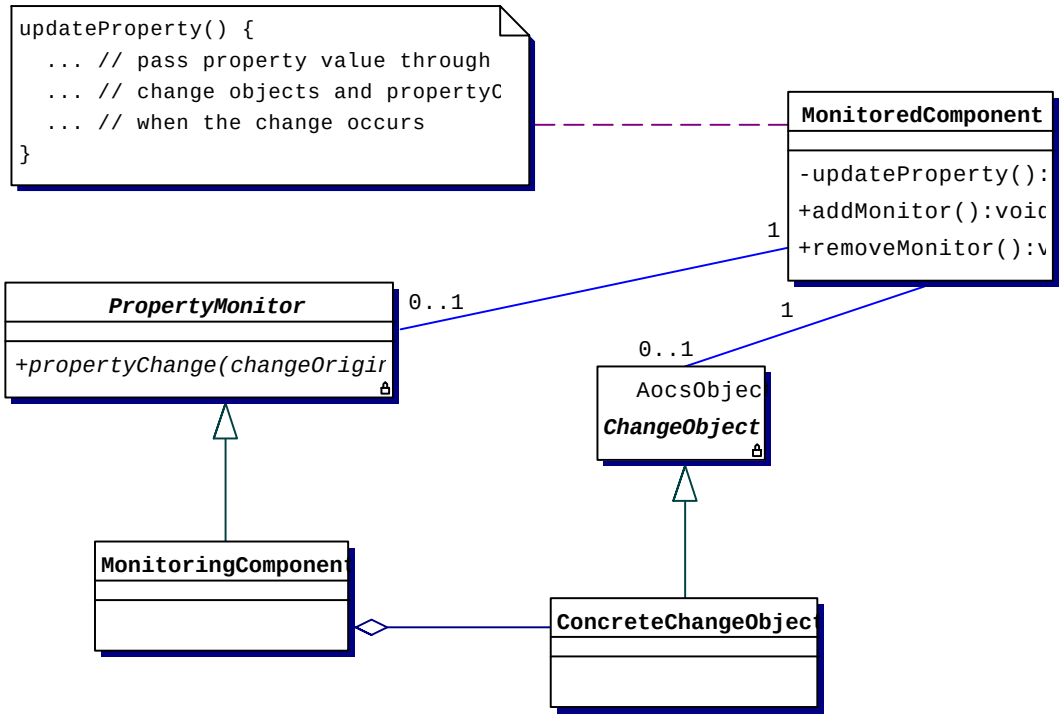
The arguments of `propertyChange` serve to identify the property that has undergone the change and its owner. The code for updating the value of property `property` would then look like this:

```
property := newValue;
for (all currently registered monitors) do {
     if (changeObject->checkValue(newValue) // a change has occurred!
          monitor->propertyChange(…);       // notify the monitor
}
```

Note that, as in the JavaBeans architecture, the monitors are notified *after* the change has taken place.

In keeping with the notation used by the JavaBeans architecture, properties that offer a notification mechanism are called *bound properties*.

The UML diagram of the monitoring through change notification design pattern is:



The action to be taken in response to the detection of a change in a monitored property is application-specific and therefore method `propertyChange` defines a framework hot-spots (the *property change* hot-spot).

12.3 Change Event Adapters

In the sample implementation proposed above, notification of a change causes a generic method `propertyChange` to be invoked on the property monitor. Since the same object might be monitoring several properties or states, monitors have to discriminate among several types of change notifications. Sometimes, it may be more efficient to use some kind of adaptation mechanism that directly routes the property change event to its handler. This would be similar to the *event adapter* mechanism of the JavaBeans architecture.

In the present version of the framework, the identification of the change that triggered a call to `propertyChange` is done though its parameters. Use of an event adapter mechanism miht be introduced in future versions if it is found that monitors typically have to monitor several change sources.

12.4 Autocode Generation

It is worth emphasizing that commercial tools exist that allow users to configure the properties and property change notification mechanisms of JavaBeans through graphical user interfaces. Beans are represented as icons and users can define and link the properties of different beans entirely through graphical means. The tool then automatically generates the code that implements the properties and the links graphically specified by the user.

Such tools could be adapted to the property mechanism described here to allow autocode customization of AOCS components.

12.5 Alternative Implementation

The JavaBeans architecture also envisages *constrained properties* whose change can be vetoed by the property monitor. It is unclear whether constrained properties are needed in the AOCS. They offer flexibility in coordinating state changes between objects but, by the same token, they also increase the coupling between the objects. Since the objective is to minimize inter-object coupling, the present version of the framework does not use constrained properties but this decision may have to be reviewed in later versions.

12.6 Reusability and Extensibility Issues

The architecture proposed here for object monitoring furthers software reusability because it decouples the task of *managing* the monitoring process from that of performing the monitoring checks. Monitors thus become independent from the type of monitoring checks that are performed.

13 INTER-COMPONENT COMMUNICATION

The AOCS software is built as a collection of software components. Framework components can exchange three types of information:

- *Command Requests*

  Command requests are sent by a component to another to force a change in the latter's internal state. Examples include: command to reset a component, command to reconfigure a unit proxy component, etc.

- *Event Data*

  Event data are produced asynchronously by objects that wish to signal a change in their internal state or the occurrence of some event. Examples of events include the notification of an error condition, the notification of a unit reconfiguration, the notification of a successful telecommand execution, etc.

- *Cyclical Data (also simply called "data" or "AOCS data")*

  Cyclical data are data that are produced or consumed on a periodic basis by AOCS components. Examples include the sensor outputs produced by sensor objects, the torque requests produced by the controller object, the telemetry data produced by the telemetry manager object, etc.

In principle, three mechanisms can be identified for inter-component communication:

- message passing;
- shared memory;
- method calls across component boundaries.

With a message passing mechanism the information transfer is mediated by a transmission channel over which a data packet is transferred that contains the information item. This mechanism is the most general but also carries the heaviest overhead. It would be useful if the AOCS software were distributed. This is not the case at present and is not expected to be the case in future systems. Hence, this mechanism is not used in the framework.

Shared memory has the advantage of decoupling production of data from consumption of data. The main disadvantage is that data integrity requires access in mutual exclusion. In the proposed AOCS architecture, shared data areas are used for cyclical and event data.

Command requests could in principle be packetized as events and passed through a shared data area. This approach would have the advantage of confining the execution threads of active objects within their own boundaries.

However, this mechanism would introduce a delay between the time when the command is issued (ie. the time when the command event is deposited in the shared data area) and the time when the command is executed (ie. the time when the command is picked up from the shared memory area). It would also cause both memory and execution overheads.

For these reasons, command requests from a component A to a component B are implemented in the framework as calls performed by A upon methods exposed by B. Thus, for instance, to the command to reset a component there must correspond a `reset` method that must be called on the component to be reset: if object A wishes to send a "reset" command to object B, it will call its `reset` method.

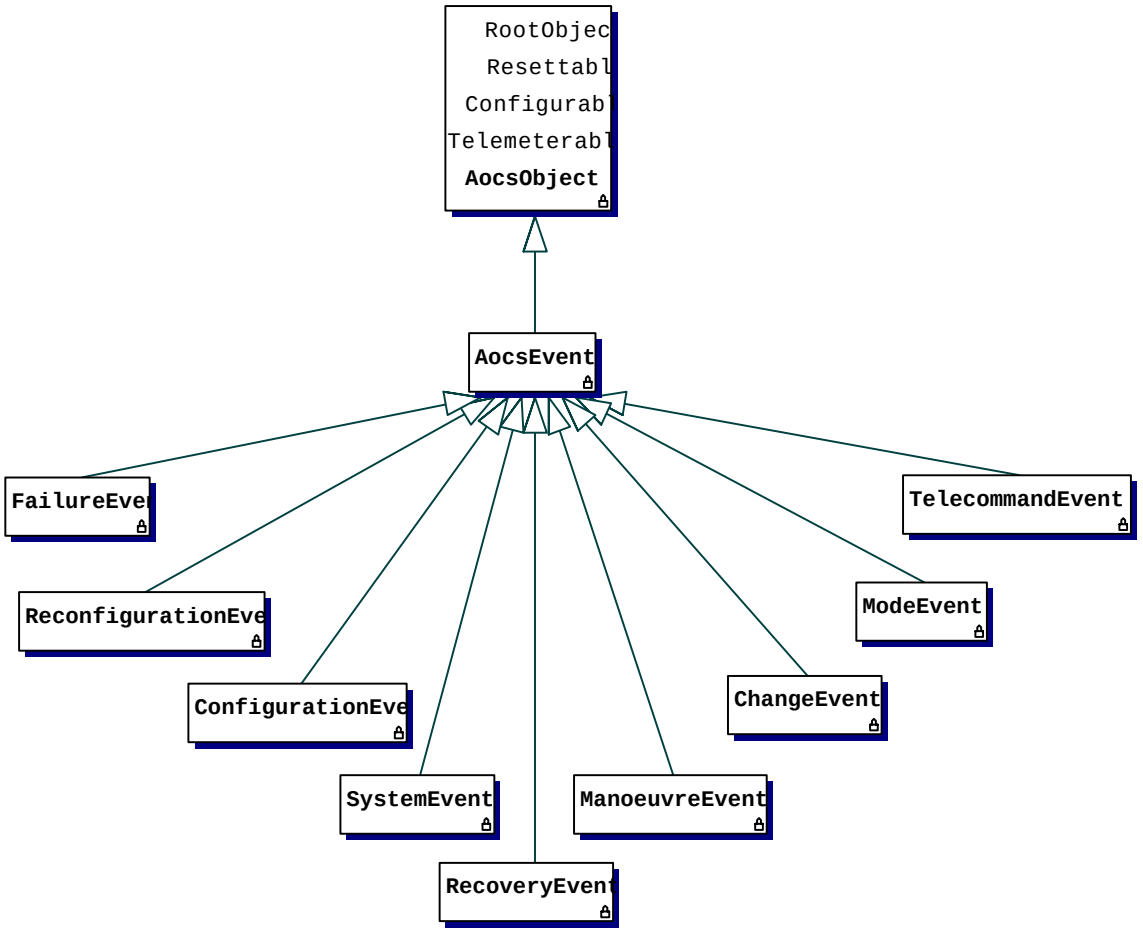
13.1 AOCS Events

AOCS are encapsulated in objects that are derived from the base class `AocsEvent`. Class `AocsEvent` defines the three following read-only properties for an event:

- time stamp (the time when the event was created)
- event creator
- event identifier

Class `AocsEvent` can be subclassed to construct more specific types of events. For example, an event flagging the failure of a failure detection test might include information on the failed test.

The main event classes in the AOCS framework are shown in the UML diagram below. The event class names are self-explanatory.



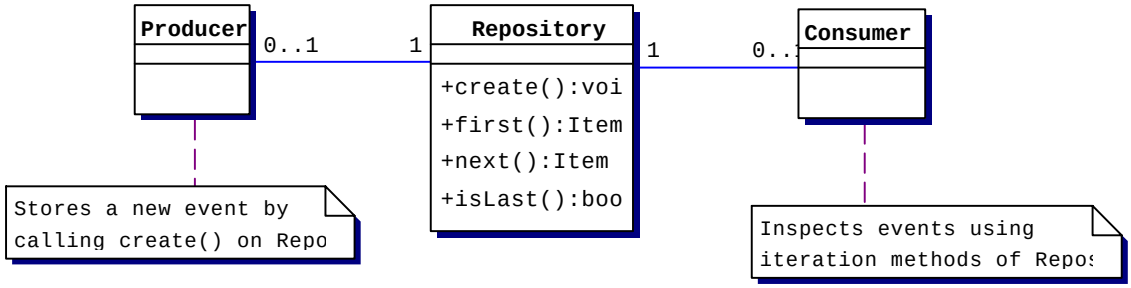
The event subclasses implemented by the framework are listed in the table:

Event Subclass Name	Event Description
`TelecommandEvent`	Describes telecommand-related events
`FailureEvent`	Describes failures detected by the AOCS
`RecoveryEvent`	Describes failure recovery actions
`ManoeuvreEvent`	Describes manoeuvre-related events
`ChangeEvent`	Describes changes in a property value
`ReconfigurationEvent`	Describes a component reconfiguration
`ModeEvent`	Describes a change in operational mode
`ConfigurationEvent`	Describes an error in the configuration of an object

`SystemEvent`	Records the occurrence of a system event.

13.2 The Shared Event Design Pattern

This design pattern is introduced to address the problem of allowing components to share access to event objects that are generated asynchronously. The pattern is illustrated in the following UML diagram:



Both the producer and the consumers of the events have access to a repository component that acts as a shared data area for the exchange of the events. The event producer calls method `create` to ask the repository to create and store a new event. The event consumer uses the iteration methods to retrieve all the events in the repository and process them as necessary.

13.3 Event Repositories

Event repositories are based on the shared event design pattern. They are the shared data areas where AOCS events are stored. Event producers store events in a repository and event consumers inspect repositories to verify whether any events of interest to them have been produced.

Event repositories serve two purposes:

- they act as *factories* of objects of type `AocsEvent`;
- they act as *storage points* for objects of type `AocsEvent`.

Events by their very nature must be created dynamically. However, in an embedded system, it is undesirable to allocated memory dynamically. Hence, each event repository pre-allocates memory and calls to its `create` method will simply result in a pre-allocated event area being filled with the description of the event.

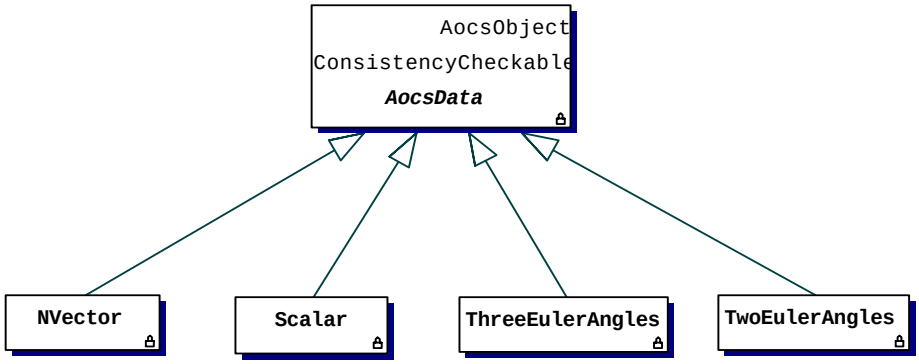
Event repositories expose methods to allow event consumers to iterate through all events currently in the repository.

Events are never explicitly destroyed. The repository has a pre-defined capacity and when that capacity is reached, the oldest event is overwritten.

Several event repositories are present, one for each category of logically related events. For each category of events of type `<EventType>`, an object of type `<EventTypeRepository>` exists (eg. Events of class `FailureEvent` are stored in repository objects instantiated from class `FailureEventRepository`).

13.4 AOCS Data

AOCS data or *cyclical data* represent the data periodically produced and consumed by AOCS objects. They are encapsulated in objects that are derived from the base class `AocsData`. This is a base type from which the various concrete types that are used in an AOCS are derived. The subclasses that are provided by the framework are:



Ideally, it would be desirable to have components that process AOCS data in a manner that is independent of their concrete type. Thus, components should be able to perform operations on the abstract `AocsData` type in the knowledge that this operation would be invisibly dispatched to the correct concrete sub-type.

This in particular applies to arithmetic operations: components should, for instance, be able to add together two `AocsData` variables without worrying about whether the addition is implemented as scalar addition, a vector addition, a quaternion addition, etc.

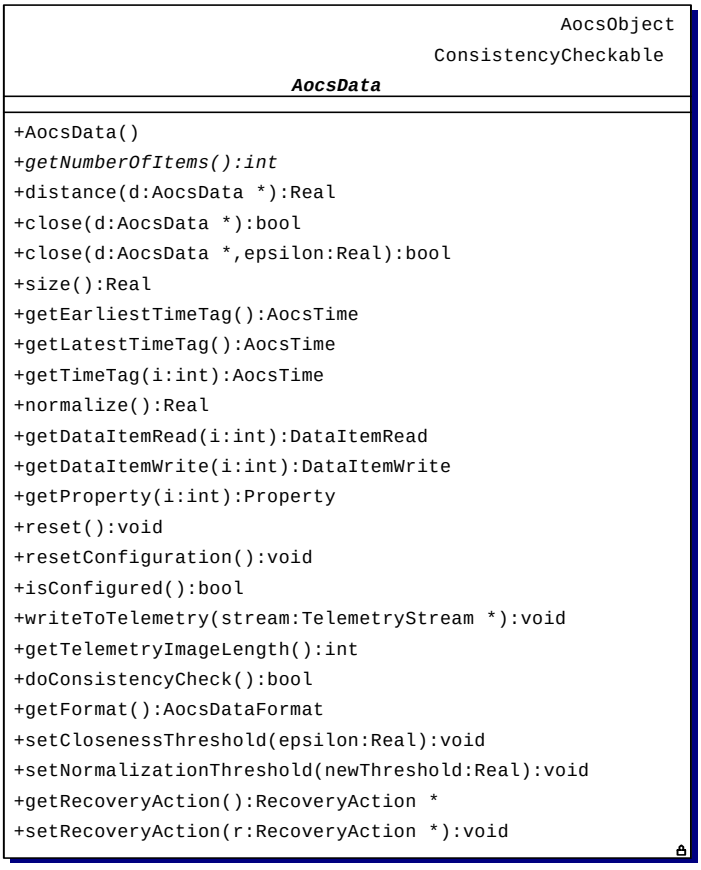
This concept was investigated but was found to be impossible to implement without making recourse to dynamic memory allocation. The concept adopted for the AOCS framework only partially attains the goal of generality of treatment.

Basically, it is recognized that AOCS data present two “faces” to the rest of the AOCS software that reflect the two main purposes for which AOCS components may need to access them. Some components access AOCS data for *housekeeping purposes* like integrity checking, telemetry reporting, failure investigations, etc. Other components access AOCS data for *computational purposes*, namely they use them as inputs or outputs of arithmetic computations.

The proposed architecture allows uniform treatment of data with respect to housekeeping access only. Computational access must instead be done on individual data items. The two forms of access are presented in the next two subsections.

13.5 Housekeeping Access to AOCS Data

All concrete data types – scalar, vectors, quaternions, etc – are derived from the base class `AocsData`. This base class offers all the functionalities required for housekeeping access to AOCS data. The `AocsData` base class is defined as illustrated in the figure:



The methods: `distance`, `equal`, `size` and `close`, are *metrics* methods. They assume that the concrete data types are defined in a space in which a metric can be defined. The basic method is `distance` that computes the distance between two AOCS data. In the case of scalars, this distance is simply the absolute value of the difference between the two items. In the case of vectors and Euler angles, the Euclidean distance is computed. In other cases, type-specific notions of distance must be implemented.

Method `equal` returns `true` only if the distance is zero.

Method `size` returns the distance of the object from the zero point in the metric space.

Method `close` returns `true` if the distance is less than `epsilon`.

By convention, attempts to compute a metrics function on a pair of data not of the same concrete type will result in some default large value being returned.

Metrics functions are useful for failure detection as they can be used to perform property monitoring. Consider for instance the monitoring of an attitude control error. If this error is represented as a variable of `AocsData` type, it will be possible to use its size methods to encapsulate it in a property object and hence to subject it to monitoring to ensure that it remains within certain boundaries.

Some data types - vectors, quaternions, and Euler angles - can be normalized. A call to method `normalize` causes the normalization to be performed. In the case of a quaternion or unitary vector, for instance, a call to `normalize` rescales the elements of the data type to ensure that their quadratic sum evaluates to 1. In the case of Euler angles, method `normalize` moves all angles to the interval [-180deg, +180deg].

Method `normalize` returns a `Real` indicating how far from the normal range the datum was. This return value can be used to perform consistency checks on data. Thus, for instance, if it was found that the quaternion representing the satellite attitude after integration of the Euler equations had a quadratic sum of, say, 1.1, then it is likely that the integration algorithm is misbehaving and the fact can be reported as an error.

The normalization threshold is set by calling method `setNormalizeThreshold`.

It is not possible to associate a single time tag to an `AocsData` as this is a composite type that contains several individual data items (eg. a `Vector` contains three elements that can be set at different times). Methods `getFirstTimeTag` and `getLastTimeTag` return. respectively, the oldest and the most recent time tags of the element in the AOCS data composite. Both may be of interest for failure detection purposes.

Class `AocsData` is derived from `AocsObject` which means that AOCS data inherit the `telemeterable` and `resettable` interfaces and can therefore be reset or written to telemetry.

Additionally, class `AocsData` implements interface `ConsistencyCheckable` and hence consistency checks can be performed upon its instances. An `AocsData` variable has a consistent state if it is normalized (ie. if a call to its `normalize` method returns a value below the normalization threshold).

All the methods discussed so far imply a pure read-only access to the datum and do not require any knowledge of its concrete type. They thus realize the objective of generality of treatment of all AOCS data.

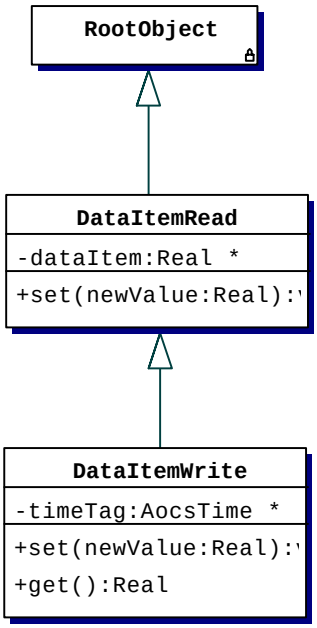
The `getDataItem` methods by contrast are used for the computational access to the AOCS data which is discussed in the next sub-sections.

13.6 Computational Access to AOCS Data

Concrete AOCS data are retrieved from data pools as references to types `AocsData`. However, as mentioned above, computations cannot be performed directly on this type. Instead, components in the AOCS framework that need to perform operations on each other's data will have to do so at the level of atomic variables of primitive type.

The term *data item* is used to designate one of the elements of primitive type that make up an AOCS datum. Unlike AOCS data, data items cannot be further decomposed into lower level entities. A variable of class `ThreeVector`, for instance, contains three data items representing the three elements of the 3-dimensional vector.

