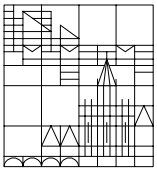


University of Constance
Dept. of Computer Science

MANOEUVRE MANAGEMENT FRAMELET

Concept And Architecture Description

Abstract

This document was written as part of the study "Design and Prototyping of a Software Framework for the AOCS" done under contract Estec/13776/99/NL/MV for ESA-Estec. The purpose of the study is the development of a software framework for the Attitude and Orbit Control Subsystem (AOCS) of a satellite. The framework will be built as a collection of framelets. This document describes the manoeuvre management framelet. This framelet proposes an architectural solution to the problem of managing manoeuvres such as wheel unloading, attitude slews, delta-V, etc. The framelet enhances reusability because it separates the task of managing the manoeuvres from the task of carrying them out.

Written By:	A. Pasetti / T. Brown	(University of Constance / SWE)
Date:	30 April 2002	
Issue:	2.1	
Reference:	SWE / 99 / AOCS / 012	

TABLE OF CONTENTS

1 REFERENCES

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

RD2 A. Pasetti (2000), *AOCS Framework – Concept Level Description*, AOCS Framework Document ref. SWE/99/AOCS/004

RD3 A. Pasetti (2000), *Inter-Component Communication Framelet – Concept and Architecture Description*, AOCS Framework Document ref. SWE/99/AOCS/005

RD4 Deleted

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

RD6 A. Pasetti (2000), *Telemetry Framelet*, AOCS Framework Document ref. SWE/99/AOCS/003

RD7 A. Pasetti (2000), *Failure Detection Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/010

RD8 A. Pasetti (2000), *Object Monitoring Framelet*, AOCS Framework Document ref. SWE/99/AOCS/008.

RD9 A. Pasetti (2001), *Software Frameworks and Embedded Control Systems*, LNCS Series, Springer-Verlag, To appear in Dec. 2001

2 ACRONYMS

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

3 INTRODUCTION

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

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

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

3.1 Context

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

The architecture proposed here follows the concept outlined in RD2.

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

3.2 Applicability to Java Version

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

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

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

– Manoeuvre objects in the C++ framework expose their current execution state as a property. Property objects do not exist in the Java framework. Manoeuvre objects are instead implemented as monitorable components (they are made to implement interface `Monitorable`).

– The manoeuvre event repository hot-spot (section9.3) and the change event repository hot-spot (section 9.4) are not applicable to the Java framework. Event repositories are event listeners and can be linked to the mode manager through the associated `addListener` methods.

3.3 Notation

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

The class diagrams use UML notation generated with the reverse engineering capabilities of the Together tool (version 4.0).

4 FRAMELET CONSTRUCTS

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

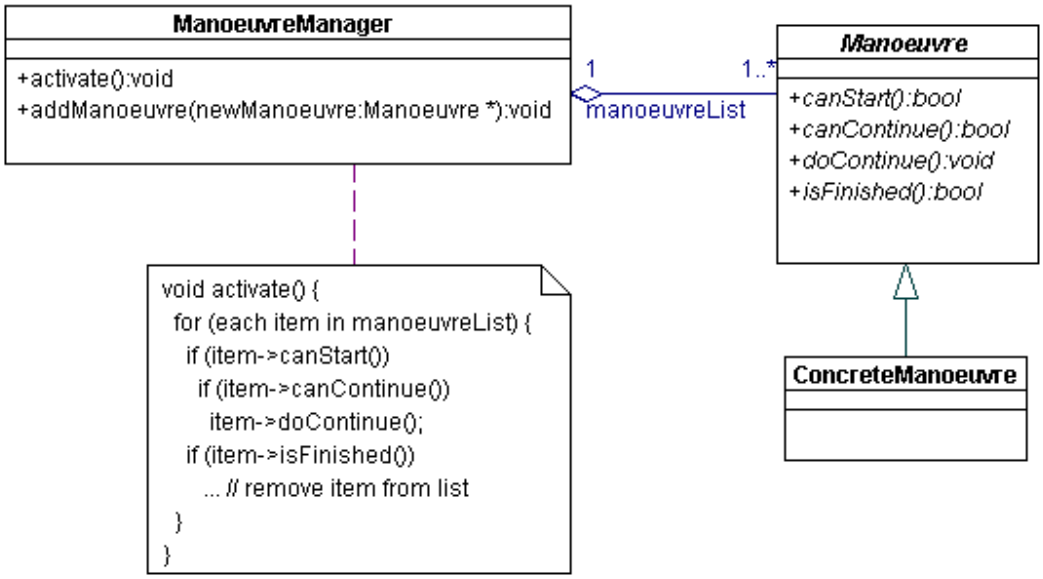
MANOEUVRE MANAGEMENT FRAMELET
Design Pattern
Manoeuvre Design Pattern: design pattern to separate the management of manoeuvres from their implementation.
Framelet Interfaces and Base Abstract Classes
`Manoeuvre` : abstract base class for manoeuvres
Framelet Core Components
`ManoeuvreManager` : component encapsulating a manoeuvre manager. This component maintains a list of currently loaded manoeuvres and is responsible for invoking methods of the `Manoeuvre` interface to control the execution of manoeuvres
Framelet Components
`ManoeuvreEvent` : component used to encapsulate information about a significant event in the life cycle of a manoeuvre `DummyManoeuvre` : a “dummy” implementation of the Manoeuvre base class useful for testing purposes `AttitudeSlew` : a manoeuvre which carries out a simple linear attitude slew