Data items are normally private class attributes that cannot be directly accessed from the outside. Access to them must therefore take place through wrapper objects. Two types of wrappers are defined to provide read-only and read-write access, respectively. The two wrappers are represented by classes `DataItemRead` and `DataItemWrite` shown in the UML diagram below:



Attribute `dataItem` is the reference to the encapsulated data item. A call to method `get` causes the value of the data item to be returned. A call to method `set` (exposed by the `DataItemWrite` class only) can be used to change the data item value. Note that it is assumed that the data item is of basic type `Real` and that to each data item a time tag can be associated defining the time when the data item was last updated.

`DataItemRead` objects can be used to establish links between producers and consumers of data. Suppose for instance that component `A` uses a data item `d` from component `B` as an input for its operations. Component `B` will then expose a method to allow `A` to get a `DataItemRead` object that encapsulates `d`. `A` will then use this `DataItemRead` object to access `d`.

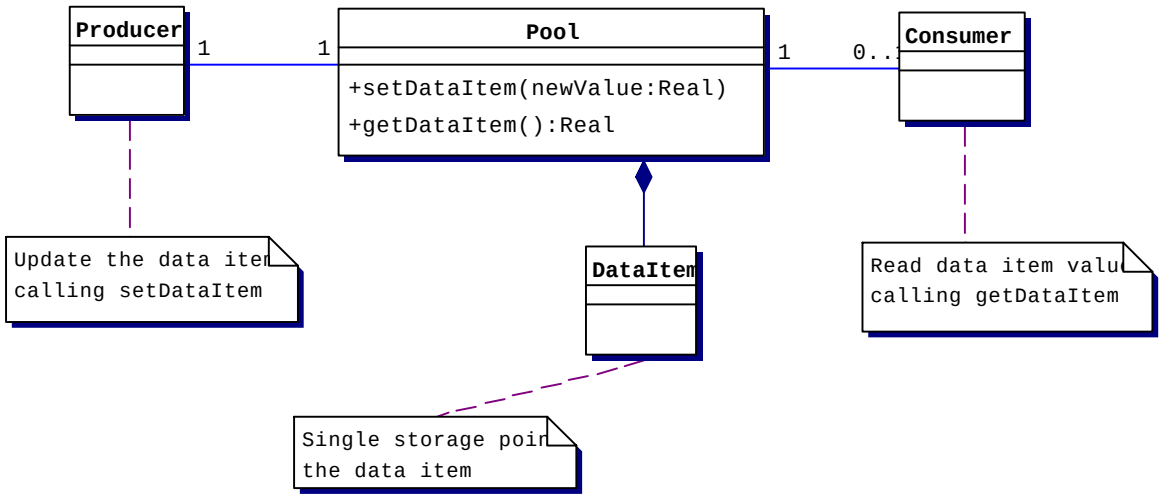
Class `DataItemRead` is designed to be very light-weight because instances of this class are extensively used in the AOCS framework

`DataItemWrite` objects extend `DataItemRead` objects to give write access to the data item they encapsulate.

This class adds to the `DataItemRead` class: a reference to the time tag of the data item and methods to set the variable and to read its time tag. The time tag is automatically set by the `set` method.

13.7 The Shared Data Design Pattern

This design pattern is introduced to address the problem of allowing components to share access to data that are generated synchronously. The pattern is illustrated in the following UML diagram:



The data pool component contains a single instance of the shared datum. The datum producer (of which there should be only one: the datum owner) calls the setter method to set the datum parameters. The datum consumers can use the getter methods to retrieve the attributes of the shared datum and, if necessary, to reconstruct it internally.

Note that data pools could in principle be either *active* or *passive*. In active data pools, clients register their interest in certain data items and are notified when the data item is updated. This design pattern prescribes a passive concept to have maximum decoupling between data producers and data consumers.

13.8 Data Pools

Data pools are based on the shared data design pattern. They are the shared memory areas through which AOCS data are exchanged among framework components.

The data pools physically contain the data items. Data users access the data in the pool through references.

Only one instance of a data of a certain type can be stored in the pool. This means that if, for instance, the method to set the control torque in the attitude data pool is called twice, then the datum written to the pool at the second call overwrites the datum written at the first call.

This mechanism is rigid but safe. It is suitable for cyclical data whose number and type is fixed and can be determined at design time.

Note that data pools introduce some overhead because data have to be *copied* to a data pool.

Several classes of data pools may be present where every data pool class groups together logically related data. The framework thus provides an interface `DataPool` from which application specific data pools can be instantiated. This is the *data pool subclass* hot-spot.

13.9 Alternative Implementation for Aocs Data

The data item interface in `AocsData` may seem superfluous. In an alternative design, one could endow `AocsData` with the following methods:

```
void set(int j, Real newValue);
Real get(int j);
```

These methods could be used to set and get the j-th component of the datum. However, in order to link components to locations in the data pools, it is necessary to have references to them.

In an another alternative design, one might then think to provide class `AocsData` with the following method:

```
Real* get(int j);
```

This method would return the reference to the j-th element and could be used to read and write the element. Now linking to components that use the element is possible but no access control is possible: any component can set the value of any AOCS data element. The data item mechanism instead discriminates between read-only and read-write access.

Finally, the type `AocsData` is a composite type that gathers together the individual elements of the concrete types it represents. These individual elements may have individual properties. For instance, each element may have its own time tag. For this reason, too, is encapsulation of individual data items useful: it makes access to the item's properties easy and natural.

13.10 Reusability and Extensibility Issues

This framelet enhances reusability in three ways. Firstly, through the use of shareable data areas for inter-component communications, it decouples the *production* of data and events from their *consumption*. Secondly, by allowing uniform treatment of all data types, it makes

component interfaces independent of the type of data they process. Thirdly, with the *data item* concept, it provides a way to link components together at run-time.

Ease of extensibility is guaranteed by the possibility of creating new subclasses of `AocsData` and `AocsEvent` to encapsulate new types of data or events.

14 SEQUENTIAL DATA PROCESSING CHAINS

AOCS data that are passed through several consecutive stages of processing will be said to go through a *sequential data processing chain.*

As an example of a sequential data processing chain consider the processing of the raw data collected by a sensor:

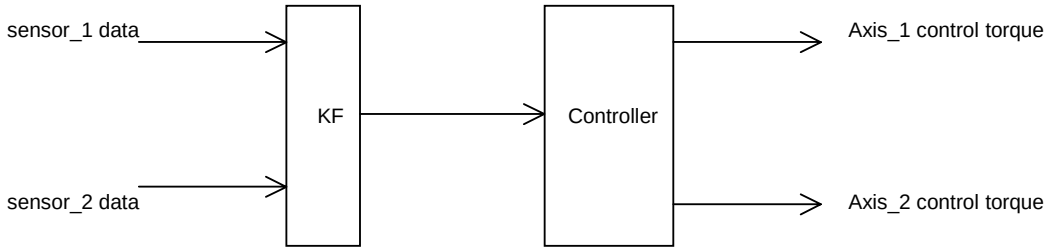
- conversion from raw data to engineering units[2];
- correction for sensor misalignments;
- bias correction;
- transformation from sensor to satellite reference frame

In another example, an attitude controller processes the raw attitude measurement as follows:

- filtering of raw data to remove the measurement noise;
- filtering to cut off frequencies that might excite solar panel oscillations;
- passage through a controller to compute the control torque.

Once again, each bullet can be seen as representing a processing stage in a sequential data processing chain.

In the previous examples, each processing stage was single-input-single-output. A processing chain can include multi-input-multi-output stages as in the case shown in the next figure. The figure shows a 2-axis controller where the sensor measurements are first passed through a Kalman Filter (KF) and are then fed to an attitude controller.



[2] By sensor raw data it is meant the data as they are acquired from the physical bus that links the sensor to the AOCS computer. The engineering unit data are the same data transformed to physical quantities (eg. Volts, degrees, rad / sec, etc.)

14.1 Implementation

In considering the implementation of sequential processing chains, a trade-off must be made between generality and complexity: the most general implementation is also the most complex and the simplest implementation cannot cover all cases of processing chains.

During the framework design process, three implementation concepts for sequential processing chains were considered. The first two – the data converter and the control channel concepts – stand at the two ends of the complexity / generality spectrum. The third one – the formal language concept – takes a more radical approach.

The *Data Converters* described in section 15 are suitable to deal with processing chains that are linear (as opposed to tree-like) and that have a well-defined first or last stage. Their range of application is therefore rather restricted but they can be implemented in a very efficient manner using the *decorator* pattern from RD1.

The *Control Channels* of section 16 are instead based on the composite pattern of RD1 and are more general as they cover the most general case of multi-input-multi-output processing stages linked in any arbitrary manner. As would be expected, their generality is paid for with greater complexity of use and of implementation.

Finally, section 17 offers a third option for the implementation of sequential processing chains. Its approach sees a processing chain as a *sentence in a simple formal language.*

In the AOCS framework, the control channel concept has been adopted and the other two concepts will not be considered further.

15 DATA CONVERTERS

Data converters are a type of *sequential data processing chain* (see section 14). They are suitable to represent processing chains with the following characteristics:

- the processing chain is linear (as opposed to tree-like)
- it is possible to identify a well-defined first or last processing stage

As an example, consider again the treatment of the read-out data from a sun sensor. The raw data from the sensor have to go through the following conversions:

- conversion from raw data to engineering units;
- correction for sensor misalignments;
- bias correction;
- transformation from sensor to satellite reference frame

This conversion chain can be implemented as a data converter because it is clearly linear and it has a well-defined first stage in the conversion from raw data to engineering unit. The latter must *always* be made and must always be the *first* conversion stage. The other conversion stages by contrast need not always be present (eg. sometimes no bias correction is performed) and their order is not fixed (eg. bias correction can be done either before or after misalignment correction).

15.1 Implementation

A data converter represents a sequence of conversion stages. The data at each stage are assumed to be of the same `AocsData` type. Thus, the operation performed at each stage can be conceptualized as:

```
output = convert(input);
```

where both `output` and `input` are of type `AocsData`.

The data converter is implemented using the *decorator pattern* from RD1.

As prescribed in RD1, decorators and decorated objects must share an interface that in the case of AOCS data conversion is naturally defined as follows:

```
class DataConverter {
     AocsData* convert(AocsData* data)=0;
}
```

Decorator objects are derived from the following class:

```
class DataDecorator: public DataConverter {
     DataConverter* component;

     void setComponent(DataConverter* component) {
          this->component = component;
     }

     AocsData convert(AocsData* data) {
          return component->convert(data);
     }
}
```

To each processing stage after the first one in a sequential data processing chain there corresponds one decorator. Passing a datum through the sequential processing chain is then equivalent to subjecting the output of the first processing stage it to successive decorations.

The implementation of the AOCS data converter is described in detail through an example in the next subsection. The terminology is the same as in RD1.

15.2 AOCS Data Conversion Example

This example considers the case of a sun sensor whose raw data must be converted to engineering units, corrected for bias, and finally transformed to satellite reference frame. The UML diagram for the example is at the end of this subsection.

First, a concrete component must be defined. This component represents the first stage of the conversion chain. In the case of the example it is an instance of the following class:

```
class SunSensorDataConverter: public DataConverter {

     AocsData* convert(AocsData* rawData) {
          . . . // convert the sun sensor raw data rawData to eng. units
          . . . // and return the converted value
     }
}
```

Objects of this class perform the basic conversion of sun sensor data from raw data to engineering unit data. Decorators are then required to perform conversions to remove bias and to transform the sensor output to the satellite reference frame. The decorator for the bias removal will be an instance of the following class:

```
class BiasDecorator: public DataDecorator {

     AocsData* convert(AocsData* data) {
          data = DataDecorator::convert(data)
          . . .  // remove bias from ‘data’ and return the result
     }
}
```

The class for the decorator responsible for the coordinate transformation is defined in a similar manner:

```
class CoordinateDecorator: public DataDecorator {

     AocsData* convert(AocsData* data) {
          data = DataDecorator::convert(data)
          . . . // tranform ‘data’ to the satellite reference frame
          . . . // and return the result
     }
}
```

The implementation of the decorator is now as follows:

```
SunSensorDataConverter  aSunSensorDataConverter;
CoordinateDecorator     aCoordinateDecorator;
BiasDecorator           aBiasDecorator;

aBiasDecorator.setComponent(&aSunSensorDataConverter);
aCoordinateDecorator.setComponent(&aBiasDecorator);

. . .

aCoordinateDecorator.convert(data);
```

The first three statements create the sun sensor data converter and its two decorators. The two statements in the middle set up the links between the three objects. The statement at the bottom will cause the three conversions (from raw to engineering data, bias removal, coordinate transformation) to be performed in sequence. Note that `aCoordinateDecorator` is itself a `DataConverter` object can therefore be used wherever AOCS data converters are used.

15.3 Alternative Implementation

As already mentioned, data converters are suitable to represent sequential processing chains that are linear and that have a well-defined first or last stage.

If the conversion chain is not linear, then the control channel pattern described in section 16 should be considered.

The need for a well-defined 'first stage' in the data conversion chain arises from the requirement to have (using the terminology from RD1) a 'concreteComponent' class. In the example of the sun sensor processing chain, for instance, the converter from raw to engineering format fulfills this role because this conversion must *always* take place and must always be the *first* conversion stage.

The implementation proposed here could be easily modified for the case where the 'concreteComponent' is the last stage in the conversion chain.

15.4 Reusability and Extensibility Issues

The architecture proposed here for data conversion furthers software reusability because it provides a standard interface for components that must perform conversions on AOCS data. The containing component can therefore be made independent of the particular conversion that it implements.

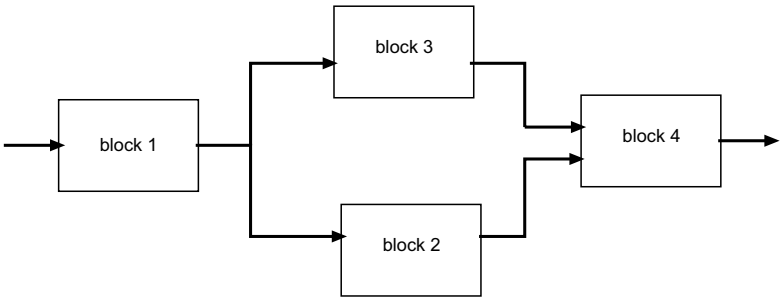
Ease of extensibility is also promoted because changing the converter objects or adding new conversion stages can be done transparently to the code that uses the output data from the converter itself. In the sun sensor example of section 15.2, for instance, it would be possible to eliminate the bias correction stage or to add a misalignment correction stage without having to modify the component that manages the conversion process. Only the code where the converter objects are instantiated at initialization time would be affected.

16 CONTROL CHANNELS

Control channels are the most general type of sequential processing chains. They can represent any sequential multi-input-multi-output processing chain and allow concatenation and nesting of individual processing blocks.

A sequential processing chain is made up of inter-connected processing blocks. More specifically, the term *control channel block* (or simply *block*) will be used to designate a processing block that cannot be further decomposed into lower level processing blocks.

Blocks can be concatenated in chains as shown in the figure:

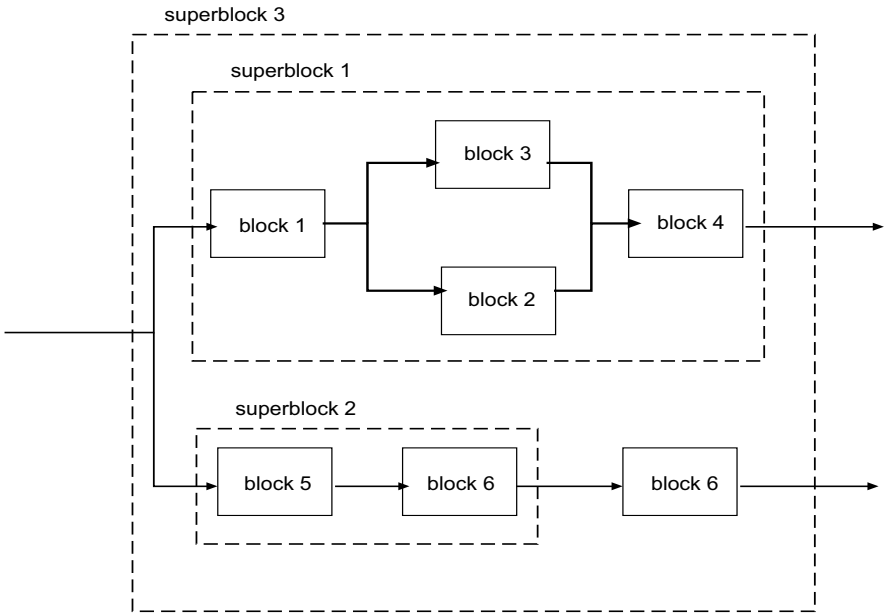


Arrows represent data flow. User input is fed to block 1. User output is taken from block 4.

Blocks chains can be nested within higher-level blocks called *superblocks*. Superblocks and blocks can be freely mixed in processing chains as in the example in the figure in the next page.

The control chain is enclosed in a superblock (superblock 3) with one input and two outputs. The superblocks contains both two superblocks (superblocks 1 and 2) and one simple block.

Note that the terminology of blocks and superblocks is the same as in Xmath. Blocks and superblocks as defined here map to the homonymous concepts in Xmath.



16.1 Data Propagation through Control Channels

In general, a control channel implements a transfer function of the following kind:

$$x_{t+\Delta t} = f(x_t, u_t)$$
$$y_t = g(x_t, u_t)$$

where the usual notation is adopted with u representing the input vector, y the output vector and x the state vector.

The fundamental operation to be performed on a control channel is the propagation of its output signal from time t to time (t+Δt). A *propagate(t)* operation is defined on control channel objects that causes their outputs to be propagated up to time t.

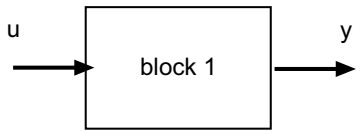
The time t to which the output values are propagated is called the *validity time* of the outputs.

In order to compute y(t), the control channel needs to know u(t). Thus, if control channels are arranged in a sequential chain, a propagation request must be passed to upstream blocks.

For example, consider again the control channel chain of the first figure in the previous section. Suppose `propagate(t)` is called on block 4. Before this operation can be performed on this block, it is necessary to update the inputs to the block and this is done by calling the same operation `propagate(t)` on blocks 2 and 3. Thus, propagate requests percolate backward along the processing chain.

Eventually, they will reach a control channel that takes its inputs from an object that is not itself a control channel. Such external signals will be assumed to be fed to a control channel using a *zero-order hold*. This means that their value will be assumed to be constant across propagation instants.

To illustrate the zero-order hold concept, consider for instance the situation in the figure:



Suppose that both input and output have the same validity time t. Suppose now that operation `propagate(t+Δt)` is called on the control channel. The control channel will responds by updating its output to y(t+Δt) and it will do so by assuming that the input remains constant and equal to u(t) throughout the propagation interval.

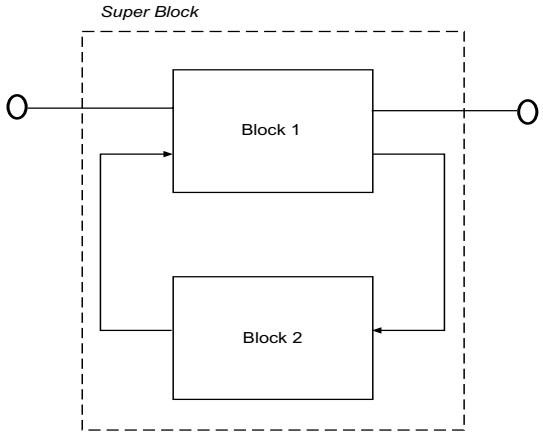
If a first-order hold were used, then the value of the input signal would be computed by linear extrapolation from its last two known values. Higher order holding systems are also possible. Zero-order holding is, however, by far the most commonly used type of holding mechanism in satellite control systems and is the only one for which the AOCS framework makes provisions.

The `propagate` method represents one of the framework hot-spot (the *control block* hot-spot) because each AOCS application must provide its own definition of the transfer functions it needs.

16.2 Signal Loops

The mechanism outlined in the previous section cannot cope with signal loops.

Consider the situation shown in the figure:



When the super block receives a `propagate` request, it will route it to block 1. Before executing the `propagate` action, block 1 will try to update its inputs. It will do so by issuing a `propagate` request to block 2. This will in turn try to update *its* inputs and will do so by issuing a `propagate` request to block 1. An endless loop will result.

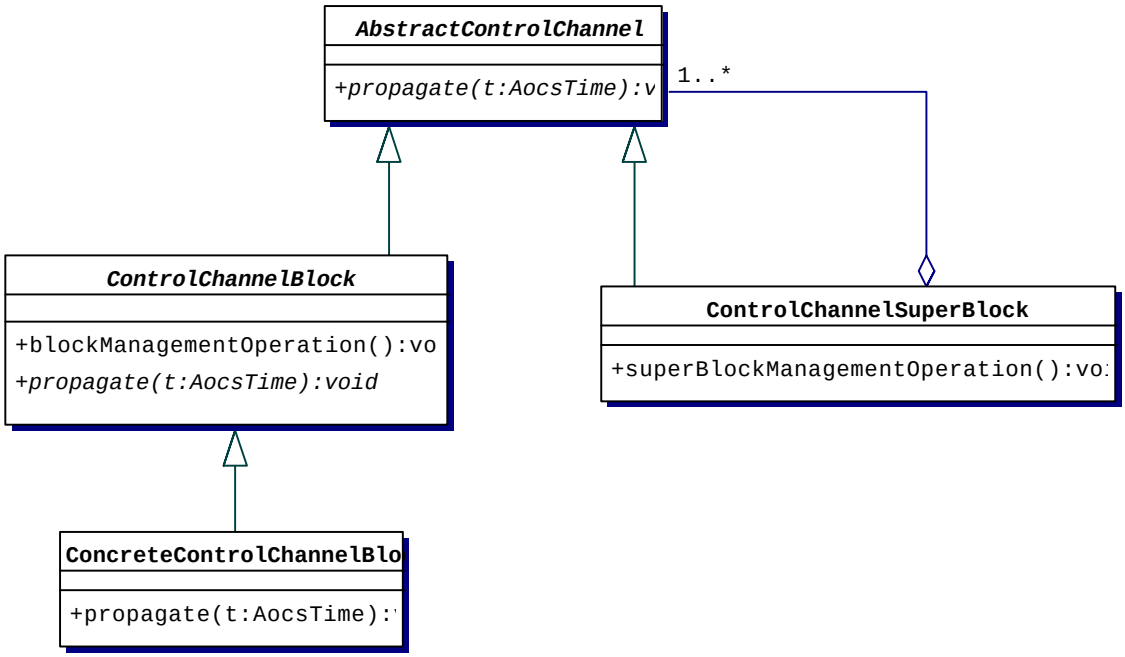
The example shows that signal loops in the control channel connections, either at block or super block level, will result in an endless loop.

The propagation mechanism could be modified to handle loops (at the cost of some overhead) but this is judged unnecessary as signal loops should not arise in an on-board control system.

Note that there is no loop detection mechanism. Responsibility for detecting loops rests with the developer who makes the control channel connections.

16.3 The Control Channel Design Pattern

This design pattern is introduced to model the control channel concept and in particular the distinction between blocks and super blocks. The control channel design pattern is obtained by instantiating the composite pattern and it is shown in the following UML diagram:



`AbstractControlChannel` is a pure interface that exposes the methods that are common to all control channels, regardless of whether they are blocks or superblocks. Control channels are always seen by their clients as instances of this class. Since both blocks and superblocks are derived from `AbstractControlChannel`, they can be treated in a uniform manner as instances of abstract control channels. The fundamental operation exposed by `AbstractControlChannel` is `propagate`(t) that propagates the input signals to the output up to time t as discussed in a previous section. This method gives rise to the control channel hot-spot through which application developers must define the transfer function to be implemented by the control channels they use.

Abstract control channels can be implemented either as control blocks or as control super blocks. `ControlChannelBlock` is an abstract class that acts as base class for all concrete control blocks. It provides concrete implementations of methods and data structure to manage block operations. This class for instance provides data structures to buffer the input and output signals and operations to reset them. Such data structures and operations are common to all control channel blocks. This class is abstract because it does not provide any implementation for method propagate. The implementation of this method defines the concrete transfer function that is implemented by the control channel. Concrete control blocks specialize `ControlChannelBlock` by providing concrete algorithms for the propagation of

the input signals through the control block. The AOCS Framework provides as default components concrete implementations of this class that implement common transfer functions such as PD blocks, PID controllers, integrators, etc.

Super blocks are represented by instances of class `ControlChannelSuperBlock` which is derived from `AbstractControlChannel`. This class defines a default component that manages a set of interconnected lower-level blocks. The lower-level blocks are seen as instances of `AbstractControlChannel` since they can be either control channel blocks or control channel super blocks.

Since both blocks and superblocks are derived from `AbstractControlChannel`, they can be treated in a uniform manner as instances of control channels.

16.4 Recursion

Calls to method `propagate`() can be recursive since when they are called on a given component A, they have to be propagated backward to all control channels directly or indirectly linked to A's inputs. The maximum depth of the recursion is given by the maximum length of a chain of connected control blocks.

16.5 Input and Output Linking

Control channels return their outputs as data items (see section 13.6), namely they return objects of type DataItemRead that encapsulate a reference to the output itself.

Control channel inputs can be linked either to external variables or to the outputs of other control channels. Input linking is also done through data items. Thus, when the input of a control channel needs to be linked to a particular external variable, it is passed the `DataItemRead` object that encapsulates the desired variable. If instead it needs to be linked to the output of another control channel, it is passed the data item returned by the source control channel to encapsulate its output.

16.6 Interface to Xmath Autocode

Xmath is a widely used simulation and modeling package for dynamic systems. Xmath can be used to set up and simulate the kind of transfer functions that are commonly used in control systems.

Xmath includes an autocode tool that can generate the code implementing the transfer function simulated by an Xmath model. The AOCS framework offers a hook where such code can be plugged in. This allows embedding of Xmath models into framework control channels.

16.7 Reusability and Extensibility Issues

The architecture proposed here for control channels furthers software reusability because it provides a standard interface for components that must perform sequential processing on AOCS data. The containing component can therefore be made independent of the particular processing that it implements.

17 A FORMAL LANGUAGE TO EXPRESS THE AOCS DATA FLOW

The data flow part of the AOCS software essentially transforms sensor outputs into actuator commands passing through data conditioning, data filtering, application of control algorithms, and other processing stages.

Section 14 introduced the concept of *sequential data processing chain* to represent the data flow subsystem of the AOCS. The following two sections, sections 15 and 16, proposed two implementation for the sequential data processing concept: *data converters* and *control channels*.

Both the control channel and the data converter concept model a data flow system as made up of interconnected and nested blocks. All blocks are treated as homogeneous since they are derived from the same base class (`AbstractControlChannel` in the case of control channels and `DataDecorator` in the case of data converters). There is therefore no syntactic restriction to how blocks can be linked to each other or nested within each other.

An interesting alternative is to view a particular data flow system as a *sentence in a simple formal language*. In the AOCS context, the basic elements of the language can be types of blocks like:

- real AOCS sensor
- fictitious AOCS sensor
- real AOCS actuator
- fictitious AOCS actuator
- controller
- filter
- state estimator

These blocks are the ‘words’ of the ‘sentence’. They can be strung together according to a small set of formal rules. Examples of rules (expressed in informal language) could be:

- a sensor must be followed either by a fictitious sensor or by: a filter, or a state estimator, or a controller;
- a controller can only be followed by: a filter, or an actuator, or a fictitious actuator;
- a fictitious actuator can only be followed by another fictitious actuator or by a real actuator.

These and other similar rules could be easily formalized to create a grammar specifying how data flow blocks can be linked together. The *interpreter pattern* of RD1 could then be used to

'evaluate' 'sentences' of the grammar. In the present context, a 'sentence' would be data flow diagram for the AOCS and the process of 'evaluation' would refer to the processing of data that are input at one end of the data flow diagram (the 'sensor end') and result in outputs being produced at the other end of the data flow diagram (the 'actuator end').

Such an approach is very attractive for its generality and ease of extension. It would result in components being made available to designers that can only be linked together in manners that are semantically meaningful.

From an implementation point of view, this approach would probably not be any more complex than the approaches proposed in the previous two sections. Perhaps its only conceptual drawback is that the it blurs the distinction between AOCS units and unit-like objects (the fictitious units in the terminology adopted here) on the one hand, and other types of data flow blocks (controllers, filters, etc). Units are different because they can issue instructions and need to be triggered. It may therefore not be appropriate to treat them like controller or state estimator blocks which are instead pure input-output devices.

Because of its potential complexity, the option of treating the data flow part of the AOCS software as a formal language was not further considered in the AOCS prototype framework but may be taken up again in future versions of the framework.

18 AOCS UNITS MANAGEMENT

The *AOCS units* are the equipment that is external to the AOCS computer but is part of the AOCS subsystem. They are represented within the AOCS software by proxy objects that encapsulate the interactions with the hardware. The AOCS framework proposes a generic interface that must be implemented by all unit proxies. This interface acts as an adapter (in the sense of the adapter design pattern of RD1) between the AOCS software and each AOCS unit.

AOCS units can potentially be very different ranging from very simple sun presence detector to very complex devices like GPS receivers or autonomous star trackers. Providing a standard interface that covers the entire range of AOCS equipment is probably impossible. In accordance with the principle laid down in section 3.3, the concept proposed here for unit management is targeted at units of the kind in use at present. The assumed unit model is described in the next subsection.

Adapting the AOCS framework to more “intelligent” units will require developing a concept for active units. It is envisaged that the solution lies in designing objects that can encapsulate a data exchange (similar in concept to the objects that encapsulate telecommands, see section 26).

18.1 Abstract Unit Model

The AOCS framework assumes that units are *passive*, namely incapable of initiating data exchanges with the AOCS computer. All data transactions with a unit occur in response to a command from the AOCS computer.

Data transactions between AOCS computer and AOCS units fall under two categories:

- *Functional Transactions*

  This type of transaction is cyclical and relates to the primary function for which a unit was designed. Thus, for instance, a thruster unit periodically receives firing profiles, a gyro periodically supplies rate information, a reaction wheel periodically receives torque requests.

- *Housekeeping Transactions*

  This type of transaction may be occasional and it consists of commands sent by the AOCS computer to the unit (eg. to change its operational mode, to switch the unit on or off, etc) and of housekeeping data sent by the unit to the AOCS computer (eg. temperature readings, results of self-tests)

The unit status is defined by:

- *Power status*

  Units can either be “power on” or “powered off”

- *Operational Mode*

  Units may be in one of several operational modes

- *Health Status*

  Units can either be “healthy” or “unhealthy”

Finally, units may be able to perform a *self-test*. The result of a self-test is an integer code.

18.2 Unit Data Formats

Data in the AOCS units can exist at two levels of abstraction:

- the *raw data level* representing the data as they are transmitted on the physical communication link between the unit and the AOCS computer;
- the *AOCS data level* representing the data as they are used by clients of the AOCS unit objects in the AOCS software. Such data are expressed in engineering units and, if they depend on reference frame, are expressed in the spacecraft reference frame.

The raw data level is mission-specific and cannot be defined here. This data level, however, is internal to the AOCS unit objects. All exchanges between the AOCS unit objects and other AOCS software objects take place at the AOCS data level. It is the responsibility of the AOCS unit object to perform conversions between raw and AOCS data levels.

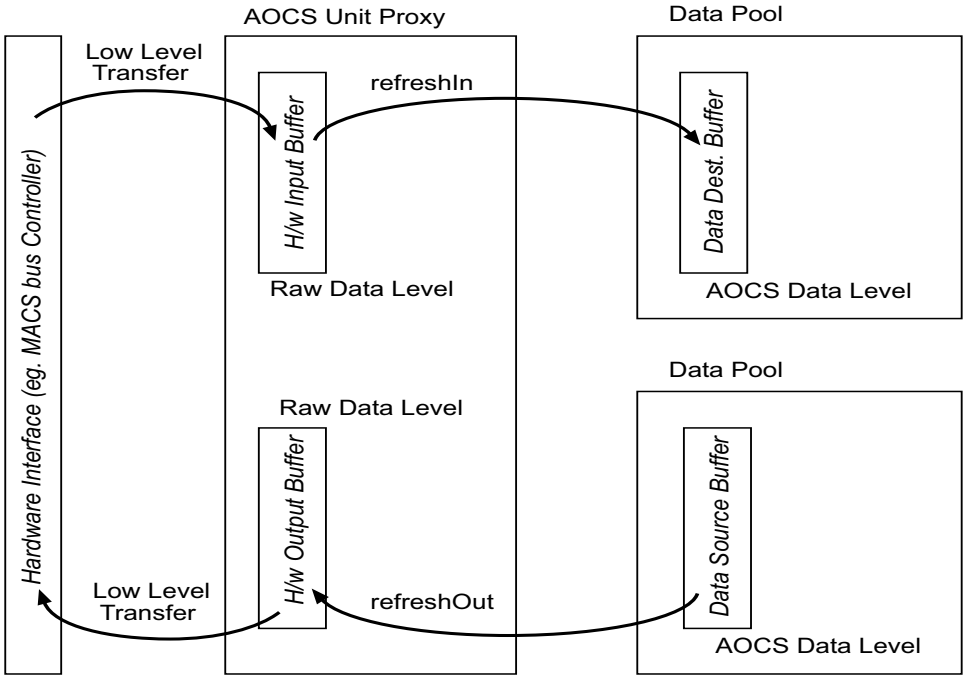
18.3 Data Exchange Model

The model for the exchange of data between hardware and AOCS unit objects is illustrated in the figure in the next page. The figure assumes the case of a unit that can both receive and send data (this could for instance be the case of a reaction wheel that receives torque requests and sends wheel velocity measurements). Most units will only work in receive or send mode.

As shown in the figure, the AOCS unit object maintains *hardware buffers* where incoming or outgoing data are deposited at raw data format. The transfer between these buffers and the hardware interface is done by low level mechanisms.

In a typical example. The hardware interface could be a MACS bus controller. Data reception triggers an interrupt whose servicing routine deposits the incoming data word in the hardware input buffer where it remains available for further processing by the AOCS unit.

Data to be sent to the unit must instead be written to an I/O address. In that case, the hardware output buffer serves as the source for the data written to the I/O port by a dedicated routine.



Only single hardware buffers are shown in the figure. In reality, the buffers may be made up of several memory locations holding related data. The hardware output buffer for a thruster, for instance, will consist of at least two locations holding the firing duration and firing delay data.

The AOCS unit object maintains links to data pool locations where the incoming and outgoing data are stored as variables at AOCS data level. The location where incoming data are stored is called *destination data buffer* and the location from which outgoing data are retrieved is called *source data buffer*. The destination and source data buffers are sets of data items in the data pools.

In the case of a sun sensor, for instance, the destination data buffer is made up of two (or, depending on the representation, three) data items representing the sun vector as measured

by the sun sensor but expressed in engineering units and referred to the spacecraft coordinate frame.

The translation between raw and AOCS data level, and hence the transfers between hardware and destination/source buffers, is done by operations `refreshIn` and `refreshOut`. Operation `refreshIn` refreshes the content of the destination data buffer updating it with the latest data received from the unit. Operation `refreshOut` refreshes the hardware output buffer ensuring that it contains the most up-to-date datum to be sent to the unit.

Operations `refreshIn` and `refreshOut` simply update the content of certain buffers. They do not initiate data transfers. Dedicated operations, `dataAcquire` and `dataSend`, are provided for this purpose. Thus, for instance, a data acquisition cycle would proceed as follows:

- Initiate the bus transaction to acquire the data from the physical unit
- Check that the acquisition process is finished
- Call `refreshIn` to convert the data from raw to AOCS data and to transfer them to a data pool.

At the end of this three-step process, the newly acquired data are located in a data pool in AOCS format and referred to the satellite reference frame. From the data pool, they are accessible to all other objects in the AOCS software.

18.4 Unit Error Handling

Errors arising while interacting with the hardware interface to physical units are reported as events that are stored in the appropriate event repository.

18.5 AOCS Unit Objects

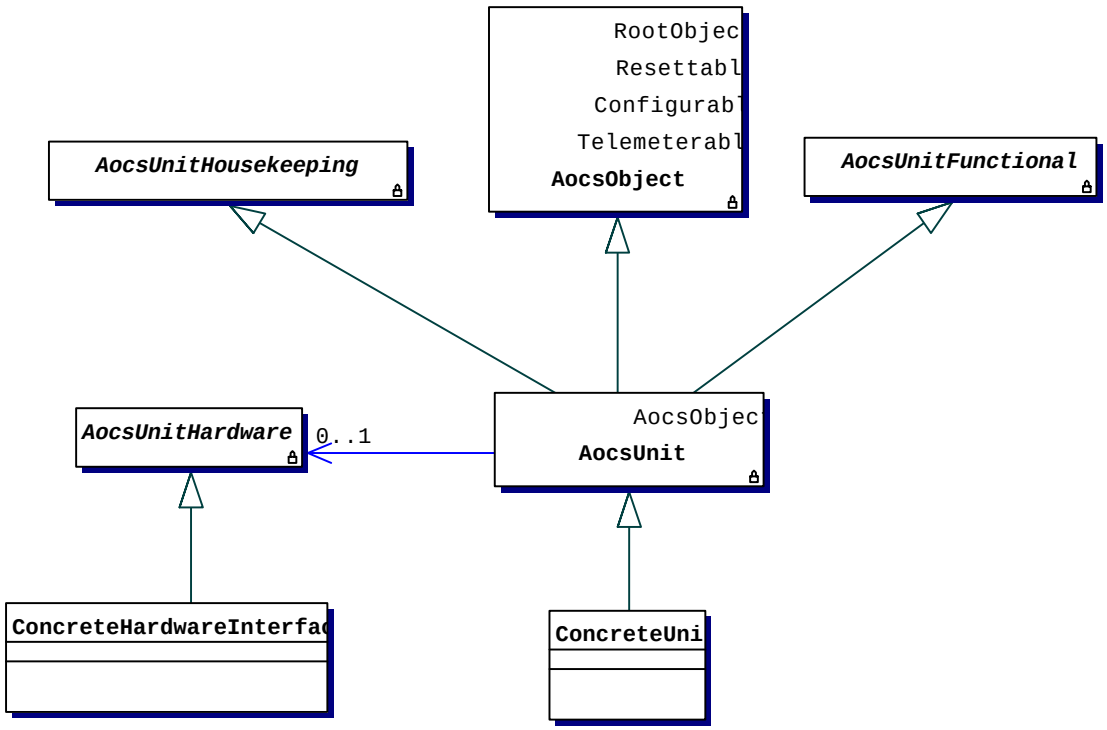
AOCS unit management is based on the *adapter pattern* of RD1. The framework introduces an abstract base class – `AocsUnit` – that defines a generic interface for exchanges with concrete units. This class in turns implements two interfaces: `AocsUnitFunctional` and `AocsUnitHousekeeping`. The two interfaces represent two conceptually different types of access to the unit.

Interface `AocsUnitHousekeeping` represents the commanding interface of the AOCS unit. Through it, it is possible to perform operations like resetting the unit, checking its health status, getting its bus address and performing other housekeeping services.

An AOCS unit is seen by the rest of the AOCS software as a server offering certain services (eg. a gyro is an object that can be queried for rate information). Interface `AocsUnitFunctional` models this high-level functional interface.

The two interface `AocsUnitFunctional` and `AocsUnitHousekeeping` could in principle be merged together. They are kept separate to allow the definition of *fictitious unit*, namely objects that, though not true AOCS units, mimic their functional behaviour. Obviously, both `AocsUnitHousekeeping` and `AocsUnitFunctional` only capture behaviour that is common to all AOCS units.

Concrete AOCS unit classes are derived from `AocsUnit` through class inheritance. The overall class diagram for AOCS units is:

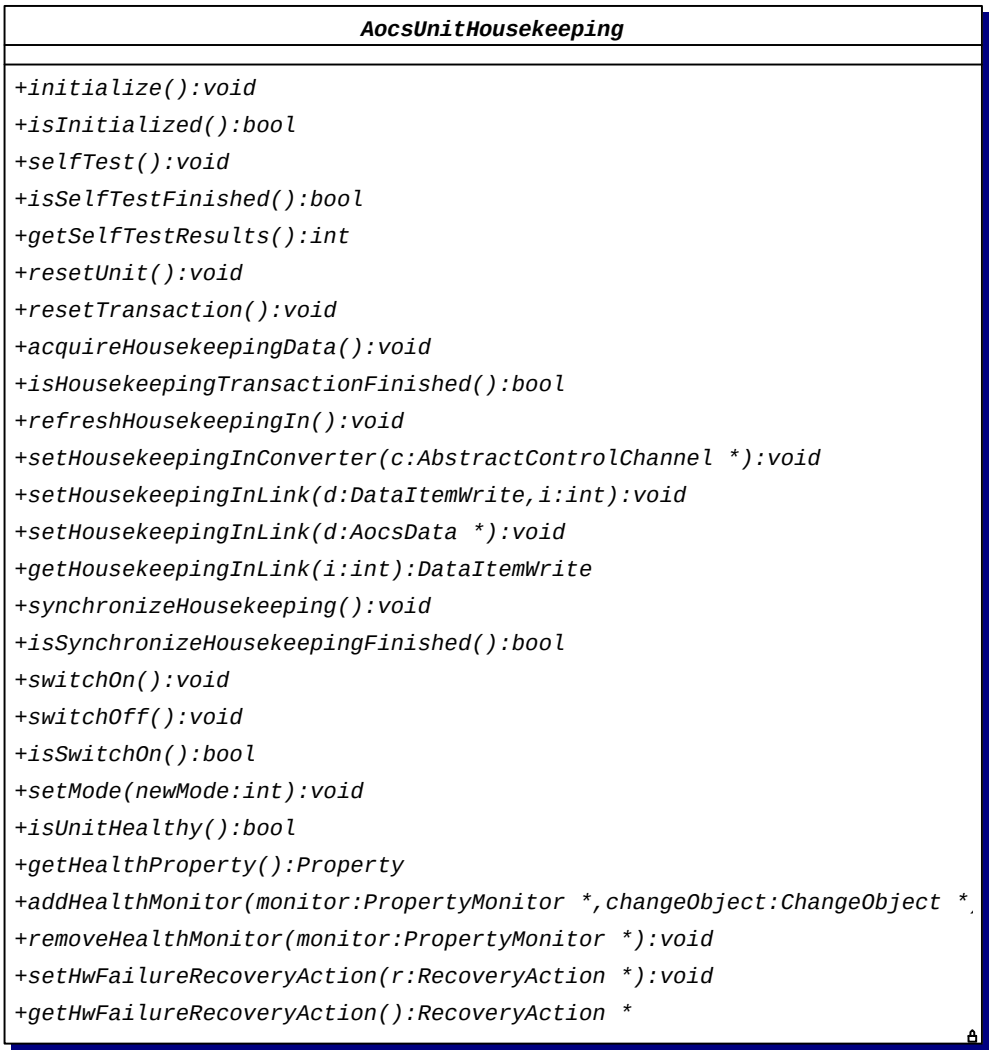


Concrete unit objects delegate low level operations to objects of class `AocsUnitHardware`. `AocsUnitHardware` objects encapsulate the direct exchanges with the hardware. They are normally not visible outside the framelet. Several AOCS units may share access to the same AOCS unit hardware object. Like most non-trivial objects in the AOCS software, unit objects are ultimately derived from class `AocsObject`.

The definition of unit-specific functionalities by subclassing `AocsUnit` represents a framework hot-spot (*AOCS Unit Hot-Spot*) as does the definition of the hardware interface with the units by implementing interface AocsUnitHardware (*AOCS Hardware Unit Hot-Spot*).

18.6 The `AocsUnitHousekeeping` Interface

The `AocsUnitHousekeeping` interface is defined as follows:



AOCS units have an address (the address of the unit on the subsystem bus or equivalent) and a health status that can be retrieved by calling methods `getAddress` and `isHealthy`. The

health status is a bound property that can be subjected to monitoring through change notification (see section 12.2).

Units can be reset by calling method `resetUnit` (note that this is different from the generic reset method that applies to all AOCS objects: the latter resets the software object, the former issues commands to reset an external unit).