The components listed above are those for the prototype version of the AOCS framework. Later versions may offer a richer set of default implementations of the framelet interfaces. In particular, more default implementations of the `Manoeuvre` abstract base class might be provided.

5 THE MANOEUVRE DESIGN PATTERN

This design pattern is introduced to address the problem of separating the management of manoeuvres from their implementation. It is based on the manager meta-pattern of RD2.

The pattern 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.

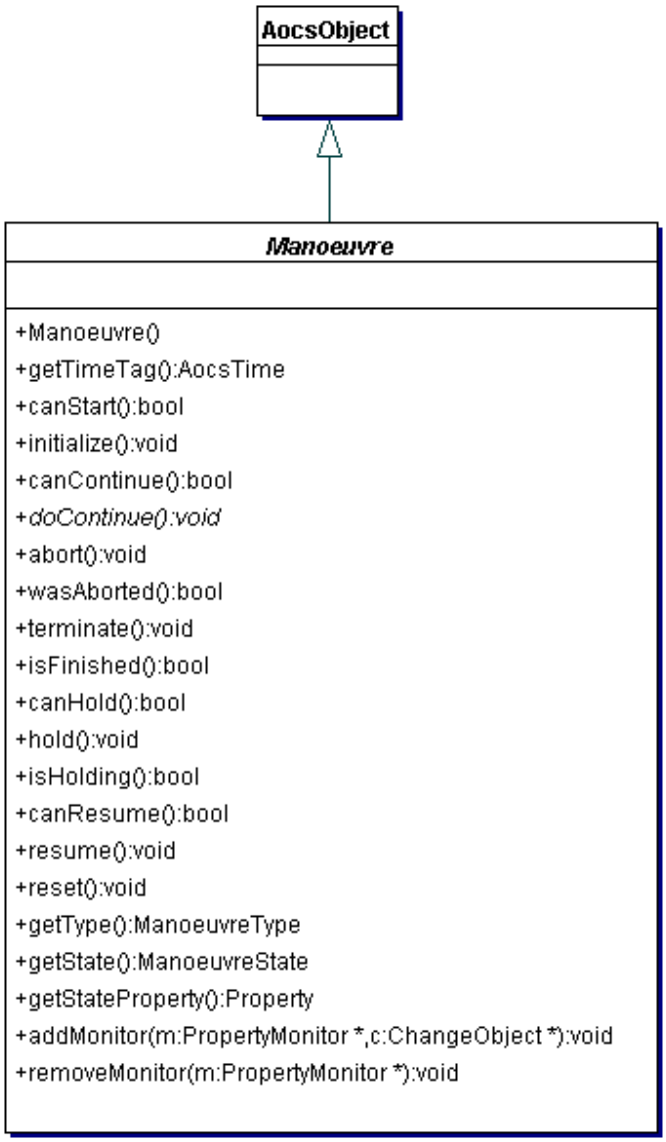
Note that, in principle, clients can only *add* manoeuvres to the manoeuvre manager list. Removal from the list is done autonomously and internally to the manoeuvre manager when a manoeuvre has terminated its execution.

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

- The manoeuvre manager is implemented as an active object and its `activate` method is the `run` method declared by interface `Runnable` (see section 8).
- The manoeuvre manager is given responsibility for creating events that record the beginning and end of a manoeuvre and other manoeuvre-related occurrences (see section 7).
- The manoeuvre base class is implemented with some additional functionalities such as manoeuvre abort or manoeuvre hold (see section 6).
- The execution status of manoeuvres is implemented as a bound property (see section 6).