Units can be powered up and down by calling methods `switchOn` and `switchOff`. Their power status can be checked through method `isSwitchedOn`.

Units can have different operational modes. Method `setMode` allows the operational mode to be changed.

Units often have to perform some kind of initialization procedure after being switched on. This is done by calling method `initialize`. The initialization procedure may involve bus transactions that take some time. The method is non-blocking and the completion status of the initialization procedure can be checked by calling method `isInitialized`.

Some units can perform self-tests. A self-test is initiated by calling method `selfTest`. This is a non-blocking call and the self-test result is obtained (at an unspecified later time) with method `getSelfTestResult`.

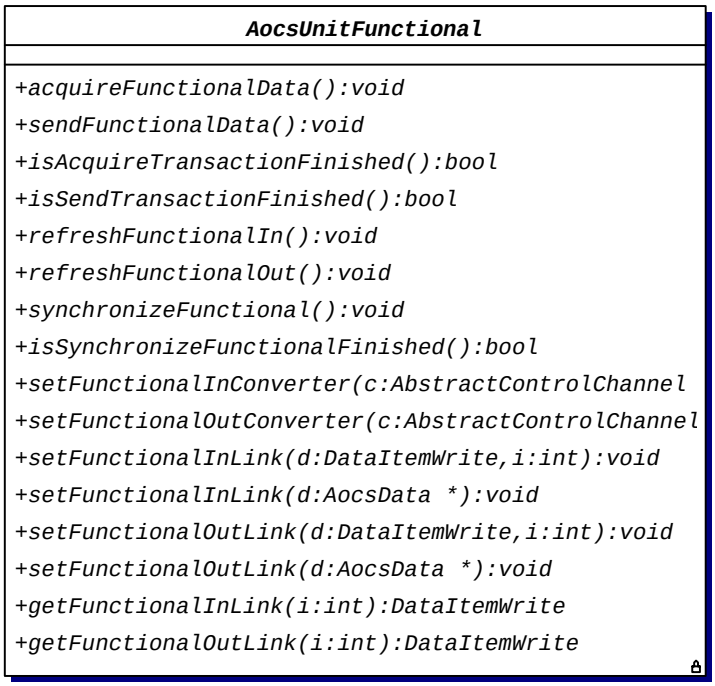
Housekeeping data are acquired by calling `acquireHousekeepingData`. A call to this method initiates the process for the acquisition of housekeeping data from the unit. This method is non-blocking and the completion status of the acquisition can be checked by calling `isHousekeepingAcquisitionFinished`.

Housekeeping data are acquired at raw data level and placed in a hardware buffer internal to the AOCS unit. Their conversion to AOCS data level and transfer to a data pool is done by calling `refreshHousekeepingOut`. The conversion, and any other processing on the raw data (eg. bias correction), is done by a control channel object that can be set with method `setConverter`. The link to the data pool locations is through data item that are set with method `setHousekeepingInLink`.

Thus, the housekeeping part of an AOCS unit can be configured (dynamically, if desired) by setting the control channel that encapsulate the type of processing performed on the raw data acquired from the physical unit and by linking it to the data pool locations where the acquired data are to be deposited after processing.

18.7 The `AocsUnitFunctional` Interface

The `AocsUnitFunctional` interface is defined as follows:



The interface exposes methods to acquire and send data, to process these data, and to link to the data pool locations where incoming data are to be deposited and outgoing data are to be retrieved. The processing is done by control channel objects that can be set as part of the configuration of the AOCS unit object. The discussion of the previous section applies, *mutatis mutandis*, equally well to these methods.

Typical operations that would be performed on incoming or outgoing data include:

- conversion between raw data and engineering units;
- correction for sensor misalignments;
- bias correction;
- transformation between sensor and satellite reference frame

The use of control channels to encapsulate such operations means that the type and sequence of operations can be easily changed.

18.8 Acquiring and Sending Atomic Data

It should be noted that the two interfaces defined above make provisions for only one set of data to be acquired from or sent to the unit because there is only one `acquireData` and `sendData` operation without arguments. Thus, a call to `acquireFunctionalData` is

translated into a set of instructions to acquire *all* the functional data required from the unit in one acquisition cycle.

Consider for instance the case of a sun sensor. Such a unit will normally provide two data words in each acquisition cycle representing the two components of the sun vector as seen by the sensor. The two data words are normally acquired in two distinct instruction cycles. The interface proposed here would not allow the acquisition of only one sun vector component: a call to `acquireFunctionalData` on the sun sensor object will cause *both* data words to be acquired and to be placed in the data pool locations to which the object is linked.

This may seem like a lack of flexibility. In reality, AOCS units in current use are invariably designed to supply or require only one type of attitude information. This information may physically be constituted of several data words but conceptually, it represents a coherent whole and users of the AOCS unit are unlikely to ever need only a subset of this information. The proposed interface, on the other hand, has the advantage of simplifying the management of the unit and is suitable for *fictitious units* (see section 19) thus allowing uniform treatment of a large array of components.

18.9 Unit Triggers

`AocsUnit` objects are passive objects. Data and command exchanges with external units need to be started by some external active object. One solution to this problem is to leave the task of initiating data exchanges to the objects that need the sensor data. Thus for instance, an attitude controller that uses sun sensor data will be responsible for triggering the sun sensor acquisition. This solution may often be acceptable but will sometimes incur the following problems:

- Data exchanges may take time and this may force the data consumer unit to poll the device after initiating an exchange;
- Data exchanges may have to be done at specific points in the AOCS cycle (typically, sensors are sampled at the beginning of the cycle and actuators are commanded towards the end of the cycle) and these times do not necessarily fall within the activation window of the object that consumes or produces the unit data.

An alternative solution is to use dedicated objects to control data exchanges with AOCS units. Such objects will be called `UnitTriggers`.

A `UnitTrigger` is an active object. Its function is simply to initiate transactions with units that need to be activated at the same time. In a typical configuration, two `UnitTrigger` objects can be used. The first one is scheduled very early in the AOCS cycle and is responsible

for triggering data acquisitions from sensors. The second one is scheduled towards the end of the AOCS control cycle and is responsible for triggering transmissions of data to the actuators.

`UnitTrigger` objects may be endowed with *operational mode* (see section 21) if the sequence and type of units they have to be trigger vary in different mission phases.

18.10 Reusability and Extensibility Issues

The architecture proposed here furthers software reusability because it provides a standard interface for AOCS units that decouples the managers and users of unit data from the units themselves. Components can be written in terms of the generic `AocsUnitFunctional` and `AocsUnitHousekeeping` interfaces and can be coupled at run-time with specific units.

Ease of extensibility is provided by the option of subclassing `AocsUnit` to implement specific types of units.

19 FICTITIOUS AOCS UNITS

AOCS units were defined in the previous section as objects that implement two interfaces: `AocsUnitHousekeeping` and `AocsUnitFunctional`. The former interface is used for issuing housekeeping commands to the unit while the latter is used for data exchanges with the unit. If the AOCS unit is seen as a server offering certain services, this latter interface captures the behaviour of AOCS units that is visible to the unit's clients. Clients of the unit would normally regard an AOCS unit object as an instance of `AocsUnitFunctional` rather than as an instance of `AocsUnitHousekeeping`.

AOCS unit objects act as proxies for physical units that exist outside the AOCS computer. Very often, however, a situation arises where a unit-like behaviour is exhibited by objects that do not directly represent any external physical equipment. Consider for instance the acquisition of angular rate information from a gyro package. The physical units are gyros that are sometimes individually addressable and to which there correspond AOCS unit proxies in the AOCS software. The users of rate information (eg. the attitude controllers) will normally interact with a higher-level object that combines the read-out from individual gyros to produce a 3-vector representing the estimated spacecraft angular velocity. From their point of view, this high-level object is a kind of unit on which the same operations can be performed as are normally performed on AOCS unit proper. This type of objects will be called *fictitious AOCS units.*

Another example of fictitious AOCS unit could be an object that combines measurements from a gyro package and from a star sensor and perform filtering on them to produce high accuracy attitude estimates. Users of the attitude estimate would like to see this object as a proxy for some high-accuracy attitude sensor.

The unit reconfiguration managers of section 20 provide still another instance of fictitious units. In this case, the reconfiguration managers takes care of reconfiguring the units it manages according to their health status while at the same time offering to clients of the units the same functional interface that the units themselves would offer.

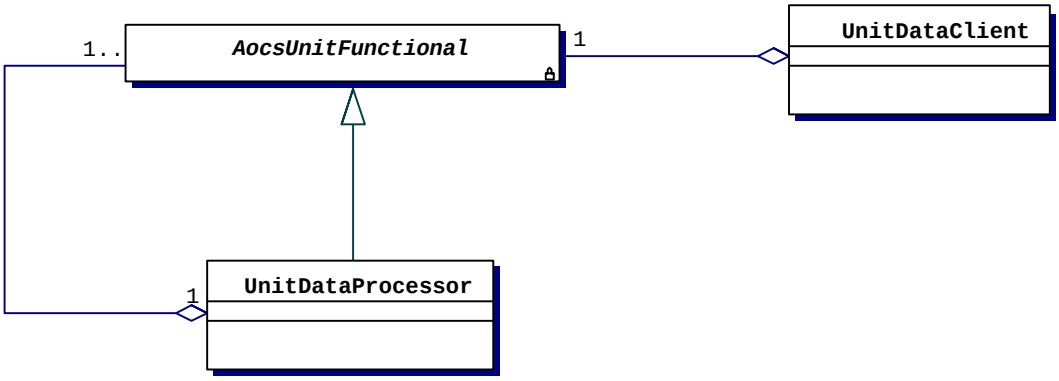
In order to handle all these situations, the framework proposes the fictitious unit design pattern described in the following sub-section.

19.1 The Fictitious Unit Design Pattern

This design pattern is introduced to address the problem of combining components that process unit data without impacting the final users of the unit data.

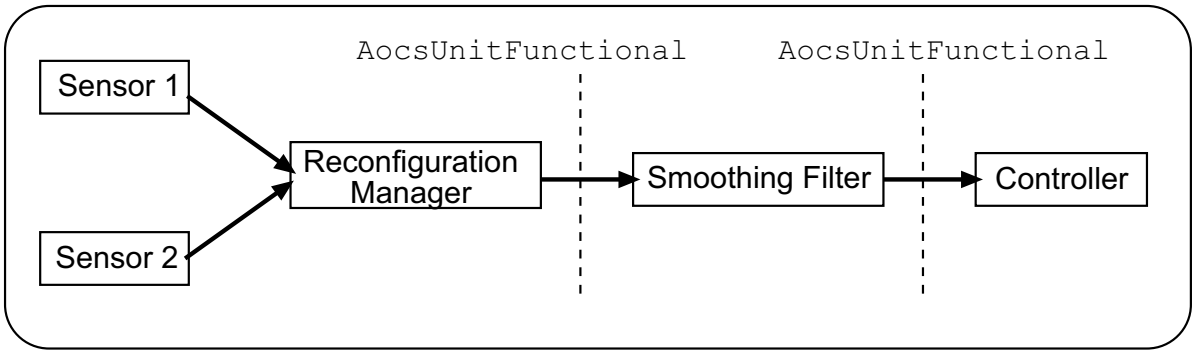
The design pattern is based on the concept of fictitious AOCS unit that is defined as an object that implements the `AocsUnitFunctional` interface. By implementing this interface, an object offers to its clients the same *functional* interface as an AOCS unit.

The fictitious unit pattern is illustrated by the following UML diagram:



The `UnitDataProcessor` is a concrete class that performs some kind of processing on the unit data. It is a fictitious AOCS unit because it implements interface `AocsUnitFunctional`. The unit data processor obtains the unit data from components that it sees as instances of type `AocsUnitFunctional`. These components may either true AOCS units or fictitious AOCS units. Interaction through the `AocsUnitFunctional` interface shields the unit data processor from having to know whether it is interacting with a real or a fictitious unit. `UnitDataClient` is the final user of the unit data. It too sees its source of unit data as an instance of type `AocsUnitFunctional` with the same advantages.

The fictitious data unit concept allows data processors to be easily combined without disrupting client's operation. An example of a combination of unit data processing elements is shown in the figure using informal notation:



The controller is the final user of the sensor data. Its operation is independent of how many filters and other processing elements are interposed between itself and the actual sensors.

The fictitious unit design pattern can also be seen as an instance of the composite pattern of RD1.

The implementation of interface `AocsUnitFunctional` to encapsulate the (application-specific) processing of unit data is one of the framework hot-spot (*fictitious unit hot-spot).*

19.2 Recursion

The fictitious unit design pattern introduces recursion (as does its close relative the composite pattern). Calls to `AocsUnitFunctional` methods may be recursive and the maximum depth of recursion is given by the maximum number of elements that are linked together in a fictitious unit chain.

19.3 Reusability and Extensibility Issues

The abstraction defined in this section makes it possible to decouple the design of the AOCS software from the characteristics of the AOCS units. This is important because the AOCS units and their interfaces are often defined late in an AOCS project.

At an early stage, the software designer can tailor the AOCS software to the characteristics of some convenient fictitious units. At a later stage, when the real units are defined, the fictitious unit will act as a transparent bridge between them and the rest of the AOCS software.

Furthermore, the fictitious unit concept makes it possible to treat components that process unit inputs and outputs as if they were themselves units thus simplifying the software design.

20 RECONFIGURATION MANAGEMENT

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*.

Reconfiguration means switching between different independent implementations of the same functionality.

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.

20.1 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 must be represented by an 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".

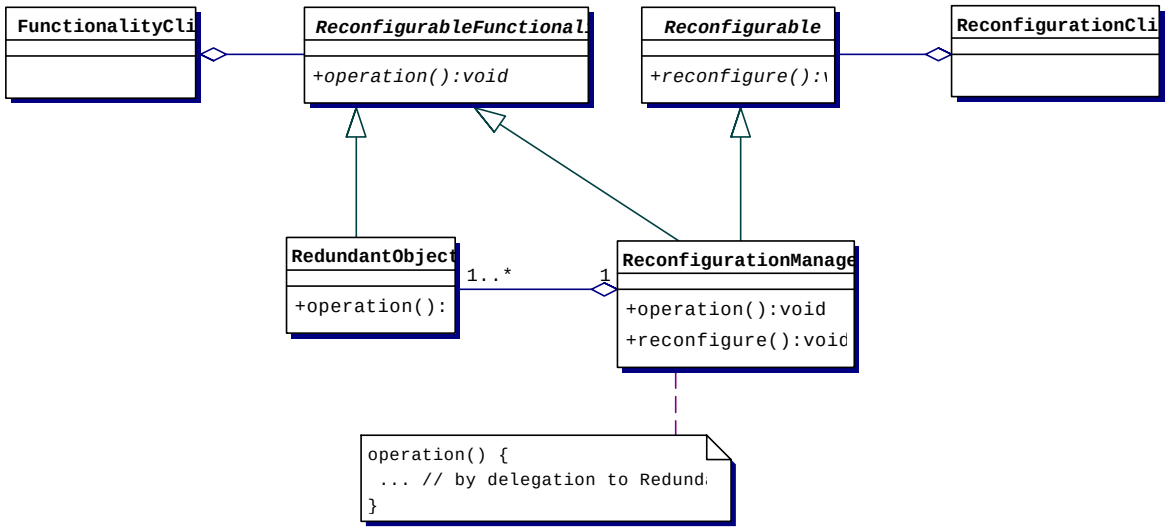
An example of a redundant group is the objects representing two aligned sun sensors (prime and redundant sun sensor). In this case, the functionality is the provision of sun position information. A reconfiguration means switching from one to the other sun sensor. The order of this reconfiguration group is 2. If the two sun sensors have identical characteristics, then the two configurations have the same ranking, otherwise the configuration including the most accurate sun sensor has the higher ranking.

Another example of a redundant group is a set of four non-aligned gyros[3]. The functionality offered by this group is the provision of 3-axis rate information. A reconfiguration means to switch from one set of three gyros to a different set of three gyros. The order of this reconfiguration group is four (there are four sub-sets of three gyros in a set of four gyros). The ranking of the four configurations depends on the characteristics of the gyros and on their geometric disposition.

Still another example of reconfiguration group is represented by a set of the following units: a fine sun sensor, a 3-axis gyro package, and an autonomous star tracker. The functionality offered by this group is the provision of high-accuracy attitude information. High accuracy attitude information can be provided either by combining fine sun sensor and gyro data or directly by an autonomous star tracker. Hence, this group has a reconfiguration order of 2 (in one configuration the FSS and GYR are used, and in the other configuration the AST is used). The ranking of the two configurations depends on the relative accuracy of the sensors and on the quality of the data filtering used in the FSS / GYR configuration.

20.2 The Reconfiguration Management 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:



[3] It is recalled that any three non-aligned gyros can provide full 3-axis rate information.

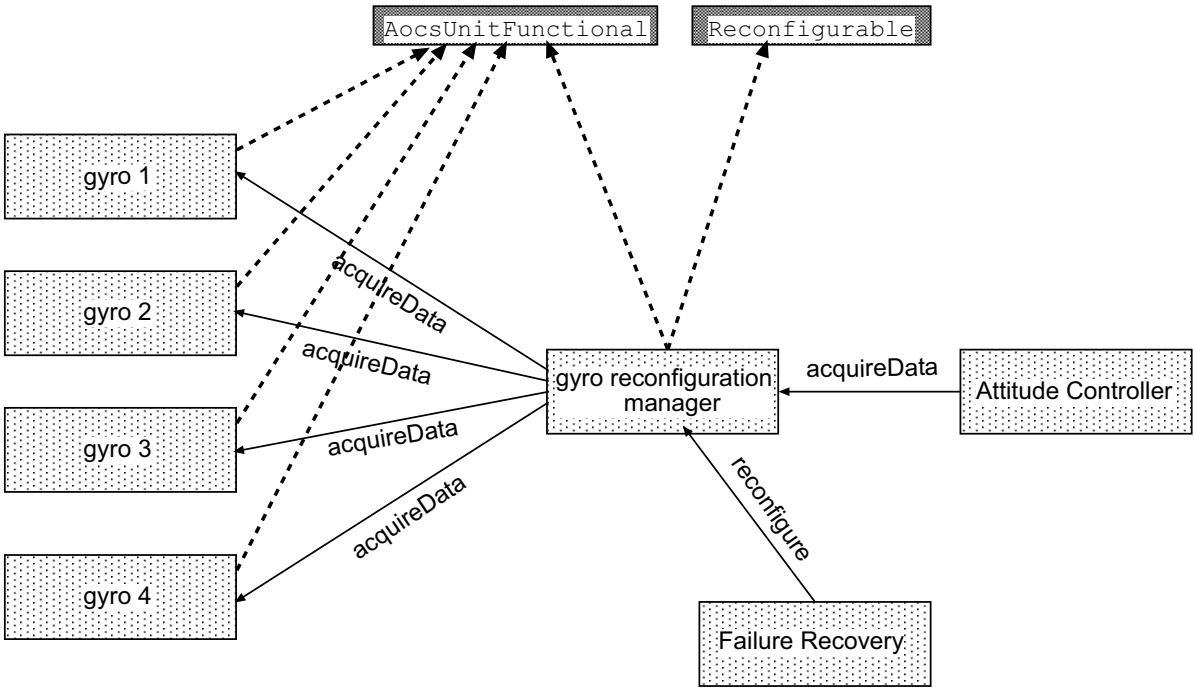
The redundant objects are instantiated from class `ReconfigurableObject`. The reconfigurable functionality they offer is encapsulated in the abstract interface `ReconfigurableFunctionality`. The reconfigurations are managed by *Reconfiguration Manager*. This component is characterized by interface `Reconfigurable` 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 the previous sub-section. The gyros – as AOCS units – would be characterized by interface `AocsFunctional` (see section 18) 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.

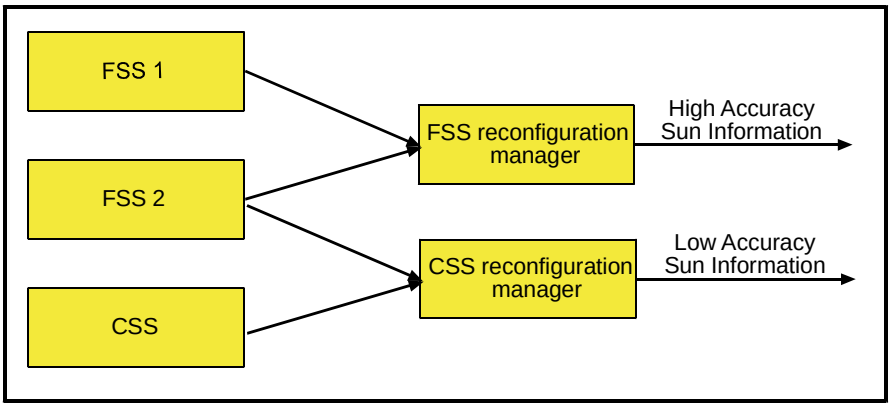
The figure shows that reconfiguration manager has two faces: it has an `AocsUnitFunctional` face that it exposes towards the attitude controller (to which it supplies the rate estimate obtained from merging the rate measurements from the three active gyros) and it has a `Reconfigurable` face that it 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.

20.3 Intersection of Configuration Groups

In a simple implementation, configurations groups are disjoint: if an object belongs to one configuration group, it cannot belong to any other.

Disjunction of configuration groups makes their management simple and efficient but may not always be optimal. Consider a mission scenario where there are two redundant fine sun sensors (FSS) and only one coarse sun sensor (CSS). The FSS's are intended for a high accuracy mode whereas the CSS is intended for a low-accuracy attitude acquisition mode. Since an FSS can also serve as a CSS (possibly with a smaller field of view), it is conceivable that one of the two FSS's should serve as redundant sensor for the CSS. Two reconfiguration groups can be defined, one providing low-accuracy sun information and the other providing high-accuracy sun information. Pictorially, they can be represented as follows:



In this case FSS2 is shared across two reconfiguration groups. Clearly, reconfiguration information should somehow be passed from one group to the next. Suppose for instance that the FSS reconfiguration manager reconfigures from FSS_2 to FSS_1. This probably means that FSS_2 is faulty and hence the CSS reconfiguration manager should be informed of the fact to exclude it from its reconfigurations.

A simple way to organize this exchange of information would be to endow the FSS with a boolean property called `configuredOut`. When one of the configuration managers attached to an FSS performs a reconfiguration that excludes the FSS, then the excluded FSS has its `configuredOut` property set to `true`. `ConfiguredOut` is a bound property and other configuration managers can register to be notified of any changes in its value. In the case of the figure, the CSS, when notified that one FSS has been excluded, could either lower the

ranking the configuration that includes it, or it could mark the configurations that include it as "unhealthy".

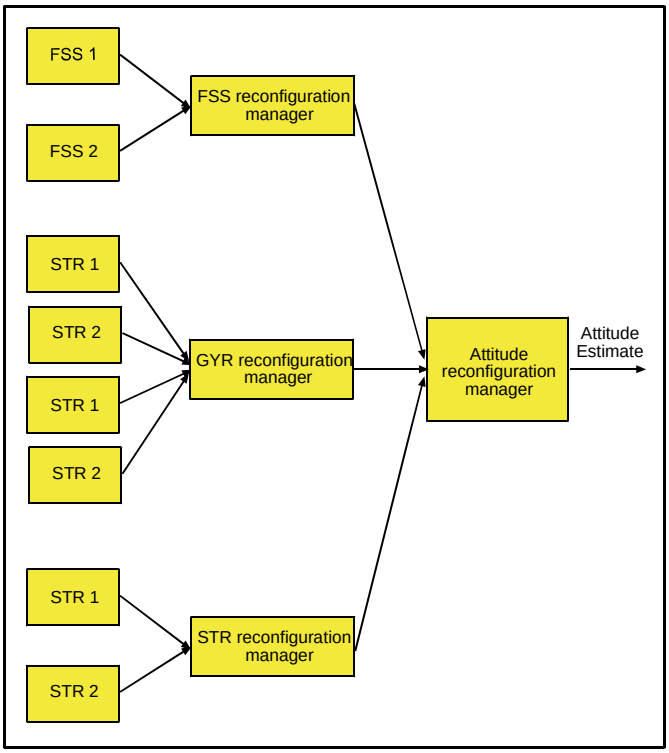
The general mechanism therefore is for objects that straddle configuration groups to carry an observable property `configuredOut`. This property is set to `true` when the object is excluded from a configuration. All configuration groups that include the object register their interest in the property and are notified when it changes. Appropriate action in response to a notification that one object has been excluded from a configuration could be:

- lowering the ranking of configurations including that object;
- marking configurations including that object as "unhealthy".

20.4 Nesting of Reconfiguration Groups

Reconfiguration groups can be nested within each other. Consider again the last example described at the end of section 20.1. Its structure is shown in the figure in the next page.

In this case, the objects reconfigured by the attitude reconfiguration manager are themselves reconfiguration managers. The reconfiguration order of the attitude reconfiguration manager is determined by the reconfiguration orders of the FSS, GYR and STR reconfiguration orders.



20.5 Direct Access to Redundant Objects

Is direct access to redundant objects required? For instance, in the case of four gyros arranged as in the figure in section 0, is direct access to the gyro objects ever required or should access to the gyros always be through the reconfiguration manager?

Two types of access to redundant objects can be envisaged: functional access and housekeeping access. Redundant objects provide services, *functional access* is the access by a client that needs these services. In the case of a gyro, for instance, a functional access is an access to obtain rate estimates. *Housekeeping access* is aimed at performing housekeeping operations on the object (initialization, mode changes, self-test, etc.).

Functional access is exclusively through the reconfiguration manager. For this reason, the reconfiguration manager is made to implement the reconfigurable interface.

Housekeeping access should preferably be done through the reconfiguration interface but can occur directly when needed. The reconfiguration manager may have to be kept informed of housekeeping operations performed on its redundant objects. In this case, the monitoring mechanism described in section 12.2 is used. Thus, for instance, if housekeeping access to a gyro is allowed to change the gyro's health status, then this health status is treated as a property with which the gyro reconfiguration manager can register.

20.6 Preservation of Configuration Data

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.

The storage of configuration information data follows the *memento design pattern* of RD1.

Configuration information is stored in an object of type `ConfigurationState`. This object encapsulate the configuration of a reconfiguration group. In particular, it stores the following information:

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

Responsibility for preserving configuration information in the event of a software or hardware reset lies with the system manager (see section 9.2 and in particular subsection 9.4).

20.7 Reusability and Extensibility Issues

The architecture proposed in this section furthers reusability because it decouples the task of managing unit reconfigurations from the task of processing of unit data. Users of unit data always see the same interface to represent a group of reconfigurable units, regardless of which particular unit is selected. The management of the reconfiguration in turn is independent of who uses unit data and how they are used.

21 OPERATIONAL MODE MANAGEMENT

Current AOCS systems are based on the concept of operational mode. The operational mode is an attribute of the AOCS software as a whole. Its purpose is to adapt the software's behaviour to various sets of external conditions.

As one moves to a component-based approach, the basic design choice is between:

- keeping a single operational mode for the whole AOCS system;
- making operational mode a property of individual components.

The first approach requires a "mode manager" object which is responsible for ensuring that each object behaves in a manner that is consistent with the current mode. This object would act as a centralized coordinator of object behaviour. As such, it would require a rather detailed understanding of how each object is constructed and of what its internal state is. This approach would weaken data encapsulation and is therefore rejected.

The selected approach makes operational mode a property of each component. Components now become responsible for updating their own operational mode in response to changes in the environment around them.

A component keeps track of changes in its environment by monitoring properties of other objects. An object monitors the property of another object by explicitly registering interest in that property and by asking to be notified when the property changes in a certain manner (see section 12.2).

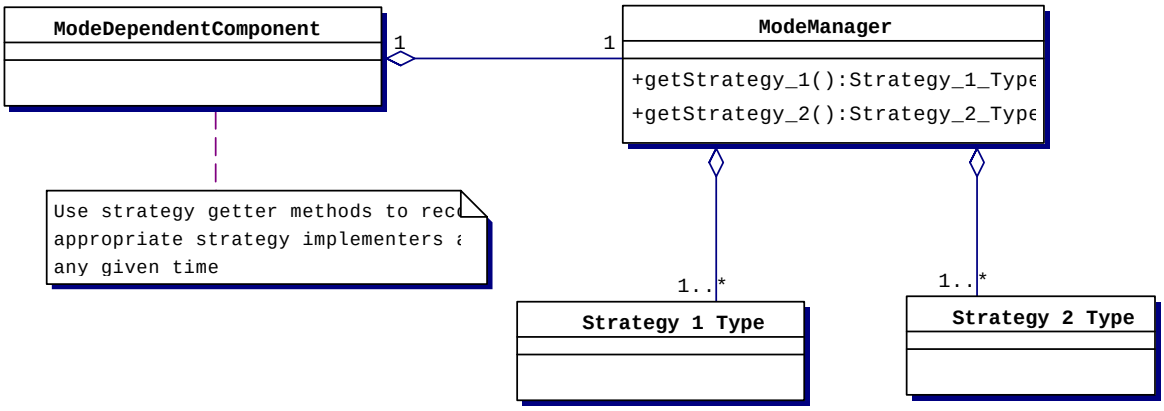
Note that a monitored property could also be the operational mode of another component. Hence, the traditional architecture with an AOCS-wide operational mode could be implemented by having each component implement the same set of modes and by having components change their own mode to follow changes initiated by a "mode manager" object.

As an example of the decentralized mode concept proposed here, consider the attitude controller object. The controller object is responsible for implementing the attitude control law. A controller object could have two modes: low accuracy control, and high accuracy control. The switch between the two modes could be done, for instance, on the basis of the size of the attitude error. The controller object would then register its interest in the attitude error and would ask to be notified when this error crosses a certain threshold. The notification that the attitude error has crossed the selected threshold would trigger a mode change.

21.1 The Mode Management Design Pattern

This design pattern is introduced to address the problem of endowing components with mode-dependent behaviour. The pattern separates the implementation of the mode-dependent behaviour from the implementation of the logic required to decide mode switches.

A mode-dependent component is a component that needs to select a particular *implementation* of one or more *strategies* depending on operational conditions. The figure illustrates the case of a component with two strategies:



The two strategies are characterized by classes `Strategy_1_Type` and `Strategy_2_Type`. These could be concrete classes (with different implementations being represented by different instances), or base abstract classes or abstract interfaces (with different implementations being represented by different subclasses).

The mode manager encapsulates the logic to decide which implementation of each strategy is appropriate at any given time. When the mode dependent component needs a strategy implementation, it uses the strategy getter methods offered by the mode manager to obtain one. From the point of view of the mode dependent component, the strategy implementations are always seen as instances of type `Strategy_1_Type` and `Strategy_2_Type`.

As an example consider again an attitude controller component. This component is responsible for implementing the attitude control algorithm and could have two modes: low accuracy control, and high accuracy control. In low accuracy mode, attitude information is provided by a low-accuracy sensor, for instance a coarse sun sensor (CSS), and is processed by a low-accuracy algorithm, for instance a PID. In high-accuracy mode instead, a high accuracy sensor, for instance a fine sun sensor (FSS), and a high accuracy control algorithm, for instance a PID with Kalman filtering, are used. In this case, the controller component (the

mode-dependent component) has two strategies (the sensor and the control algorithm) and each strategy has two implementations (the CSS and the FSS for the sensor strategy and the PID and PID+KF for the control algorithm strategy). Thus, a skeleton code for an `AttitudeController` component could be as follows:

```
class AttitudeController {

     AttitudeSensor* sensor;
     AttitudeControlAlgorithm* controlAlgorithm;
     AttitudeControllerModeManager modeManager;

     . . .

     void computeControlTorque() {

          // Retrieve strategy implementations
          sensor = modeManager->getAttitudeSensor();
          controlAlgorithm = modeManager->getControlAlgorithm ();

          // Use strategies to compute control torque
          sensor->acquireData();
          controlAlgorithm->processSensorData();
     }
}
```

The mode manager of this example will maintain two versions of the attitude control algorithms and two versions of the attitude sensor and will supply the appropriate one to the attitude controller algorithm based on operational conditions. The controller component can thus concentrate on computing the control torques without having to keep track of changes in operational conditions.

This design pattern gives rise to two framework hot-spots. The *mode manager hot-spot* is used by applications to define the application-dependent mode switching logic. The *mode implementer hot-spot* is used by applications to load the application-specific strategy implementations.

21.2 Mode Change Action

Often, a mode switch by a component must be accompanied by some special actions related to the exit from the old mode and / or the entry into the new one.

Consider for instance an attitude controller component (see section 28) that manages several control algorithms corresponding to different operational mode. When a new mode is entered, its associated control algorithm should be reset to remove the memory of previous activations.

In another example, consider a unit trigger (see section 18.9). To each of its mode a certain set of units is associated. When the unit trigger leaves a mode, it might have to switch off units that are no longer needed in the new mode and, conversely, when it enters the new mode it should turn on and initialize the newly needed units.

Actions that are associated to a mode transition are called *mode change actions*. The AOCS framework offers two mechanisms for their implementations. The most straightforward approach is simply to hard-code the mode change actions into the concrete mode managers. Thus, for instance, in the case of the attitude controller example, when the mode manager finds that it has to perform a change mode, then it executes any actions that are associated to the mode change.

A second, more systematic approach, is to encapsulate the mode change actions into objects that implement the following interface:

```
class ModeChangeAction {
     void doModeEntryAction(int oldMode)=0;
     void doModeExitAction(int newMode)=0;
}
```

A mode change action object is then associated to mode manager and the mode manager calls method `doModeExitAction` when a mode is exited and method `doModeEntryAction` when a mode is entered.

Mode change action objects take the mode as their argument. In an alternative implementation, a mode change action object is associated to each mode managed by a mode manager and, when it leaves a mode, the mode manager calls `doModeExitAction` on the associated mode change action object and, when it enters a mode, it calls its `doModeEntryAction` method.

Mode change actions are a framework hot-spot (the *mode change action* hot-spot) since it is through them that applications define the actions to be taken upon mode transitions.

21.3 Coordination of Operational Mode Changes

The operational mode is a property of (some) components. In order to ensure consistency of behaviour at the system level, it is necessary to ensure that changes in the operational mode of individual objects are coordinated. For instance, it is clearly dangerous if the attitude controller is operating in a mode that assumes that high accuracy sensors are powered up and supplying attitude information while the FDIR object assumes that only basic sensors are switched on.

In order to guarantee coordination, an object's operational mode is designated as *bound property* (see section 12.2). This gives other objects the chance to register their interest in changes in its operational mode.

21.4 AOCS Mission Manager

Components are responsible for managing their own operational mode. They decide on mode changes based on their observation of the external environment. Part of this information can come from monitoring the state of other AOCS objects but part must come from outside the AOCS.

The ground station could, for instance, provide information about changes in orbital conditions, or the beginning of delta-V manoeuvres. Similarly, the OBDH could send commands to force the AOCS into survival mode.

An object – the *AOCS Mission Manager* – is provided to supply this information to the rest of the AOCS software. This object has an operational mode. Its operational mode plays a special role and is therefore given a special name: *mission mode*. The mission mode would normally be set by telecommands originating either in the OBDH or at the ground station.

In principle, all components endowed with operational mode should register their interest in the mission mode property.

21.5 Operational Mode Control

In current AOCS systems, the ground can either force the AOCS to go into a specified mode or it can inhibit certain autonomous mode transitions from taking place. It is unclear whether it still makes sense to implement such a control over mode management with the proposed, more distributed, mode concept.

In principle, it is of course possible to have telecommands to control the mode of individual objects but because changes in the mode of one single object reverberate throughout the AOCS through the environment observation mechanism, it may be unsafe to do so.

If fine control over mode changes is required, it is safer to introduce methods that inhibit certain mode transitions at the object level. Such methods would normally be called by telecommands. Inhibition of mode changes is safer than forcing mode changes because it always leaves the AOCS in a consistent state.

Fallbacks to safe “survival” modes could be realized through the mission manager. If the mission manager is endowed with a survival mode, its transition into this mode would alert all AOCS objects of the need to go to a correspondingly safe mode.

21.6 Reusability and Extensibility Issues

The concept proposed here achieves the objectives of reusability and ease of extension because it separates the implementation of the mode-specific algorithms from the mode switching logic. It then becomes possible to construct components whose mode-specific behaviour is delegated to other plug-in components. Such components can be reused across missions without any changes. Functionality extensions occurs through object composition.

22 MANOEUVRE MANAGEMENT

A *manoeuvre* is a sequence of actions that must be performed by the AOCS at specified times to achieve a specified goal. Examples of manoeuvres include:

- A sequence of small torque impulses imparted to the spacecraft to cause the speed of the reaction wheels to change in a desired manner (*wheel momentum unloading*).
- A slew to change the direction of pointing of the spacecraft payload.
- A sequence of thruster firings to change the spacecraft orbit in open-loop mode.

Manoeuvres are highly mission-dependent. The architecture proposed here is simply intended to provide a standard, re-usable, interface through which manoeuvres can be controlled. Specific manoeuvres will obviously have to be defined on an *ad hoc* basis.

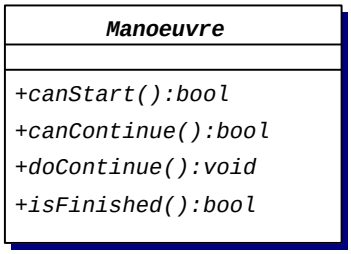
Manoeuvres are encapsulated in *manoeuvre objects.*

The software component in charge of executing manoeuvres is the *manoeuvre manager*.

22.1 Manoeuvre Objects

In order to allow uniform and mission-independent treatment of manoeuvres, it is necessary to encapsulate them in objects derived from a common base class or implementing the same abstract interface.

The following abstract base class is defined for manoeuvre objects:



Manoeuvre actions must be performed over a well-defined time interval which may be defined by the manoeuvre itself. Method canStart returns true when the manoeuvre is ready to start.

Manoeuvres are executed in several steps. Method `doContinue` is called by the manoeuvre manager to advance the execution of the manoeuvre. When a manoeuvre object receives this command, it check the current time or it verifies current operational conditions and it performs any actions that are due for execution.

Manoeuvre objects must be able to check whether the conditions for their own continued execution are appropriate. For instance, a manoeuvre to perform a slew should periodically check that the spacecraft is indeed following the slew profile. If the deviation from the slew profile exceeds some pre-specified threshold, then an error is likely to have occurred and the manoeuvre should be aborted. In another example, consider the manoeuvre to perform an open-loop delta-V. This manoeuvre should only be performed if the spacecraft is in Orbit Correction Mode. If this is not the case, the manoeuvre should be aborted.

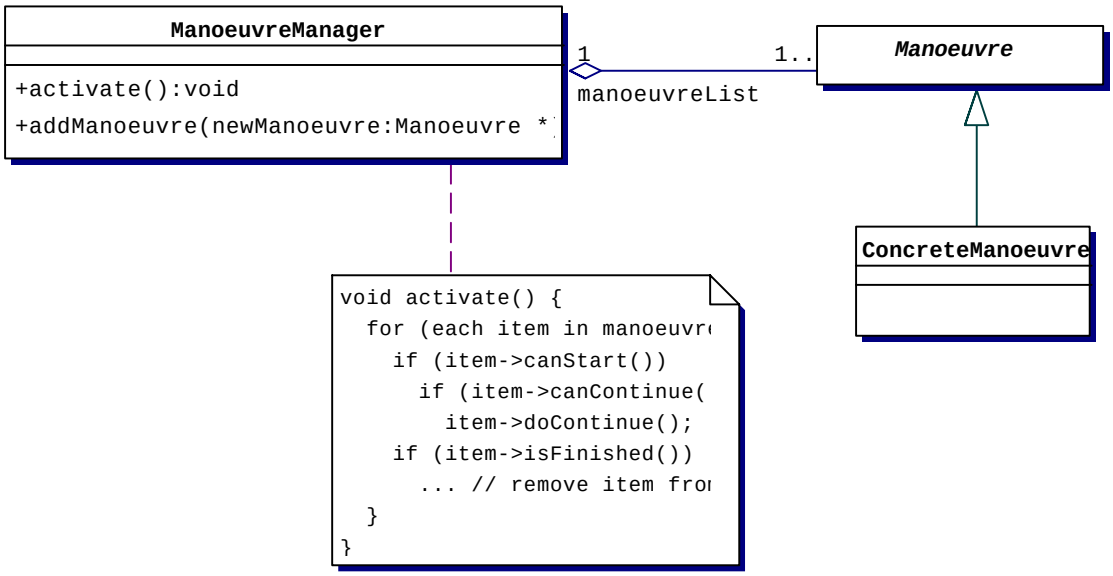
Method `canContinue` can be called to verify if the conditions for the continued execution of the manoeuvre hold.

Method `isFinished` returns `true` when the manoeuvre has terminated.

Since manoeuvres are application-specific, the implementation of the abstract class `Manoeuvre` represents one of the framework hot-spots (the *manoeuvre hot-spot*).

22.2 The Manoeuvre Design Pattern

This design pattern is introduced to address the problem of separating the management of manoeuvres from their implementation. The design pattern is based on the manager meta-pattern of section 5.4 and is illustrated in the following class diagram:



The manoeuvre manager holds a list of manoeuvre objects that are seen through their base class `Manoeuvre`. Presence of this abstract class separates the management of manoeuvres from their implementation.

At each activation, the manoeuvre manager goes through the list of pending manoeuvres, checks which ones are due for execution and which ones are already executing and are in a condition to continue execution and on all these it calls method `doContinue` to advance the manoeuvre execution. Finally, the manoeuvre manager checks whether manoeuvres have terminated their execution and, if they have, remove them from the list of pending manoeuvres.

Note that, in principle, clients can only *add* manoeuvres to the manoeuvre manager list. Removal from the list is done autonomously and internally to the manoeuvre manager when a manoeuvre has terminated its execution.

The manoeuvre pattern is instantiated as follows in the AOCS framework:

- The manoeuvre manager is implemented as an active object (see section 8) and its `activate` method is the `run` method declared by interface `Runnable`.
- The manoeuvre manager is given responsibility for creating events that record the beginning and end of a manoeuvre and other manoeuvre-related occurrences.
- The manoeuvre base class is implemented with some additional functionalities such as manoeuvre abort or manoeuvre hold.
- The execution status of manoeuvres is implemented as a bound property.

22.3 Manoeuvre Initiation

In a typical implementation, the AOCS software will internally store a number of manoeuvre objects to perform often recurring manoeuvre such as wheel unloading, slews, delta-V, etc. When conditions warrant it or when the ground commands it, one of these pre-defined manoeuvre objects can be loaded into the manoeuvre manager (by calling its method `addManoeuvre`). This will cause the manoeuvre to be executed.

Thus, for instance, when the ground decides to perform a delta-V manoeuvre, and assuming that the manoeuvre object is already available on board, it can uplink a telecommand that performs the following method call:

```
manoeuvreManager.addManoeuvre(deltaVManoeuvre);
```

In another example, if the failure detection mechanism has detected an over-speed condition in one of the reaction wheel, it will report the fact as an event. This event will then be

processed by the failure recovery manager that could respond by loading the reaction wheel manoeuvre into the manoeuvre manager.

Obviously, new manoeuvre objects can also be uplinked by telecommand.

22.4 Profiles

A very common kind of manoeuvre consists in driving a controller to follow a pre-defined profile. This will typically result in the spacecraft performing slews but profiles can also be used to perform wheel unloading (in which case it is the wheel speed that is controlled in closed loop to follow a pre-defined profile).

Given the frequency and importance of this type of manoeuvre, it might be advisable to define a dedicated class for them that extends the base class `Manoeuvre`. This however is not done in the framework prototype.

22.5 Alternative Implementation

There is an obvious similarity between manoeuvres as defined here and telecommands as defined in section 26. Both categories of objects encapsulate actions that must be performed on the AOCS software. Perhaps the main difference is that telecommands are intended to be performed in one shot whereas the execution of manoeuvres extends over time. This is not an essential difference though because telecommands could be seen as special cases of manoeuvres with a punctual execution. Future versions of the AOCS framework might therefore consider merging the two concepts of telecommand and manoeuvre.