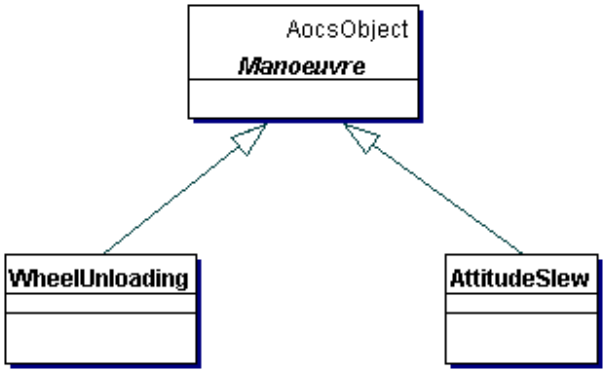
6 MANOEUVRE OBJECTS

In order to allow their uniform and mission-independent treatment, manoeuvres are encapsulated in objects. The base class for manoeuvre objects is:



(Note that, as is often true with UML class diagrams, not all operations are shown in the above diagram. Only those operations which need to be highlighted for the purposes of the present discussion are shown.)

Manoeuvre is an abstract class providing default – and mostly trivial – implementations of its methods. Concrete manoeuvres are instances of subclasses of `Manoeuvre` as shown in the following class diagram:



`WheelUnloading` and `AttitudeSlew` are two examples of concrete manoeuvres.

Manoeuvres are loaded in a manoeuvre manager that is responsible for correctly executing them. The manoeuvre manager uses the methods exposed by class `Manoeuvre` to control manoeuvre execution. The methods of the Manoeuvre class are described in the following table.

`getTimeTag()`	
	Manoeuvres may carry a time tag defining the time after which they can be considered for execution. This method is provided to retrieve a manoeuvre's time tag.
`canStart()`	
	Reaching the time tag is not necessarily enough to trigger execution of a manoeuvre. Sometimes, other conditions have to be satisfied to ensure that the manoeuvre can execute safely. For instance, an attitude slew should only start if the spacecraft angular rates are below a pre-defined threshold. This method is provided to check these kind of peripheral conditions. It should return true if the AOCS context is such as to make execution of the manoeuvre safe. The manoeuvre manager will only start manoeuvre execution if `canStart` returns true.

`initialize()`
Manoeuvres may have to perform some special initialization action when their execution begins. This method encapsulates this type of action. It is called by the manoeuvre manager the first time the manoeuvre is executed.
`canContinue()`
Manoeuvre execution should only proceed if external conditions are safe. For instance, a manoeuvre that performs 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. The manoeuvre manager can call this method to ask the manoeuvre to verify if the conditions for its continued execution hold. If they do, then method `doContinue` can be called and the manoeuvre execution is advanced. If they do not, the manoeuvre is aborted. After a manoeuvre is aborted it is cancelled from the list of pending manoeuvres and will no longer be executed.
`doContinue()`
Manoeuvres execute over a prolonged period of time. This method is called by the manoeuvre manager to advance the execution of the manoeuvre. When a manoeuvre object receives this command, it retrieves the current AocsTime and performs any actions that are due for execution at that time. This is an abstract (or pure virtual) method and must be defined in concrete subclasses of Manoeuvre.
`abort(), wasAborted()`
The `abort` method is called to allow a manoeuvre to perform any clean-up actions before it is descheduled. The manoeuvre manager calls `abort` when a manoeuvre must be aborted. This may happen either because the manoeuvre itself declares that it is unable to continue execution (i.e. `canContinue` returns false) or because the manoeuvre manager autonomously decides to abort the manoeuvre. Method `wasAborted` returns true if the manoeuvre has been aborted.

`terminate(), isFinished()`
The method `isFinished` returns true when the manoeuvre has terminated. When the manoeuvre manager detects that the manoeuvre has terminated execution, it calls `terminate` to give the manoeuvre the chance to perform any final closedown actions and then de-schedules it.
`canHold(), hold(), isHolding(), canResume(), resume()`
The manoeuvre manager may wish to temporarily interrupt execution of a manoeuvre. The manoeuvre manager can call the method `canHold` on the manoeuvre to determine whether or not execution of the manoeuvre can be temporarily interrupted. If the manoeuvre manager decides to interrupt execution of the manoeuvre, it should call the method `hold` on the manoeuvre to allow it to take any action required in connection with a suspension of the manoeuvre. Similarly, the manoeuvre manager can request from the manoeuvre and indication of whether or not its execution can be resumed by calling the method `canResume`. When the manoeuvre execution is resumed, method `resume` should be called. The manoeuvre manager can determine whether or not the manoeuvre is currently suspended by calling the method `isHolding`.
`reset()`
Resetting a manoeuvre is similar, but not quite identical, to aborting the manoeuvre. Resetting a manoeuvre, which is done by calling the `reset` method of the manoeuvre, can be thought of as a more gentle form of aborting the manoeuvre. Resetting a manoeuvre causes the manoeuvre's `canContinue` method to return false. When the `canContinue` method returns false, the manoeuvre manager responds by aborting the manoeuvre. More information about resetting a manoeuvre is provided in the Reset Interface section below.