22.6 Reusability and Extensibility Issues

The concept proposed here achieves the objectives of reusability because it completely decouples the task of *managing* manoeuvres from the task of *carrying them out*. In this way, the manoeuvre manager becomes mission-independent and reusable across missions without changes.

Manoeuvre are encapsulated in objects and can therefore be changed with only local repercussions thus promoting ease of extensibility.

23 OVERALL FDIR APPROACH

The next two sections deal with two framelets that taken together cover the failure detection and failure recovery functionality. The present section describes the overall approach to FDIR in the AOCS framework. Failure detection and failure recovery are discussed first and failure isolation is discussed in the last sub-section.

It should be stressed that FIDR algorithms are very mission-specific and cannot therefore be part of the AOCS framework. Hence, the main issue in designing the framework is *not* the identification and implementation of generic FIDR algorithm but it is rather the definition of an architecture that will support the implementation of such algorithms.

23.1 Failures and Configuration Errors

The AOCS framework recognizes two major categories of software faults: *failures* and *configuration errors.*

The term *configuration error* is used to designate any fault that is detected by a component while it is being configured. Configuration errors would typically occur during application initialization when the application components are configured but they could also occur as a result of a telecommand or some other action attempting to reconfigure a component dynamically.

The term *failure* is used to designate any fault that is detected during the normal operation of a component and that affects its ability to perform its allotted task. Failures would normally occur only during normal operations.

Both failures and configuration errors are recorded through the creation of dedicated events (the *failure events* and the *configuration error events*).

The rest of this section deals with failures only.

23.2 Separation of Functions

The AOCS framework regards failure detection and failure recovery as separate functionalities allocated to distinct components. The failure detection component performs failure checks and, when it detects a failure, it creates an event that describes it and stores it in an event repository.

The failure recovery component inspects the failure event repository to check whether any failures have been identified and then uses the information stored in the failure events to decide on the appropriate recovery action.

The alternative approach would have been to perform failure recovery as soon as a failure is detected. The advantage of this approach is promptness of response and the potentially greater amount of information about the failure that is available at the point where the failure is detected.

The first advantage however is minimal – provided that the failure recovery component is scheduled with the same frequency as the failure detection component. The second advantage is offset by the possibility that a separate failure recovery manager has to inspect *all* failures detected in a certain control cycle to perform global recovery actions that take account of the interrelationships of different individual failures.

23.3 Encapsulation of Functionalities

One of the basic principles at the heart of the AOCS framework is the separation of the management of a functionality from the implementation of the functionality itself. It is this separation that makes it possible to define mission-independent components.

In the case of failure detection, the failure test can either be encapsulated in a *monitoring check object* (see section 24.3) or in the very object that must be subjected to the failure test. In the latter case, this is done by having the object implement an interface that specifies that the object must be able to perform a consistency check upon itself (see section 24.1).

In the case of failure recovery, *recovery actions* – namely sets of related actions that address a specific failure – are encapsulated in dedicated objects. The framework specifies that to each failure detection check there must correspond one or more recovery actions that can be invoked to recover from that failure. Thus, whenever the application designer incorporates a component that performs a failure check into an application, he must also specify the corrective action to be associated to the detection of that failure.

As an example, consider for instance, a failure check that verifies whether a Kalman Filter diverges. An obvious recovery action that might associated to this failure could be to reset the filter or to change its parameters to remove the divergence.

23.4 Autonomous Detection of Failures

The failure detection component implements systematic failure checks but not all failures are detected by it. Consider, for instance, the case of an AOCS unit object that tries to access the AOCS bus to send an instruction to an external unit and finds that a bus error is returned. This object has detected an error. It must respond to the detection by creating an event and storing it in the event repository for failure detection events.

In general, all objects that encounter a failure during their normal operation can respond by creating the corresponding failure event and storing it in the failure event repository.

23.5 Failure Detection Levels

The current architecture of the AOCS framework foresees a single level of failure detection. This means that regardless of where they are found, failures are always treated in the same manner: they give rise to events that are stored in the same failure event repository.

This in particular means that failures encountered by the failure recovery manager while processing the failure event repository are also stored in the failure event repository itself. Thus, the failure recovery manager may end up processing failures that it has itself generated as a result of the implementing its failure recovery strategy.

Clearly, this situation could potentially give rise to the same failure events being generated cyclically. Note, however, that whether or not this situation arises depends on which recovery actions are associated to which failures. Since recovery actions are plug-in components that the application developer can load when it configures the framework, their judicious choice can avoid the problem from arising. This is the approach taken at present.

An alternative approach recognizes two levels of failure detections and recovery. To each level, is associated a distinct failure event repository. The framework normally runs in level 1. Failures detected at this level are stored in `FailureEventRepository_1`. When the failure recovery manager runs, the framework switches to level 2 and failure events are sent to `FailureEventRepository_2`. Thus, the danger of circularity is removed because while the failure recovery manager processes `FailureEventRepository_1`, any failures that are encountered are sent to `FailureEventRepository_2`.

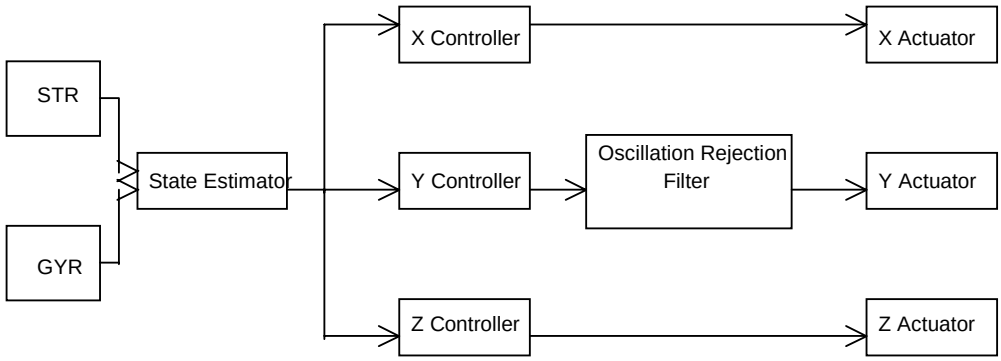
The drawback of this alternative approach is that failures sent to `FailureEventRepository_2` remain unprocessed. One could envisage a level 2 recovery manager that is responsible for `FailureEventRepository_2` but, if circularity is again to be avoided, this would require a third level of failure detection. In short, if circularity is to be avoided by design, then it is inevitable that there will be some failure events that cannot be processed within the framework.

23.6 Failure Isolation

Failure isolation is not supported by the prototype version of the AOCS framework because no generic failure isolation mechanism could be identified. One candidate mechanism that was considered but eventually discarded was based on the concept of *failure tracing*.

When a consistency or monitoring check fails on an object, the cause of the failure may lie either internally to the object itself or may be the result of a failure propagation from some other object. In general, a component has one or more *source components* (ie the components from which its data originate) and one or more *sink components* (ie the component where its data are forwarded). One way to support failure isolation is therefore to endow components with knowledge about who their sources and sinks are.

Consider for instance the figure below that illustrates the conceptual data flow[4] for a controller that uses star tracker and gyro outputs filtered by a state estimator to implement a 3-axis controller:



If, for instance, a consistency check reveals a failure in the `Xactuator` object, then it is useful for the failure isolation manager to be able to check whether the data source objects for this object also fail their consistency check. This might indicate that the source of the failure is not in the `Xactuator` object but in some other object upstream. Similarly, if a consistency check fails on object `StateEstimator`, it may be useful to check downstream objects. If all downstream objects also fail consistency check then this strengthens the hypothesis that the `StateEstimator` (or one of its data sources) are the origins of the failure.

In order to support this type of analysis, an interface of the following kind could be imposed on all framework components:

```
interface DataFlowObject {
     DataFlowObjectList getDataSource();
     DataFlowObjectList getDataSink();
```

[4] This is only a *conceptual* data flow because, as explained in section 13, the actual data are not transmitted directly from one object to another but are exchanged through shareable data areas (*data pools*).

```
}
```

Objects implementing this interface are part of a data flow path. Methods `getDataSource` and `getDataSink` return the list of data source and data sinks for the object and therefore allow tracing the data path both upstream and downstream.

As already mentioned, it was eventually decided not to support this mechanism because the information it would provide would not be useful in current AOCS systems (see section 3.3). It remains however an interesting option for future framework upgrades.

24 FAILURE DETECTION MANAGEMENT

The objective of the failure detection functionality is to perform failure detection tests on individual components. A failure is declared when a failure detection test fails.

The failure detection functionality is implemented by a *failure detection manager* that is based on the manager meta-pattern (see section 5.4).

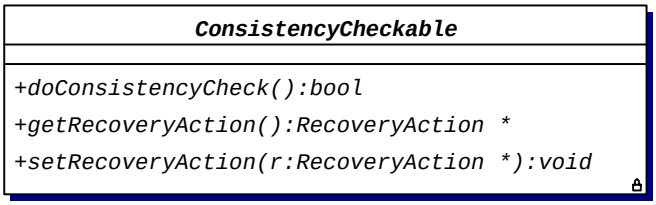
The general principle adopted in the framework is that the knowledge required to perform the failure detection test should be confined to the tested component. The task of the failure detection manager is simply to ask for the test to be performed, to record the results and to pass them on to other interested components. The actual test is performed by the tested component.

Two types of failure detection tests are foreseen: consistency checks and monitoring of property values.

24.1 Consistency Checks

A consistency check test verifies that the internal state of the object is consistent. For instance, a consistency check on a quaternion object verifies that the squared sum of the quaternion components is equal to 1 (within a certain tolerance band). In another example, a consistency check on a sensor object verifies that the sensor's outputs are in the range of physically possible values.

Objects that can perform a consistency check upon themselves implement the following interface:



A call to `doConsistencyCheck` causes the consistency check to be executed. When the consistency check succeeds (ie. when no errors are found), the method returns `true`. If the consistency check finds an inconsistency in the object's state, it returns `false` and the failure is then reported as an event. The event contains all the information that the object can make available to assist the identification of the cause of the failure. This event is stored in the

failure event repository from where it can be retrieved by other framework components (typically, the failure recovery manager, see section 25).

As explained in section 23.3, to each failure check must be associated a recovery action. Hence the `ConsistencyCheckable` interface provided setter and getter methods to allow the recovery action object to be defined and retrieved.

The consistency check mechanism described above can obviously implement s*ingle sensor* consistency checks but it can also be made to implement the *multi sensor* consistency checks. Suppose for instance that it is desired to check the mutual consistency of the sun sensor and star tracker. This can be done by creating an object that contains references to both the sun sensor and the star tracker objects. The `doConsistencyCheck` method of this new object will be responsible for performing the multi-sensor consistency check.

24.2 Consistency Check Management

The failure detection manager internally maintains a list of objects that must be subjected to a consistency check at every activation. Its core can be represented by code like:

```
ObjectList* consistencyCheckableList;

for (all objects in 'consistencyCheckableList') do {
     if ( !Object.doConsistencyCheck() )
       . . . //    a failure has been detected, form a failure event
       . . . //    and store it in its event repository
}
```

Thus, at each activation, the detection manager cycles through the objects in its list and calls `doConsistencyCheck` on each of them.

The `doConsistency` method is one of the framework hot-spots (the *consistency checkable* hot-spot*)* because it is the point where applications define their application-specific consistency checks.

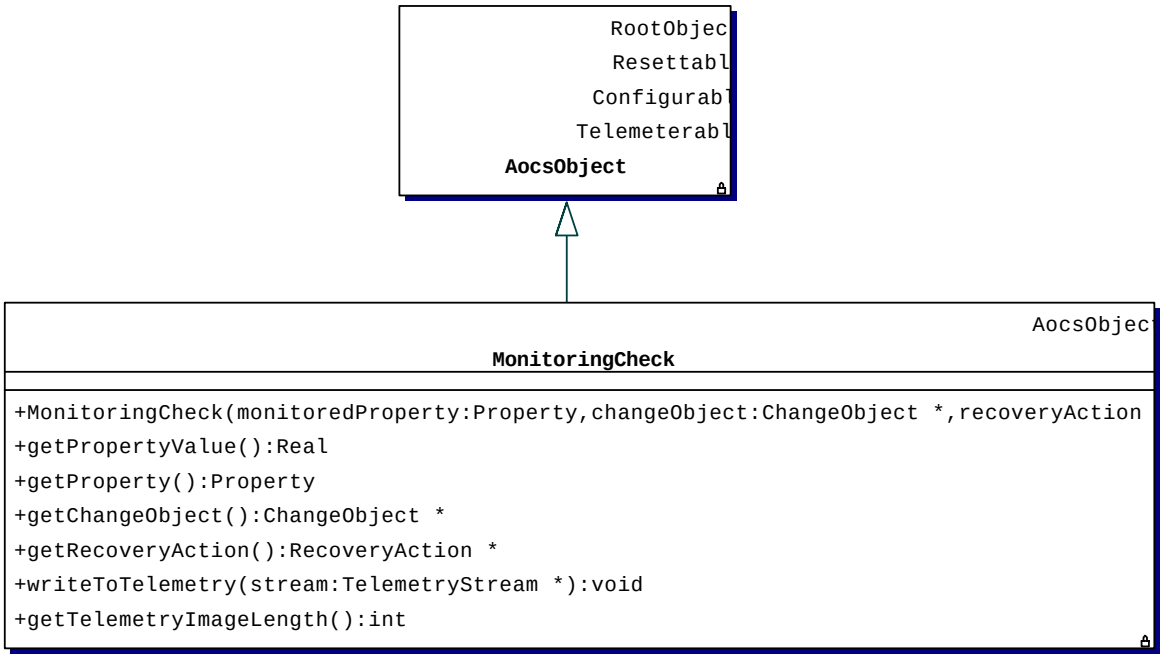
24.3 Property Monitoring

The concept of property monitoring was introduced in section 12. The monitored property is encapsulated in an object of type `Property`. The type of monitoring is encapsulated in an object of type `ChangeObject`.

The failure detection framelet defines an additional object, the *monitoring check*, that encapsulates a failure detection check based on property monitoring. A monitoring check

object packages in a single object the property to be checked, the change type and the recovery action to be performed in case the change is found to have occurred.

A monitoring check object is an object that is derived from the following base class:



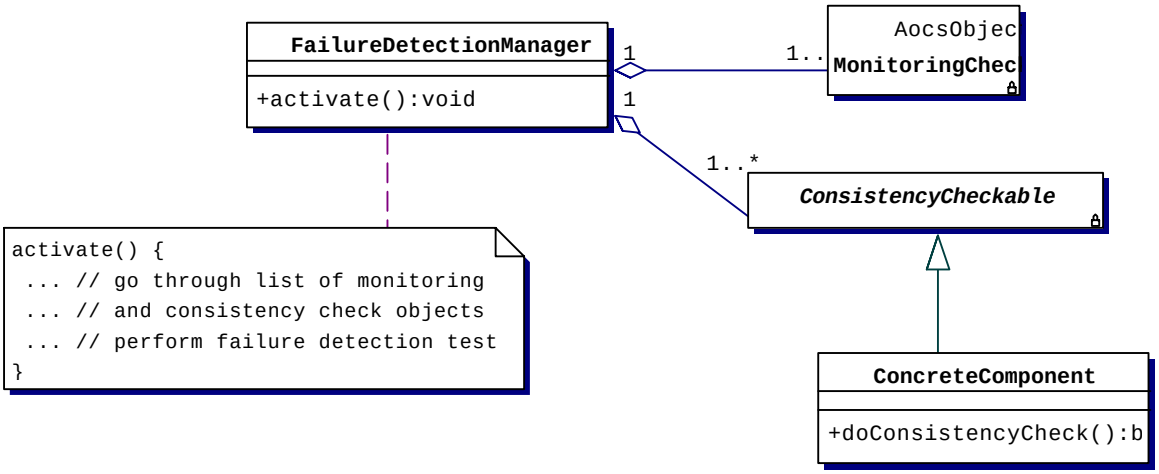
The implementation of the object monitoring part of failure detection is done by maintaining a list of monitoring check objects and passing each property value through the corresponding change object. When a change is detected, the fact is reported as an event stored in the failure event repository. The event also records the recovery action associated to the property and its change objects.

The type of monitoring check is obviously application specific and therefore the instantiation of the `MonitoringCheck` class is one of the hot spots of the framework (the *monitoring check* hot-spot).

24.4 The Failure Detection Design Pattern

This design pattern is introduced to address the problem of separating the management of failure detection tests from their implementations. It is based on the manager meta-pattern of section 5.4.

The pattern is illustrated in the following class diagram:



The failure detection manager maintains lists of monitoring check objects and of consistency checkable objects. When it is activated, it goes through the list and performs the failure detection tests on each item in the lists.

This pattern is instantiated as follows for the framework:

- the failure detection manager is an active object (see section 8) and its `activate` method is the `run` method declared by interface `Runnable`.
- If failures are found, they are reported as failure events stored in the failure event repository.
- If a monitoring check fails, then a property change event as well as a failure event is created.
- In most cases, the list of components to be subjected to consistency checks and the list of monitoring check objects depends on operational conditions. This is taken into account by making the failure detection manager mode-dependent (see section 21). The failure detection mode manager then manages two strategies corresponding to the list of consistency checkable objects and to the list of monitoring check objects.

24.5 Alternative Implementation

Part of failure detection is based on monitoring certain objects for specific types of change that might indicate that a failure has arisen. Object monitoring can be done either directly or by change notification. The AOCS framework bases failure detection on direct monitoring mechanism, which seems more appropriate to this function. However, monitoring by change

notification would also have been possible. In that case, the failure detection manager would have a purely passive role: it would simply provide a `PropertyChange` method to be called by the monitored object when a change has been detected.

24.6 Reusability and Extensibility Issues

The concept proposed here achieves the objectives of reusability and ease of extension because it completely decouples the task of managing the failure detection function from the task of carrying out failure detection tests. In this way, the failure detection manager becomes reusable across missions. Failure detection tests are encapsulated in objects and can be changed with only local repercussions. Changing the objects that are subjected to failure detection testing can be done dynamically without any impact on the software architecture.

25 FAILURE RECOVERY MANAGEMENT

The objective of the failure recovery function is to react to reports of failure detections to try to remove the cause of a failure.

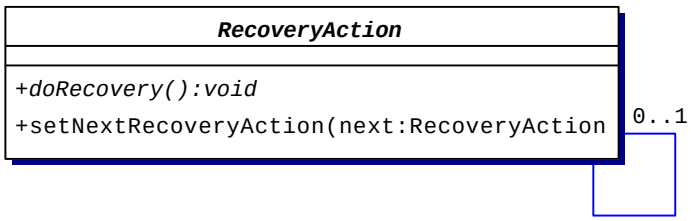
The failure recovery function is implemented by a *failure recovery manager*.

Failure recovery is highly mission-dependent. The software architecture proposed here is merely intended to assist the implementation of specific failure recovery algorithms, it *cannot* itself be considered as a failure recovery algorithm.

25.1 Failure Recovery Action

In general, the AOCS software can react to a failure event by performing one or more *failure recovery actions.* A failure recovery actions therefore represent a *local* response to a failure. The response is said to be local because it is based on a single failure report. A *global* response would take account of sets of failure reports.

Failure recovery actions are encapsulated in objects derived from class `RecoveryAction`:



The key method is `doRecovery` that causes the recovery action to be performed. This is a critical method that can only be called by the failure recovery manager.

The same failure event may give rise to several failure recovery actions. For this purpose, recovery actions can be chained together. Method `setNextAction` can be used to build up a chain of recovery actions.

The implementation of method `doRecovery` encapsulates an application-specific response to failures. It therefore represents a framework hot-spot (*recovery action hot-spot).*

25.2 Types of Recovery Action

Class `RecoveryAction` should be seen as an abstract class[5]. Concrete recovery actions are defined as subclasses of `RecoveryAction`. Typical failure recovery actions would include:

- *Reset of the entire AOCS software*

  This recovery action uses the services of the `SystemReset` object of section 9.2 offers a service to perform a complete software reset of the AOCS.

- *Reset of one or more AOCS objects*

  All AOCS objects must implement a reset method. This allows the failure recovery manager to selectively reset some objects.

- *Reconfiguration of one or more units*

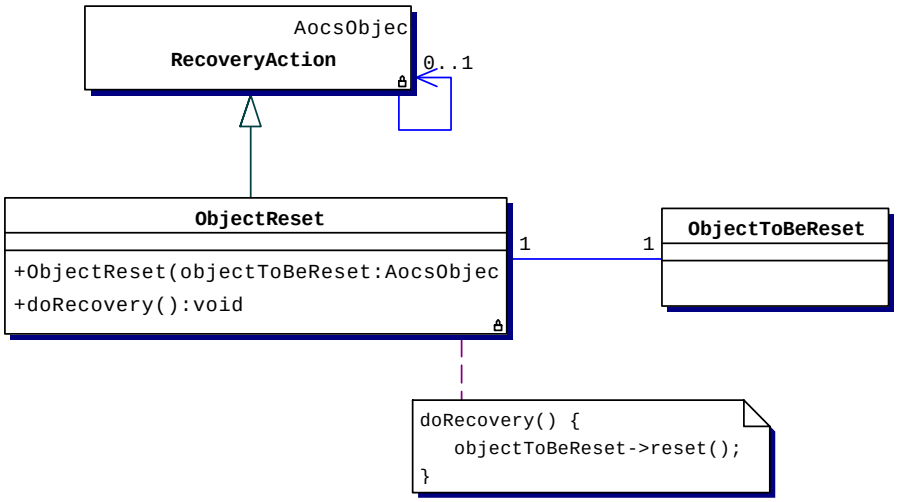
  Reconfigurable units are gathered together in reconfiguration groups for which a reconfiguration manager object is responsible (see section 0). Reconfigurations are performed by calling the `reconfigure` method on the reconfiguration manager.

- *Fall-back to a lower operational mode*

  In the AOCS framework, operational mode is a local property of each active object. Additionally, a mission manager is present to expose the AOCS mission mode. Mode fall-backs can therefore be implemented either at local level (changing the mode of single object) or at AOCS level (changing the mode of the mission manager).

As an example, consider the recovery action to reset an object together with a sample implementation of its `doRecovery` method:

[5] The framework actually implements it as a concrete class because it uses `RecoveryAction::doRecovery` to forward `doRecovery` calls along the chain of linked recovery actions.



The recovery action object maintains a reference to the object to be reset and when its `doRecovery` method is called, it resets it.

Thus, recovery actions can be packaged as configurable components and the AOCS framework offers a number of pre-defined components to perform the most common types of recovery actions.

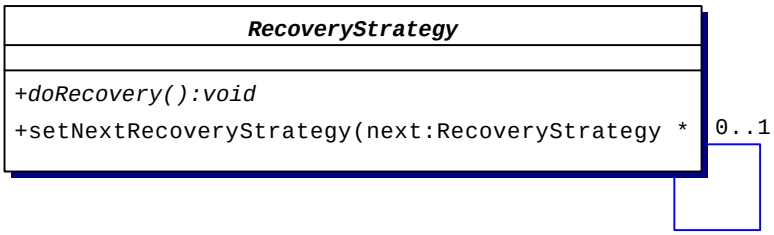
25.3 Recovery Actions with Memory

Recovery actions can be endowed with "memory". Consider for instance the case of a Kalman Filter to which a recovery action is associated that is triggered when the filter diverges. The nominal recovery for a filter divergence may be a filter reset. However, the recovery action object may be made to remember the last time it was called and, if it finds that it is called too frequently, it can decide that there is a fundamental control failure and may react by commanding a mode fall-back.

25.4 Failure Recovery Strategy

A *failure recovery strategy* is a set of coordinated responses to the failure events in the failure event repository. Unlike recovery actions, that act on individual failure reports, recovery strategies can take into considerations several failure reports and can therefore offer a higher-level response to failures.

Failure strategies are encapsulated in objects of type `RecoveryStrategy`:



Method `doRecovery` inspects the event repository and implements the appropriate failure recovery response.

Failure strategies can be chained together. This allows several failure strategies to be implemented in sequence.

The implementation of method `doRecovery` encapsulates an application-specific response to failures. It therefore represents a framework hot-spot (*recovery strategy hot-spot).*

25.5 Types of Failure Recovery Strategies

`RecoveryStrategy` should be seen as an abstract class[6]. Concrete failure recovery strategies are implemented as instances of concrete subclasses of `RecoveryStrategy`.

Typical failure recovery strategies include:

- *Sequence of local recovery actions*

  This strategy retrieves failure events from the event repository in sequence and performs the recovery action associated to each event. Checks are in place to avoid calling the same recovery action more than once.

- *System reset on too many failures*

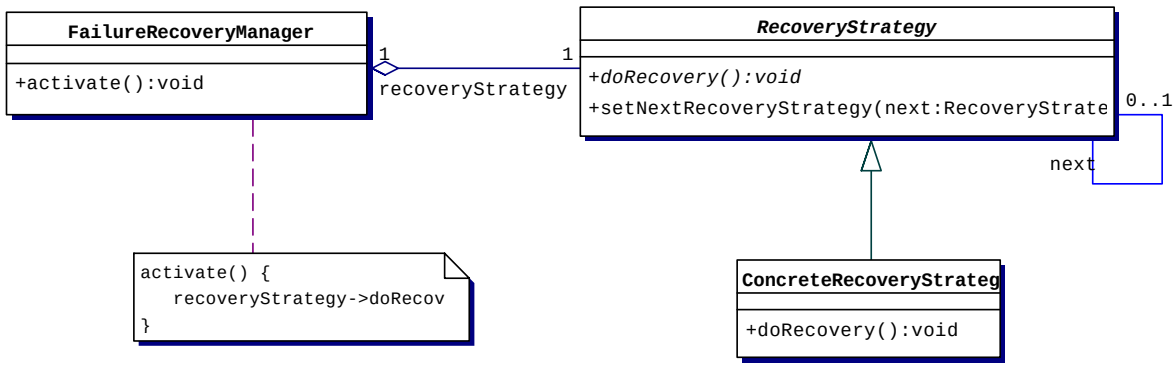
  This strategy checks the number of failure events in the repository and if it finds that it exceeds a predefined threshold, it commands a system reset.

These types of strategies are packaged in configurable components that are then provided as default framework components.

[6] The framework actually implements it as a concrete class because it uses `RecoveryStrategy::doRecovery` to forward `doRecovery` calls along the chain of linked recovery strategies.

25.6 Failure Recovery Design Pattern

This design pattern is introduced to address the problem of separating the management of failure recovery from the implementation of failure recovery actions and strategies. The design pattern is illustrated in the figure:



The failure recovery manager is essentially based on the *chain of responsibility* design pattern from RD1 but it can also be seen as an instance of the manager meta-pattern of section 5.4 where the list of functionality implementers only contains one element.

In the classical version of the chain of responsibility pattern, the client's request (in this case, the request to perform a recovery) is passed along the chain of handlers (the recovery strategies) until one is found who is able to handle it. Each request is intended to be handled by only one handler. In the application of the pattern to failure recovery, however, a recovery strategy when it receives a recovery requests performs the following actions:

- it handles the recovery request, and
- it checks whether the recovery request should be passed on to the next recovery strategy or whether recovery processing should terminate.

The recovery strategies are therefore executed in sequence but every recovery strategy has the chance to interrupt the chain. This incidentally means that the *order* in which the recovery strategies are linked in the list is important.

It would have been possible to implement failure recovery using a more straightforward version of the manager meta-pattern where the recovery strategies are arranged in a list and the recovery manager, when it is activated, goes through the list and executes each strategy in

sequence. This architecture, however, would have made it more awkward to give each a recovery strategy the option to interrupt the recovery process.

The failure recovery pattern is instantiated as follows for the framework:

- the failure recovery manager is an active object (see section 8) and its `activate` method is the `run` method declared by interface `Runnable`.
- Recovery events are created for each recovery strategy and recovery action that is executed.
- In most cases, the recovery strategy to be executed depends on operational conditions. This is taken into account by making the failure recovery manager mode-dependent (see section 21). The failure recovery mode manager then manages a single strategy corresponding to the recovery strategy to be supplied to the recovery manager.

25.7 Recursion

Use of the chain of responsibility design pattern introduces the possibility of recursion. A call to method `RecoveryStrategy::doRecovery` can be recursive if several recovery strategies are linked together. The maximum depth of the recursion is given by the maximum number of recovery strategies that are linked together.

Recursion can also arise because of the way recovery actions are linked together. A call to method `RecoveryAction::doRecovery` can be recursive if several recovery strategies are linked together. The maximum depth of the recursion is given by the maximum number of recovery actions that are linked together.

25.8 Alternative Implementation

As discussed in section 23.5, failure detection and recovery in the present version of the AOCS framework is done on a single level. From an implementation point of view, moving to the alternative approach that envisage two (or more) levels of failure detections and recovery would be straightforward. Multiple failure recovery manager can be simply created by instantiating multiple objects from class `FailureRecoveryManager.` Reporting of failures is delegated to a plug-in component[7]. Thus, the switch from one to a second level of failure reporting only requires the replacement of this plug-in component.

[7] Object `failureEvtRep` in class `AocsObject`.

25.9 Reusability and Extensibility Issues

The concept proposed here achieves the objectives of reusability and ease of extension because it completely decouples the task of managing the failure recovery function from the task of carrying out the failure recovery actions.

Two orthogonal levels of failure handling are provided. Recovery actions provide localized failure handling capabilities. Failure handlers implement global failure recovery strategies. Failure handling strategies and failure recovery actions can be defined without affecting any part of the failure handling code. In this way, the failure recovery manager becomes fully reusable across missions.

Both failure recovery actions and failure handling strategies can be dynamically updated thus providing operational flexibility.

26 TELECOMMAND MANAGEMENT

Telecommands encode actions to be performed on or by the AOCS software. The *execution* of a telecommand causes the action to be performed. The *telecommand manager* is the software component in the AOCS computer that is responsible for executing the telecommands.

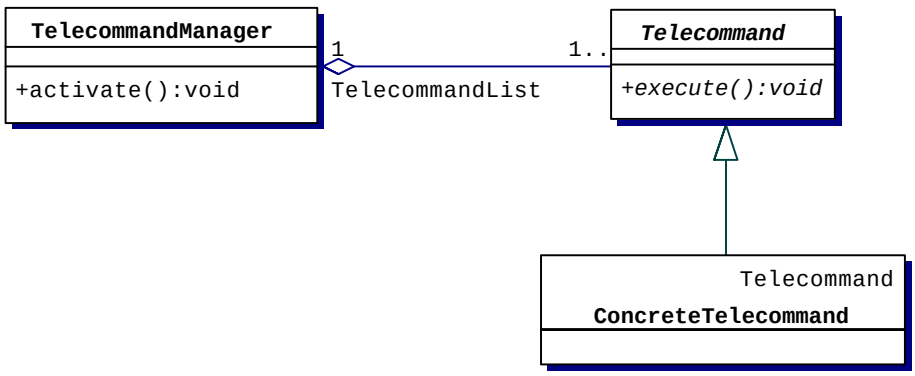
The AOCS software is assumed to be built as a collection of objects that expose certain interfaces. The telecommands use these exposed interfaces to perform their assigned tasks.

In practice, a telecommand is a component that exposes a method called `execute` which performs the actions associated with the telecommand itself. The `execute` method is called by the telecommand manager which in this way remains completely insulated from any knowledge of what the telecommand does and how it does it.

Consider as an example the case of a telecommand that must switch on a sensor. The sensor is implemented as an instance of class `AocsUnit`. As discussed in section 18.6, this class exposes a a `switchOn` method. The telecommand's `execute` method then simply calls `switchOn` on the sensor object.

26.1 The Telecommand Management Design Pattern

The telecommand management design pattern is introduced to address the problem of separating telecommand management from telecommand implementation. It is closely based on the *command design pattern* from RD1 as illustrated in the figure:



The telecommand manager maintains a list of pending telecommands and, when it is activated, it goes through the list and executes all telecommands in sequence.

The telecommand pattern can also be seen as an instance of the manager meta-pattern where telecommand execution is the pattern functionality, the telecommand manager is the functionality manager, and the `Telecommand` class decouples the functionality management from the functionality implementation.

The telecommand management pattern is instantiated in the AOCS framework as follows:

- The telecommand manager is implemented as an active object (see section 8) and its `activate` method is the `run` method it inherits from interface `Runnable`.
- Telecommand events are created when telecommands are executed, when their execution fails, when they loaded in the telecommand manager, etc.
- The `Telecommand` interface is replaced by a concrete class with overridable methods. This allows it to encapsulate some functions that are common to all telecommands.
- The base class `Telecommand` is endowed with extra functions in addition to telecommand execution to disable and enable telecommands, to allow telecommands to check whether operational conditions are compatible with their execution, etc
- A telecommand loader component is introduced to load telecommands into the telecommand manager (see section 26.4).

The definition of concrete telecommands – through the subclassing of class `Telecommand` – is obviously application-specific and represents a framework hot-spot (the *telecommand hot-spot).*

26.2 The Telecommand Transaction Design Pattern

In current systems, execution failure for a telecommand is reported in telemetry and it is then left to the ground to take whatever corrective action is appropriate.

The AOCS framework introduces the *telecommand transaction design pattern* to address the problem of treating some telecommands or some sequences of telecommands as *transactions* where the term “transaction” is used in the same sense in which it is used in database systems to designates an atomic operation that can either succeed or fail and that, in case of failure, restores the initial state of the system. Thus, transactions are safe because even in case of failure they leave the AOCS software in a consistent state.

Consider for instance a situation where the AOCS is in mode A and must make a transition to mode B. Suppose also that mode B requires units 1, 2 and 3 to be switched on. One would use the following sequence of telecommands to perform the desired transition:

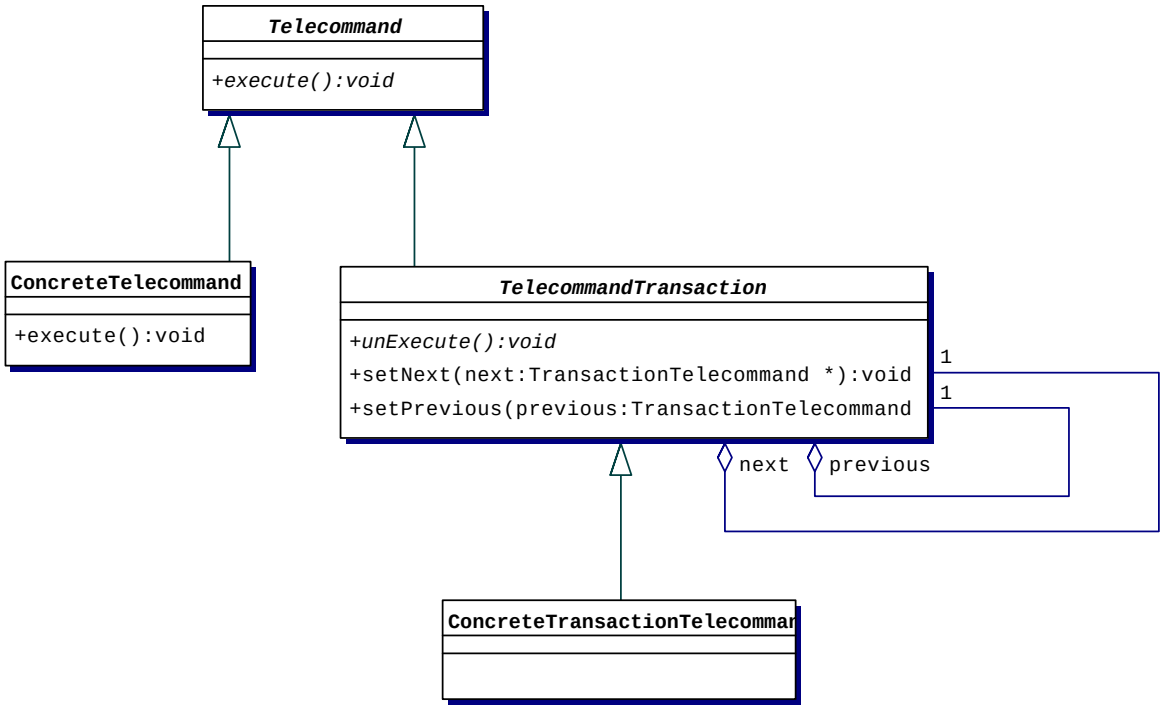
- switch on unit 1;
- switch on unit 2;
- switch on unit 3;
- switch to operational mode B.

If, say, the second telecommand fails, two corrective actions should be performed:

- abort the telecommand sequence, and
- switch off unit 1.

Note that simply aborting the telecommand sequence would leave the AOCS in an inconsistent state where unit 1 (which should be switched off in mode A) remains powered. Consistency in case of intermediate failures will be assured by treating the telecommand sequence as a transaction.

The telecommand transaction design pattern calls for the introduction of a class `TransactionTelecommand` that extends calls `Telecommand` with an `unExecute` method that "undoes" the actions of the telecommand:



Note that, as shown in the class diagram, telecommand transactions must be chained in a link to allow their manager to recursively undo all telecommands in the same transaction.

The telecommand transaction pattern is instantiated in the AOCS framework as follows:

- The interface `TransactionTelecommand` is replaced by a concrete class derived from class Telecommand
- Class `TransactionTelecommand` provides default (and overridable) implementations of methods `execute` and `unExecute` that take care of checking the success or otherwise of a telecommand and, if necessary, undo its action by calling `unExecute`. These implementations are such that the telecommand manager need not be aware of the distinction between telecommands and telecommand transactions.

Note that concrete transaction telecommands must provide implementations for both `execute` and `unExecute`. In some cases, unexecution is either not possible or not desired. In this case, the telecommand transaction becomes merely a *telecommand sequence*, namely a sequence of telecommands that are executed as a single telecommand but for which there is no guarantee of system integrity in case of partial or complete failure.

26.3 Recursion

The telecommand transaction design pattern introduces the possibility of recursion since a call to method `execute` can now be recursive. The maximum depth of the recursion is given by the maximum number of telecommands that are chained into a single telecommand transaction.

26.4 Telecommand Loading

Telecommands are loaded dynamically into the AOCS computer memory. The loading mechanism is application-dependent. Two common mechanisms are:

- the telecommand is loaded via DMA under the control of hardware that is external to the AOCS computer. The completion of a load operation is indicated by an interrupt to the AOCS software.
- the telecommand words or bytes are loaded by the AOCS software via I/O commands. The arrival of a new word or byte is indicated by an interrupt to the software.

A telecommand is an object made up of data and code. The link between the data and the code is provided by a virtual function table. Two types of telecommand loading are foreseen:

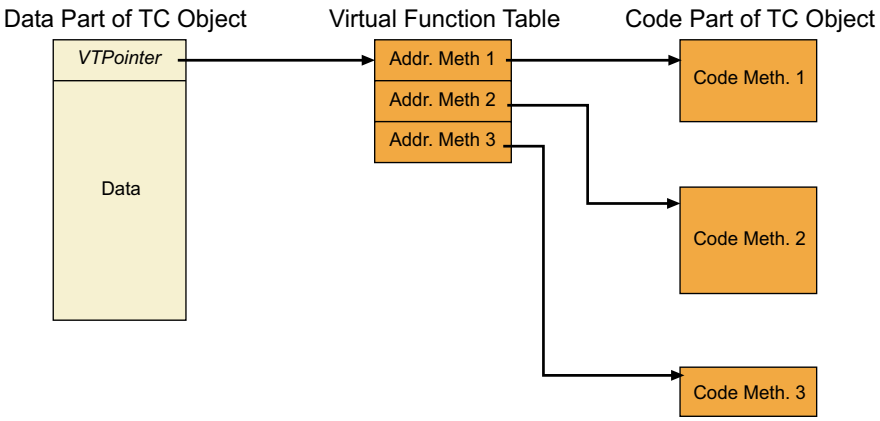
- *Data Load*

The code for the telecommand is already present in the AOCS software and only the data part of the telecommand object is physically loaded into memory.

- *Full Load*

  Both the code and the data for the telecommand object are loaded.

The two telecommand load types are illustrated in the figure:



In a data load, only the yellow part of the figure is physically loaded onto the AOCS computer memory. In the full load case, both the yellow and red parts are loaded. The example in the figure assumes that the code for all methods required by the telecommand must be loaded. In practice, some methods will be able to rely on code already present in the AOCS computer (typically, this is because such methods are inherited without changes from a superclass of the concrete telecommand class that is already present in memory).

Normally, an AOCS application will include some predefined telecommand classes implementing common telecommand actions. For such telecommands, only the data need to be loaded. Unusual or unforeseen telecommands need to be loaded in full.

Since telecommand loading is application-specific, the framework cannot provide a generic telecommand loader component. Instead, it provides an abstract interface that has to be implemented by concrete telecommand loaders and that allows their integration with other framework components. This interface is called `TelecommandLoader` and the components implementing it are called *telecommand loaders.*

The definition of concrete telecommand loaders – through the implementation of interface `TelecommandLoader` – is obviously application-specific and represents a framework hot-spot (the *telecommand loader hot-spot).*

26.5 Reusability and Extensibility Issues

The *advantages* of the concept proposed here are:

- Very high expressiveness;
- Complete de-coupling between telecommand structure and telecommand handler;
- Ease of extensibility;
- Complete re-usability of telecommand handling software across missions;
- Very high re-usability of telecommands across missions.

The *disadvantage* of this concept is:

- Bigger telecommand size

The greater size of telecommands comes from the fact that the telecommands are carrying code as well as data. However, the complexity of this code should not be exaggerated. Note also that the base classes from which telecommands are derived can be stored on board. The *concrete telecommands* only need to carry along the code that defines their differential behaviour with respect to the default implementation in the base class. In most cases, this delta code will reduce to a single method calls.

The high degree of re-usability, of both individual telecommands and telecommand management software, is a consequence of the complete de-coupling between the content of telecommands and the logic required to execute them. Individual telecommands can be re-used because they consist of calls to public methods – whose syntax presumably does not change from mission to mission. The telecommand manager being independent of telecommand content can also be re-used without changes.

The independence of the telecommand handlers from the content of telecommands also means that new telecommands can be added without any impact on the telecommand management software thus achieving the objective of ease of extensibility.

The expressiveness of the telecommands with this approach can be exploited to combine into a single telecommands actions that would otherwise result in several telecommands. Consider for instance the case where four units have to be switched on in sequence. With the traditional telecommand concept, this requires sending four telecommands one after the other. With the proposed telecommand concept, the commands can be naturally combined into a single telecommand. This results in a reduction of overhead that somewhat compensates for the higher overhead intrinsic to the active telecommand concept.

27 TELEMETRY MANAGEMENT

Telemetry data are generated cyclically by the AOCS for transmission to the ground station. They represent (a subset of) the software state of the AOCS and are used on ground to reconstruct (a subset of) the software state of the AOCS.

Telemetry data are transmitted in frames. A *telemetry frame* represents the set of data that are sent by the AOCS to the ground in one telemetry transmission cycle.

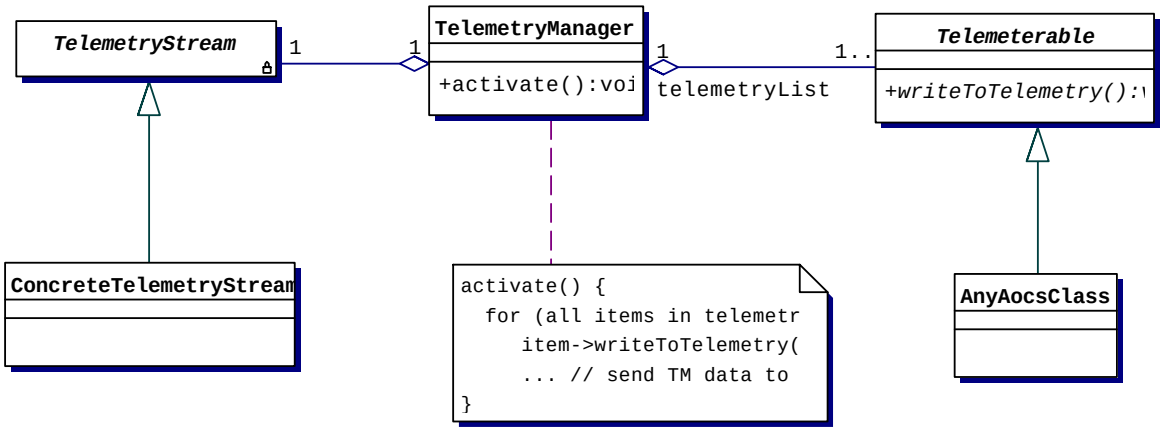
The *telemetry stream* is the data sink to which telemetry data are written. It represents the channel through which telemetry data are forwarded to the ground station.