Note that manoeuvres are executed by the manoeuvre manager. Thus, for instance, a call to method `hold` does not actually cause the manoeuvre to hold since only the manoeuvre manager can decide to suspend manoeuvre execution. The method is provided to *inform* the manoeuvre that it is being held so as to give it chance to perform any actions that are required to ensure that the AOCS remains safe and that the manoeuvre can be correctly resumed at a later time. Calls to methods `terminate`, `resume`, and `abort` serve similar purposes.

The remaining methods shown in the diagram are discussed in subsequent sections.

6.1 Manoeuvre Type

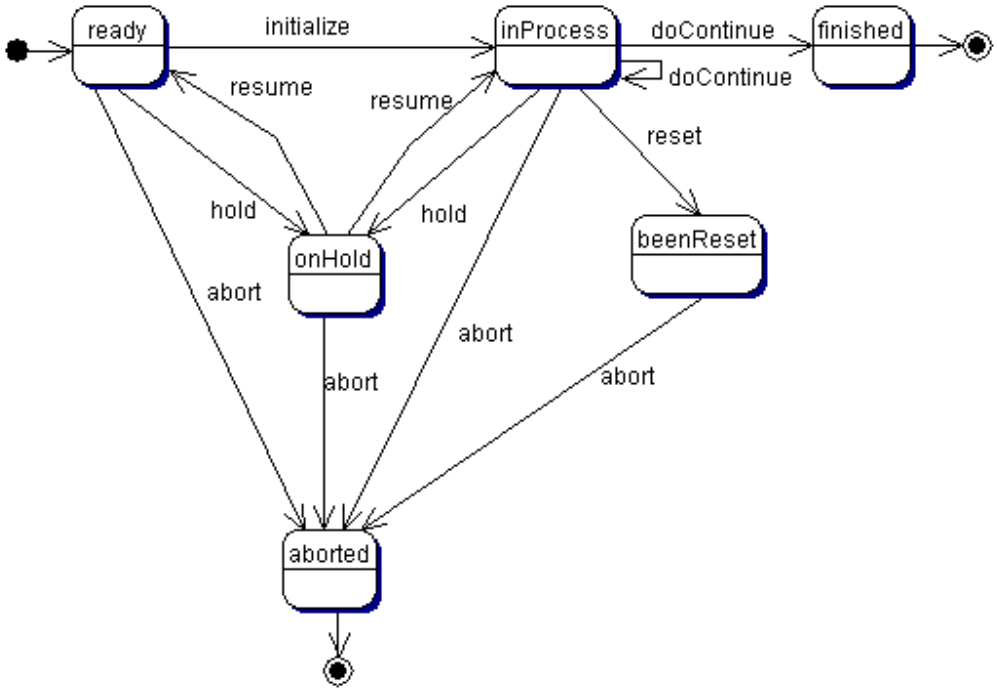
Manoeuvres have a type attribute. Functionally similar manoeuvres have the same type. Thus, all reaction wheel unloading manoeuvres or all attitude slew manoeuvres belong to the same type. The type of a manoeuvre can be retrieved by calling method `getType`.

In the current implementation, the type of a manoeuvre coincides with its class and therefore the implementation of method `getType` is the same as that of method `getClassIdentifier`. The type `ManoeuvreType` is therefore identical with the type `IdType`.

Generally speaking, the manoeuvre manager allows operations to be performed either on individual manoeuvres or on all manoeuvres of the same type. In the former case, the manoeuvre to be operated upon is specified by giving a reference to its manoeuvre object. In the latter case, the manoeuvre type is specified by giving the manoeuvre type as an instance of type `ManoeuvreType`.

6.2 Manoeuvre State

Each manoeuvre has a current state. The possible manoeuvre states are: `ready`, `inProcess`, `finished`, `aborted`, `onHold`, and `beenReset`. The transitions that can be made between these states are shown in the following UML statechart diagram.



In the diagram, the transition names match the names of methods that cause the transition from one state to another. Note that when the `doContinue` method is invoked, the manoeuvre may remain in the inProcess state or transition into the finished state. Also notice that transitioning out of the finished or aborted state is not possible.