The telemetry concept proposed by the AOCS framework assumes that the AOCS software is organized as a collection of objects with each object potentially capable of writing its own state to the telemetry stream. A telemetry manager component is responsible for managing the flow of telemetry data.

27.1 The Telemetry Management Design Pattern

This design pattern is introduced to address the problem of separating the management of telemetry data from the layout and content of the telemetry frames. It is based on the manager meta-pattern of section 5.4.

The pattern is illustrated in the following class diagram:



The telemetry manager maintains a list of references to objects of type `Telemeterable`. Telemeterable objects are objects that are capable of writing their own internal state to the telemetry stream.

The telemetry manager additionally maintains a reference to the telemetry stream. The characteristics of concrete telemetry streams vary widely across AOCS applications and therefore no generic telemetry stream component can be provided. The AOCS framework characterizes telemetry stream through the abstract interface `TelemetryStream`.

The telemetry management pattern is instantiated in the AOCS framework as follows:

- The telemetry manager is implemented as an active object (see section 8) and its `activate` method is the `run` method it inherits from interface `Runnable`.
- Methods are added to the `Telemeterable` interface to control the format of the telemetry image generated by telemeterable components: it is assumed that telemeterable objects can generate four different telemetry images with different content and layout. Interface `Telemeterable` allows the telemetry manager (or other components) to select the format of the telemetry image of each telemeterable object.
- The telemetry stream is passed as a reference to the `writeToTelemetry` method. Forwarding of telemetry data to the telemetry stream is thus done directly by telemeterable objects.
- In order to ensure that all non-trivial AOCS objects can potentially have their state included in telemetry, interface `Telemeterable` is implemented by class `AocsObject` (see section 6.9)

The definition of the layout and content of the telemetry image of telemetry object – through the implementation of interface `Telemeterable` – is obviously application-specific and represents a framework hot-spot (the *telemetry hot-spot).*

The definition of concrete telemetry stream – through the implementation of interface `TelemetryStream` – is also application-specific and represents a further framework hot-spot (the *telemetry Stream hot-spot).*

27.2 Alternative Implementation

The most straightforward way to implement the proposed telemetry concept is to treat telemetry reporting as a form of *serialization* (in the Java sense of the term, see RD6 for a description). Telemetry then becomes an output stream to which the state of objects marked for inclusion in telemetry are serialized.

This approach presents the following advantages:

- *Simplicity of Implementation*

As in Java, serialization can be handled either by default methods or by object-specific methods. If the default serializer is used, then serialization is effectively transparent to objects thus simplifying their development.

- *Strong Heritage*

  Serialization is implemented in a number of languages and is therefore a well-understood process for which good architectural and implementation models exist. This would result in a robust and reliable implementation.

- *Symmetry between On-Ground and On-Board Telemetry Processing*

  Serialization allows objects to be fully reconstructed on ground. This achieves the objective of symmetric treatment between on-board and on-ground telemetry. Telemetry becomes a means of “cloning” AOCS objects on ground with considerable advantages in terms of reliability and simplicity of ground control software.

The drawbacks of this approach are:

- *Need for Language Support*

  An efficient implementation of serialization requires dedicated language support. Firstly, the language must make a distinction between data members that are part of the *persistent state* of an object, and should therefore be serialized, and *transient* data members that can be ignored during the serialization process.

  Secondly, meta-language support is required to let the serializer extract information about the number and type of the data members present in an object.

- *High Bandwidth Requirements*

  Standard implementations of serialization tend to generate large amounts of data Serialization introduces overheads because it does not packetize data in the most efficient manner (eg a boolean variable will take one or more bytes rather than just one bit) and because it adds to the stream possibly redundant information about class names.

  This requires a higher capacity for the telemetry channel which in turn translates into higher requirements on the bandwidth of the telemetry downlink.

The first drawback can be overcome depending on the language that is selected for the AOCS software. In the case of C++, for instance, standard libraries exist that provide language support for serialization. The second drawback is more serious because bandwidth is usually severely limited on spacecraft. Hence, serialization was not considered further in the design of the framework prototype. It may be reconsidered in future upgrades of the framework.

27.3 Reusability and Extensibility Issues

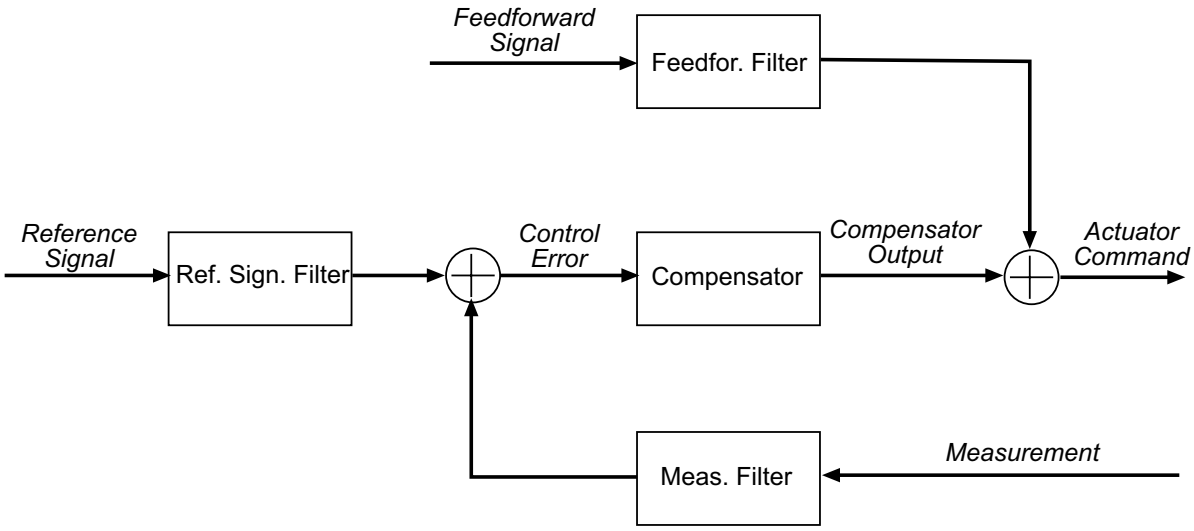
The concept proposed here achieves the objectives of reusability and ease of extension because it completely decouples the task of managing telemetry from the task of writing objects to telemetry. In this way, the telemetry manager becomes reusable across missions.

Changing the information going to telemetry can be done through local changes that affect only the objects concerned. Adding new objects for inclusion in telemetry instead can be done dynamically and does not have any impact at all on an existing system.

28 CONTROLLER MANAGEMENT

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 framework 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.

28.1 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.

28.2 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

The actuator command outputs are mandatory. The other two outputs are optional.

28.3 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

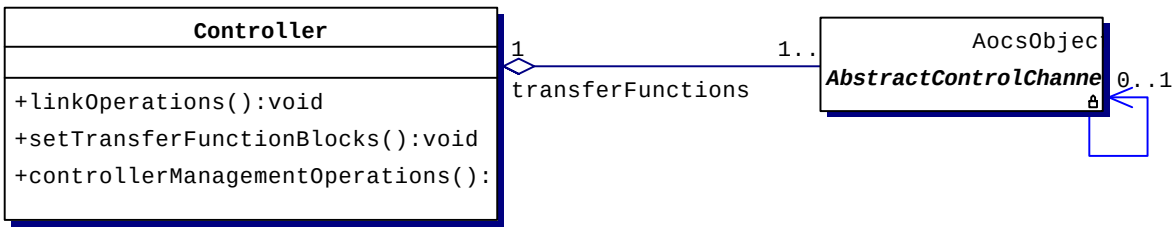
Explicit specification of the compensator transfer function is mandatory. The other transfer function blocks need not be specified by the user in which case they are implicitly assumed to be equal to 1.

The measurement filter block often represents an estimator.

The transfer function blocks are implemented in the AOCS framework using the control channel concept of section 16.

28.4 Controller Objects

The framework encapsulates closed-loop controllers of the kind presented in the first part of this section in dedicated components called *controller objects*. Controller objects are implemented as instances of class `Controller`. The conceptual structure of this class is:



This class exposes three categories of methods:

- methods to link the controller's inputs and outputs to data pool locations;
- methods to load the control channels representing the controller transfer functions;
- methods to perform controller management operations such as setting the controller in open or closed loop, checking the controller's stability, performing a control action, etc.

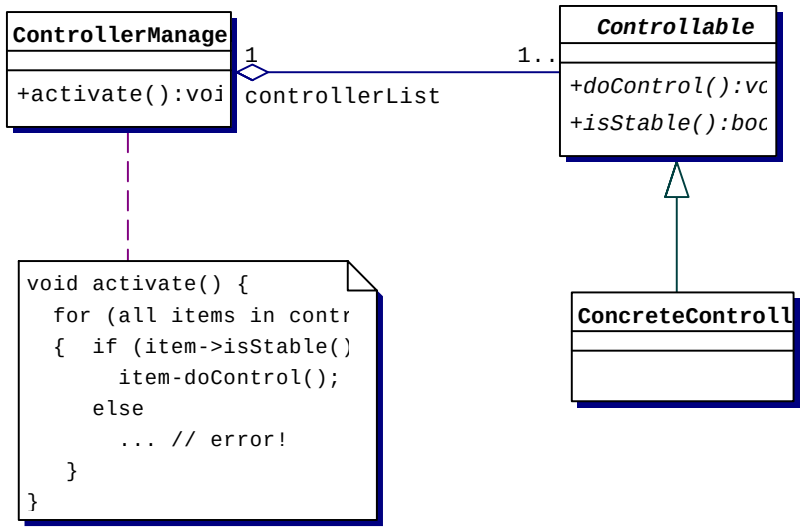
The controller class additionally holds multiple references to class `AbstractControlChannel` representing the control channels that implement the controller transfer functions.

The transfer functions to be implemented by a controller are application specific and represent one of the framework hot-spots (the *controller hot-spot*).

28.5 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 section 5.4.

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.

This pattern is instantiated as follows in the AOCS framework:

- The controller manager is implemented as an active object (see section 8) 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 28.4).
- 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 (see section 21). The controller mode manager then manages a single strategy represented by the list of controller objects that have to managed by the controller manager.

29 DOMAIN ABSTRACTION DICTIONARY

Domain abstractions capture the commonalities of the AOCS applications in the framework domain. They describe concepts and functional features that span application boundaries and are found in all domain applications and provide the vocabulary to describe the AOCS framework architecture.

Abstract Control Channel	Encapsulation of a generic multi-input-multi-output transfer function. See section 14 and RD23.
AOCS Data	Data that are produced or consumed on a periodic basis by AOCS components. See section 13 and RD21.
AOCS Event	An object that encapsulate a description of an asynchronous event. See section 13 and RD21.
AOCS Object	An object that is instantiated, directly or indirectly, from class `AocsObject` and that has access to the following basic services: time recovery, failure reporting, and configuration error reporting; and has the following properties: resettability, configurability, and telemeterability. See section 6.9 and RD20.
Bound Property	A property that is subject to monitoring through change notification. See section 12.2 and RD22.
AOCS Unit	Any device (normally a sensor or an actuator) that is external to the AOCS computer but internal to the AOCS subsystem. See section 18 and RD24.
Change Object	Encapsulation of a specific time profile for a property value. See section 11 and RD22.
Configurable	Property of an object that can clear its internal configuration and that can check whether it is correctly configured. See section 9 and RD20.
Configuration Error	Any fault that is detected by a component while it is being configured. Configuration errors would typically occur during application initialization when the application components are configured. See section 23.1.
Consistency Check	Failure detection test where a component is asked to check its own internal consistency. See section 24.1 and RD28.
Control Block	An abstract control channel that cannot be further decomposed into lower level control blocks. See section 14 and RD23.

Control Channel	Same as Abstract Control Channel above.
Control Super Block	Container for a chain of interconnected control channels. See section 14 and RD23.
Controller	Encapsulation of a generic closed-loop controller. See section 28 and RD33.
Cyclical Data	Same as AOCS Data above.
Data Pool	Shared memory area where AOCS data are stored. See section 13.8 and RD21.
Event Repository	Shared memory area where AOCS events are stored. See section 13 and RD21.
Failure	Any fault that is detected during the normal operation of a component and that affects its ability to perform its allotted task. Failures would normally occur only during normal operations. See section 23.1.
Fictitious AOCS Unit	Component that behaves to its client like a proxy for an external unit. See section 19 and RD24.
Manoeuvre	Sequence of actions to be performed by the AOCS at specified times to achieve a specified goal. See section 22 and RD27.
Mode Change Action	Actions to be performed when a mode-dependent component changes mode. See section 21.2 and RD26.
Monitor	A component that performs a monitoring action. Also called a monitoring component. See section 12 and RD22.
Monitored Property	A property that is subject to a monitoring action by a monitor. See section 12 and RD22.
Monitoring Action	Observation of a change over time in the value of a property. See section 12 and RD22.
Monitoring Check	Failure detection test where it is verified that a property does not undergo a pre-specified change. See section 24.3 and RD28.
Operational Mode	Attribute of a component whose behaviour depends on operational conditions. See section 21 and RD26.
Property	Attribute of an object that describes one aspect of its behaviour or of its internal state and that is accessible to external objects. See section 10 and RD22.
Property Object	Object of class Property that encapsulates a property. See section 10

	and RD22.
Reconfiguration	Switch between different independent implementation of the same functionality. See section 20 and RD25.
Reconfiguration Group	Set of objects that together offer a redundant functionality. See section 20 and RD25.
Recovery Action	Set of actions to be taken in response to a single failure report. See section 25.1 and RD30.
Recovery Strategy	Set of actions to be taken in response to a set of failure reports (normally, to the failure events present in the failure event repository). See section 25.4 and RD30.
Redundant Functionality	Functionality for which two or more independent implementations are provided. See section 20 and RD25.
Redundant Object	Member of a reconfiguration group. See section 20 and RD25.
Resettable	Property of an object that can bring its internal state to some initial default value. See section 9 and RD20.
Sequential Data Processing Chain	Sequence of independent processing stages through which AOCS data are passed. See section 14 and RD23.
State of an Object	The attributes of an object that are updated as part of the object's performing its allotted task as distinguished from those of its attributes that are normally set during the application initialization to configure it. See section 9.
Strategy	Encapsulation of a mode-dependent behaviour: to each operational mode there corresponds a strategy. See section 21 and RD26.
System Management Functionality	A function that is performed systematically on all AOCS objects present in the AOCS software at a given time. The *system manager* is the component responsible for performing system management functions.
Telecommand	Encapsulation of an action to be performed upon the AOCS software and uplinked from the ground station. See section 26 and RD32.
Telecommand Loading	Process whereby telecommands are loaded from an external interface ultimately communicating with the ground station into the AOCS application software. See section 26.4 and RD32.
Telecommand Transaction	A sequence of telecommands that are executed as a single transaction and whose actions, in case of partial or complete failure, can be reversed to leave the AOCS in the same state in which it was

	before the transaction was started. See section 26.2 and RD32.
Telemeterable	Property of an object whose state can potentially be included in the telemetry stream. See section 27 and RD31.
Telemetry Frame	Set of telemetry data sent to the telemetry stream in one single telemetry cycle. See section 27 and RD31.
Telemetry Stream	Data sink to which telemetry data are written representing the channel through which telemetry data are sent to the ground. See section 27 and RD31.
Transaction Telecommand	One of the telecommands making up a telecommand transaction. See section 26.2 and RD32.

30 FRAMEWORK DESIGN PATTERNS

The table below lists the framework-level design patterns identified in this document.

Control Channel Pattern
This design pattern is introduced to model the control channel concept and to allow both blocks and super blocks in a sequential data processing chain to be treated in a uniform manner. See section 16.3 and RD23.
Controller Pattern
This design pattern is introduced to separate the management of closed-loop controllers from their implementation. It is based on the manager meta-pattern. See section 28 and RD33.
Direct Property Monitoring Pattern
Pattern to directly monitor an object's property by accessing the property value or the property object through getter methods exposed by the property owner. Inspired by the JavaBeans architecture. See 12.1 and RD22.
Failure Detection Pattern
This design pattern is introduced to separate the management of failure detection tests from their implementation. It is based on the manager meta-pattern. See section 24 and RD28.
Failure Recovery Pattern
This design pattern is introduced to separate the management of failure recovery from the implementation of failure recovery actions. It is based on the manager meta-pattern and on the chain of responsibility pattern. This pattern introduces recursion. See section 25.6 and RD30.
Fictitious Unit Pattern
This design pattern is introduced to address the problem of combining components that process unit data without impacting the final users of the unit data. This pattern introduces recursion. See section 19 and RD24.
Manoeuvre Pattern
This design pattern is introduced to separate the management of manoeuvres from their implementation. See section 22 and RD27.
Mode Management Pattern

This design pattern is introduced to endow components with mode-dependent behaviour while separating the implementation of the mode-dependent strategies from the implementation of the logic governing mode switches. See section 21 and RD26.
Monitoring through Change Notification Pattern
Pattern to monitor a property by registering with the property owner and asking it to notify the monitor when a change of a certain type (as encapsulated by a change object) has occurred. Inspired by the JavaBeans architecture. See 12.1 and RD22.
Reconfiguration Management 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. See section 20.2 and RD25.
Shared Data Pattern
This design pattern is introduced to address the problem of allowing components to share access to data objects that are generated synchronously . See section 13.7 and RD21.
Shared Event Pattern
This design pattern is introduced to address the problem of allowing components to share access to event objects that are generated asynchronously. See section 13.2 and RD21.
System Management Pattern
The system management design pattern is introduced to address the problem of systematically performing the same set of operations on a target set of objects. The system management design pattern is obtained by instantiating the manager meta-pattern. See section 9 and RD20.
Telecommand Management Pattern
This design pattern is introduced to address the problem of separating the management of telecommands from their implementation. This design pattern is obtained by instantiating the manager meta-pattern. See section 26.1 and RD32.
Telecommand Transaction Pattern
This design pattern is introduced to address the problem of treating some sequences of telecommands as *transactions,* namely as atomic operation that can either succeed or fail and that, in case of failure, restores the initial state of the system. See section 26.2 and RD32.
Telemetry Management Pattern
This design pattern is introduced to address the problem of making the management of telemetry

data independent of the layout and content of telemetry frames. See section 27 and RD33.

31 FRAMEWORK HOT-SPOTS

The table below lists the framework-level hot-spots identified in this document. Detailed description of the hot-spots can be found in the framelet-level documents.

AOCS Unit Hot-Spot
AOCS applications must subclass `AocsUnit` to encapsulate interactions with external units. See section 18.5 and RD24.
AOCS Unit Hardware Hot-Spot
AOCS applications must implement interface `AocsUnitHardware` to encapsulate the interaction with the low-level hardware that controls data exchanges with external units. See section 18.5 and RD24.
Change Object Hot-Spot
AOCS applications must implement interface `ChangeObject` to encapsulate the change profiles in which they are interested. See section 11 and RD22.
Configurable Hot-Spot
AOCS objects must provide an implementation of interface `Configurable` to destroy their internal configuration and to check that they have been configured. See section 9 and RD20.
Consistency Checkable Hot-Spot
AOCS objects must provide an implementation of interface `ConsistencyCheckable` to define their application-specific consistency checks. See section 24 and RD28.
Control Block Hot-Spot
AOCS applications must provide the definition of the propagation algorithms for the control channels they use. See section 16.1 and RD23.
Controller Hot-Spot
AOCS applications must provide the definition of the control algorithms for the closed-loop controllers. See section 28 and RD33.

Data Pool Subclass Hot-Spot
AOCS applications must define the data pools where application data are stored. A data pool is created by deriving a class from `DataPool`. The derived class contains AOCS data objects that are logically related (eg. all AOCS data objects relative to attitude control, all AOCS data objects relative to orbit control, etc.). See section 13.8 and RD21.
Fictitious Unit Hot-Spot
AOCS applications must implement interface `AocsUnitFunctional` to encapsulate the processing of unit data. See section 19 and RD24.
Manoeuvre Hot-Spot
AOCS applications must define the application-dependent manoeuvres by subclassing class `Manoeuvre`. See section 22 and RD27.
Mode Change Action Hot-Spot
AOCS applications must define the application-specific actions to be associated to mode changes in mode-dependent components. See section 21.2 and RD26.
Mode Implementer Hot-Spot
AOCS applications must define the application-dependent strategy implementations for mode-dependent components. See section 21 and RD26.
Mode Manager Hot-Spot
AOCS applications must define the application-dependent mode switching logic for mode-dependent components. See section 21 and RD26.
Monitoring Check Hot-Spot
AOCS applications must define the properties that need to be monitored during failure detection tests and the type of change in their value that determines a failure. See section 24 and RD28.
Property Change Hot-Spot
AOCS applications must define the actions to be taken when a property subject to monitoring through change notification has undergone a change. See section 12.2 and RD22.
Reconfigurable Hot-Spot

AOCS applications must define the reconfiguration logic for the reconfiguration groups in their application. See section 20.2 and RD25.
Recovery Action Hot-Spot
AOCS applications must define the recovery actions to be used in their applications. See section 25.1 and RD30.
Recovery Strategy Hot-Spot
AOCS applications must define the recovery strategies to be used in their applications. See section 25.4 and RD30.
Resettable Hot-Spot
AOCS objects must provide an implementation of interface `Resettable` to reset their internal state. See section 9 and RD20.
Telecommand Hot-Spot
AOCS applications must provide their own application-specific telecommands by subclassing the base class `Telecommand`. See section 26 and RD32.
Telecommand Loader Hot-Spot
AOCS application must provide their own application-specific telecommand loader by implementing the abstract interface `TelecommandLoader`. See section 26.4 and RD32.
Telemetry Hot-Spot
AOCS applications must provide their own application-specific implementation of interface `Telemeterable` defining the content of the telemetry images of their components. See section 27 and RD31.
Telemetry Stream Hot-Spot
AOCS application must provide an application-specific implementation of interface `TelemetryStream` defining their telemetry stream. See section 27 and RD31.