In addition to using such methods as `wasAborted`, `isFinished`, and `isHolding` to determine a manoeuvre's state, there is a more generic method available called `getState`.

A manoeuvre's state is an *internal property* of the manoeuvre as defined in RD8. Thus direct access to the property is provided by the `getState` method and access to the property as a property object is provided by the `getStateProperty` method.

6.3 Event Generation

A PropertyMonitor can monitor an individual manoeuvre's state. When a manoeuvre's state changes, the manoeuvre is responsible for notifying any registered property monitors of its state change. This notification is carried out via the creation of a monitored property change event and the invoking of a monitor's `propertyChange` method. This follows the "Monitoring Through Change Notification" pattern described in RD8. Thus manoeuvres provide the methods `addMonitor` and `removeMonitor` as required by that pattern.

If the manoeuvre encounters a failure, this – like all failures – must be reported as a failure event. Creating the failure event and storing it in the failure event repository is the responsibility of the manoeuvre itself. Note that manoeuvre objects inherit a setter method for the failure event repository from `AocsObject`.

6.4 Telemetry Interface

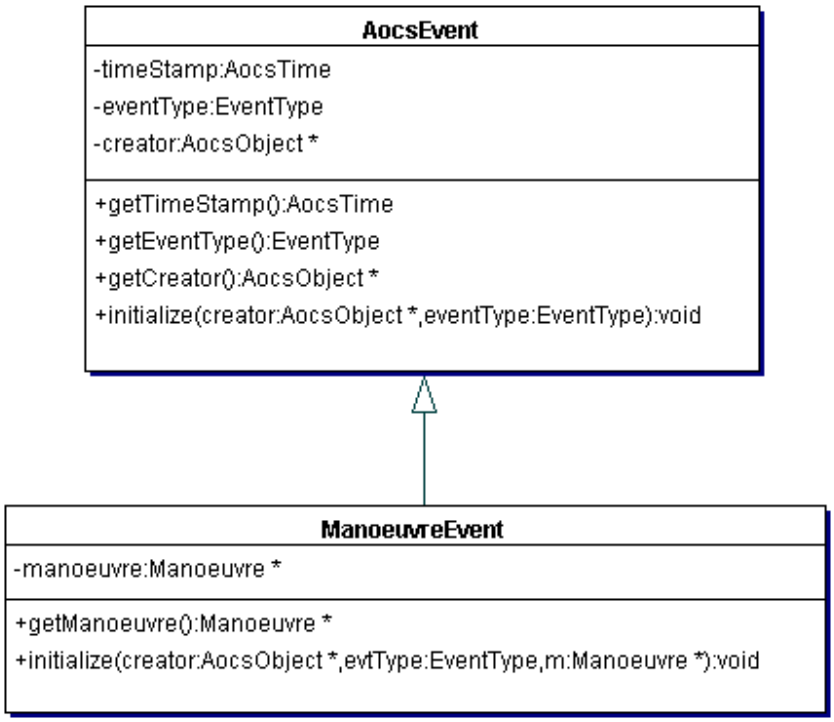
Manoeuvre objects inherit the `Telemeterable` interface from `AocsObject`. The implementation of method `writeToTelemetry` is, however, specific to each concrete manoeuvre class.

6.5 Reset Interface

Manoeuvre objects inherit the `Resettable` interface from `AocsObject`. The implementation of method `reset` is, however, specific to each concrete manoeuvre class. Note that in general a manoeuvre that has been reset while it is executing will not be able to continue its execution (i.e. its `canContinue` method will return `false`).

7 MANOEUVRE EVENTS

The inter-component communication framelet identifies – without defining – a class dedicated to recording manoeuvre-related events. The manoeuvre event class is defined as shown in the following class diagram:



The reference `manoeuvre` points to the manoeuvre object that occasioned the creation of the event.

Thus, a manoeuvre event stores the following information:

- time stamp identifying the time when the event was detected
- event type identifier (see below)
- reference to the event creator (usually, the manoeuvre itself)
- reference to the manoeuvre to which the event is related

Except for the last item, all other data items are inherited from the base class `AocsEvent`.

The following manoeuvre-specific event types are foreseen:

- manoeuvre execution has started

- manoeuvre has been put on hold
- manoeuvre has been resumed
- manoeuvre has been aborted
- manoeuvre has terminated

Note that manoeuvre execution failures are not recorded since manoeuvre failures – like any other type of failure – are reported as failure events (see section 6.3).

7.1 The Telemetry Interface

The manoeuvre event is a telemetry object because it (indirectly, through `AocsEvent`) inherits from `AocsData` the `telemeterable` interface.

Only one telemetry format, *normal*, applies to recovery events. Normal telemetry prints the identifier of the manoeuvre object.

7.2 The Reset Interface

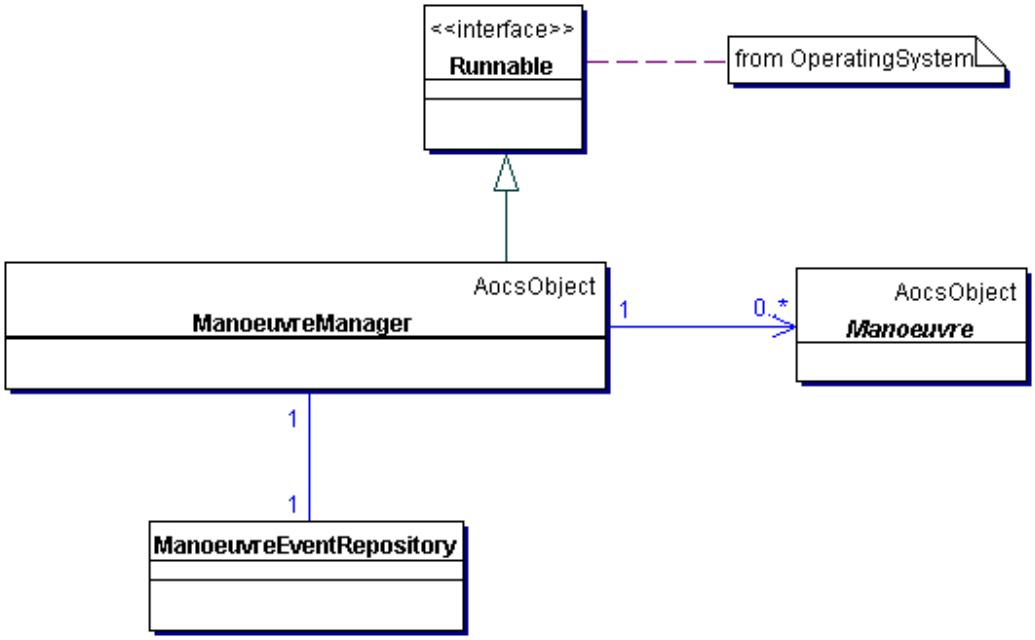
The manoeuvre event (indirectly, through `AocsEvent`) inherits from `AocsData` the `resettable` interface and must therefore implement the corresponding method.

Method `reset` on manoeuvre events simply causes the settable fields in the event to be set to zero.

8 MANOEUVRE MANAGER

The manoeuvre manager is an active component that is responsible for controlling execution of manoeuvres. Manoeuvres are *loaded* into the manoeuvre managers and from that moment onward their execution, termination, holding, and resumption remain under the control of the manoeuvre manager.

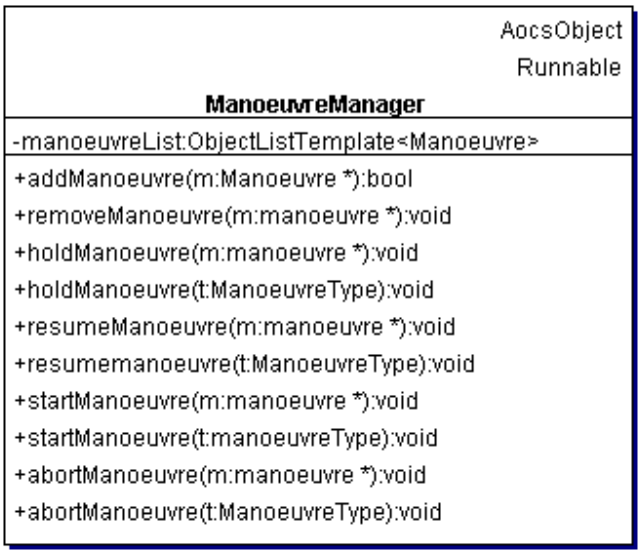
The manoeuvre manager class is shown in the following class diagram:



The manoeuvre manager inherits from `Runnable` to signify that it is an active object. The currently loaded manoeuvres are maintained as a list of references to `Manoeuvre` objects.

The manoeuvre manager maintains a link to the manoeuvre event repository. This link is not currently used. But a possible future enhancement is to add notifications of events that involve any manoeuvre in a particular group. In the case that this enhancement is made, it may become the responsibility of the manoeuvre manager to create and store events reflecting these group operations. See section 8.1 for more information.

The operations exposed by the manoeuvre manager are shown in the following diagram:



The methods of the ManoeuvreManager class are described in the following table.

`addManoeuvre(), removeManoeuvre()`
Operations `addManoeuvre` and `removeManoeuvre` are used to load and unload manoeuvres. Currently executing manoeuvres cannot be unloaded (their execution has to be aborted first). Both methods could potentially fail: the manoeuvre list can be full or the manoeuvre to be removed may not be in the list or may be currently executing. Each method reports such failures using the standard event mechanism described in RD3. That is, it creates a configuration error event and stores it in the appropriate event repository.

`holdManoeuvre(), resumeManoeuvre()`
Manoeuvres can be held and resumed by calling `holdManoeuvre` and `resumeManoeuvre`. Two versions of each of these methods exist. The first version operates on a specific manoeuvre; the second version operates on all manoeuvres of a certain type. The version that operates on a specific manoeuvre creates and stores configuration error events if the specified manoeuvre is not currently loaded in the manoeuvre manager or if the specified manoeuvre is not currently in a state in which it can be put on hold (as determined by the `canHold` method.) The version that operates on all manoeuvres of a specified type simply puts "on hold" all manoeuvres of the specified type which are both loaded in the manoeuvre manager and can be put on hold. Corresponding `resumeManoeuvre` methods, one for individual manoeuvres and one for all manoeuvres of a specified type, are also available.
`startManoeuvre()`
Manoeuvres normally start when their time tag has been reached and when the conditions for their initiation (as defined by method `canStart`) are satisfied. However, authorized clients of the manoeuvre manager can force the start of a manoeuvre by calling `startManoeuvre`. Two versions of this method are provided. One operates on individual manoeuvres; the other operates on all manoeuvres of a specified type. Note that the `canContinue` test is not overridden by calling `startManoeuvre`. Execution of a manoeuvre is continued only if its `canContinue` method returns true regardless of whether the manoeuvre was started by the manoeuvre manager or whether manoeuvre was forced to start by an external object calling `startManoeuvre`.
`abortManoeuvre()`
Manoeuvres are normally aborted in response to method `canContinue` on the manoeuvre object returning false. This can be overridden through use of the method `abortManoeuvre` on the manoeuvre manager. Again two versions of the method are available: one that forces an individual manoeuvre to abort and one that aborts all manoeuvres of a specified type.

8.1 Possible Future Enhancement to Manoeuvre Monitoring

Execution and termination of manoeuvres are important events in an AOCS with system-wide implications. In particular, some components may want to change their mode in response to changes in the execution status of manoeuvres. As is described in sections 6.2 and 6.3, notification is achieved through the generic property monitoring mechanism. The execution status of a manoeuvre has been encapsulated in a property that can then be subjected to monitoring through change notification.

However, one possible addition to this mechanism is the ability for an object to be notified of manoeuvre related events that involve any member of a group of manoeuvres, for example any manoeuvres of a specific type.

This feature is not available in the current implementation. Adding this feature in the future will likely involve creating a ManoeuvreGroup class and adding methods to the manoeuvre manager to allow an object in the system to register its interest in being notified about events which occur involving any manoeuvre which is a member of a ManoeuvreGroup. The ManoeuvreGroup concept is similar in nature to the ThreadGroup concept in the Java thread mechanism.

8.2 Telemetry Interface

The manoeuvre manager inherits the `Telemeterable` interface from `AocsObject`.

Only the `normal` telemetry format applies to the manoeuvre manager.

The manoeuvre manager writes to the telemetry stream the identifiers of the currently loaded manoeuvres and their execution status.

Calls to `writeToTelemetry` are not propagated to the manoeuvre objects as manoeuvre objects are considered to be external to the manoeuvre manager (the manoeuvre manager's relationship to manoeuvres is one of mere association with its manoeuvres).

8.3 Reset Interface

The manoeuvre manager inherits the `Resettable` interface from `AocsObject`. A call to its method `reset` will cause currently executing manoeuvres to be aborted and all manoeuvres to be unloaded.

9 FRAMELET HOT-SPOTS

This section classifies the framelet hot-spots defined in the previous sections of this document. The classification is as described in RD9.

9.1 Manoeuvre Hot-Spot

Name: **Manoeuvre Hot Spot**
Visibility Level: framework-level
Adaptation Time: compile-time
Adaptation Method: derivation from base class `Manoeuvre`
Pre-defined Options: AttitudeSlew manoeuvre component exported by this framelet
Related Hot-Spots: none
Description The abstract base class `Manoeuvre` implements all the basic mechanisms of a manoeuvre but leaves several hooks for manoeuvre specific behavior. In particular, subclasses of Manoeuvre must override the pure virtual method doContinue to perform the actions associated with the specific manoeuvre. Subclasses can override other methods of the `Manoeuvre` class (including, but not limited to: canStart, initialize, canContinue, abort, terminate, hold, resume, and reset) to implement manoeuvre specific behaviors. Implementation of a concrete manoeuvre thus requires derivation of a class from `Manoeuvre`. The derived class must, at a minimum, provide an implementation for method `doContinue`. Additionally, it may be necessary to override the reset and telemetry related methods to send some manoeuvre specific data to the telemetry stream.

9.2 Manoeuvre State Change Handler Plug-In

Name: **Manoeuvre State Change Handler Plug-In**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `Manoeuvre` class (method `addMonitor`)
Pre-defined Options: none. By default, no handlers are associated with a change of manoeuvre state.
Related Hot-Spots: none
Description Manoeuvres expose their state as a monitorable property. Monitors can register their interest in monitoring the manoeuvre state and can ask to be notified whenever the state change in accordance with certain time profile as specified by a `ChangeObject` object.

9.3 Manoeuvre Event Repository Plug-In

Name: **Manoeuvre Event Repository Plug-In**
Visibility Level: framelet-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `Manoeuvre` class (method `setManoeuvreEventRepository`)
Pre-defined Options: `ManoeuvreEventRepository` component exported by inter-component communication framelet.
Related Hot-Spots: none
Description Manoeuvres log changes in their execution state as events stored in the manoeuvre event repository. This hot-spot allows the manoeuvre event repository component to be loaded. Note that this component is loaded as a `static` reference.

9.4 Change Event Repository Plug-In

Name: **Change Event Repository Plug-In**
Visibility Level: framelet-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `Manoeuvre` class (method `setChangeEventRepository`)
Pre-defined Options: `ChangeEventRepository` component exported by inter-component communication framelet.
Related Hot-Spots: none
Description Manoeuvres whose execution state is being monitored by other components log changes in the state as events stored in the change event repository. This hot-spot allows the change event repository component to be loaded. Note that this component is loaded as a `static` reference.

10 FRAMELET FUNCTIONALITIES

This section defines the functionalities offered by the framelets together with their mutual relationships and their mappings to framelet architectural constructs. The definition follows the guidelines of RD9.

10.1 Conventions

The functionality code defines the type of the functionality according to the following convention:

- CF = can-functionality
- DF = do-functionality
- OF = offer-functionality

The following numbering conventions are used:

- if Fx is a functionality, then the functionalities that are obtained by expanding it are numbered as Fx.n where n is 1, 2, 3, etc
- if CFx is a can-functionality, then the offer-functionality that implement it are numbered OFx,n where n is 1, 2, 3, etc

10.2 Functionality List

The functionalities for the manoeuvre management framelet are shown in the table below. Each entry covers one functionality giving its definition, its relationships to other functionalities (if any) and its mappings to framelets architectural constructs (if any).

CF1	The manoeuvre management framelet can manage and execute an arbitrary set of manoeuvres
	expands-to *DF1.1 and CF1.2*
DF1.1	The manoeuvre management framelet implements an algorithm for executing a set of concurrent manoeuvres
	is-implemented-by *the ManoeuvreManager component*
CF1.2	The manoeuvre management framelet can implement any manoeuvre

	matches *the Manoeuvre Hot Spot (and thus the Manoeuvre component)*
CF2	The manoeuvre management framelet can take any action in response to a significant event in a manoeuvre's life cycle
	matches *the Manoeuvre State Change Handler Plug-In (and thus the ManoeuvreEvent component)* ***uses*** *DF2.1*
DF2.1	The manoeuvre management framelet reports significant events in each manoeuvre's life cycle so that they may be responded to appropriately
	The Manoeuvre component ***uses*** *the Monitoring through Change Notification functionality of the object monitoring framelet*
DF3	The manoeuvre management framelet provides operations to dynamically hold, resume, abort, and reset an active manoeuvre.
	is-implemented-by *the hold, resume, abort, and reset operations in the ManoeuvreManager component*
DF4	The manoeuvre manager reports configuration errors detected at run-time.
	The ManoeuvreManager component ***uses*** *event repository functionalities of the component communication framelet*
DF5	Each manoeuvre reports configuration errors detected at run-time.
	The Manoeuvre component ***uses*** *event repository functionalities of the component communication framelet*