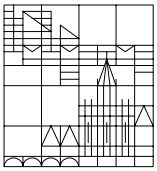


University of Constance
Department of Computer Science

OPERATIONAL MODE 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 operational mode management framelet. This framelet proposes an architectural solution to the problem of managing operational mode changes in AOCS objects. It enhances reusability because it separates the implementation of mode-specific algorithms from the mode switching logic.

Written By: A. Pasetti

Date: 30 April 2002

Issue: 2.3

Reference: SWE/99/AOCS/009

TABLE OF CONTENTS

1 REFERENCES

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

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

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

2 ACRONYMS

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

3 INTRODUCTION

This document describes the *operational mode 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 operational mode changes in AOCS objects.

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

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 mode management.

The architecture proposed here follows the concept outlined in RD2 based on a design pattern for the implementation of mode-dependent components.

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.
- The core mode manager in the C++ framework exposes the current operational mode as a property. Property objects do not exist in the Java framework. The core mode manager is instead implemented as a monitorable component (it implements interface `Monitorable`).
- The mode event repository hot-spot (section 11.6) and the change event repository hot-spot (section 11.7) 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 tool of the *Together* tool.

4 FRAMELET CONSTRUCTS

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

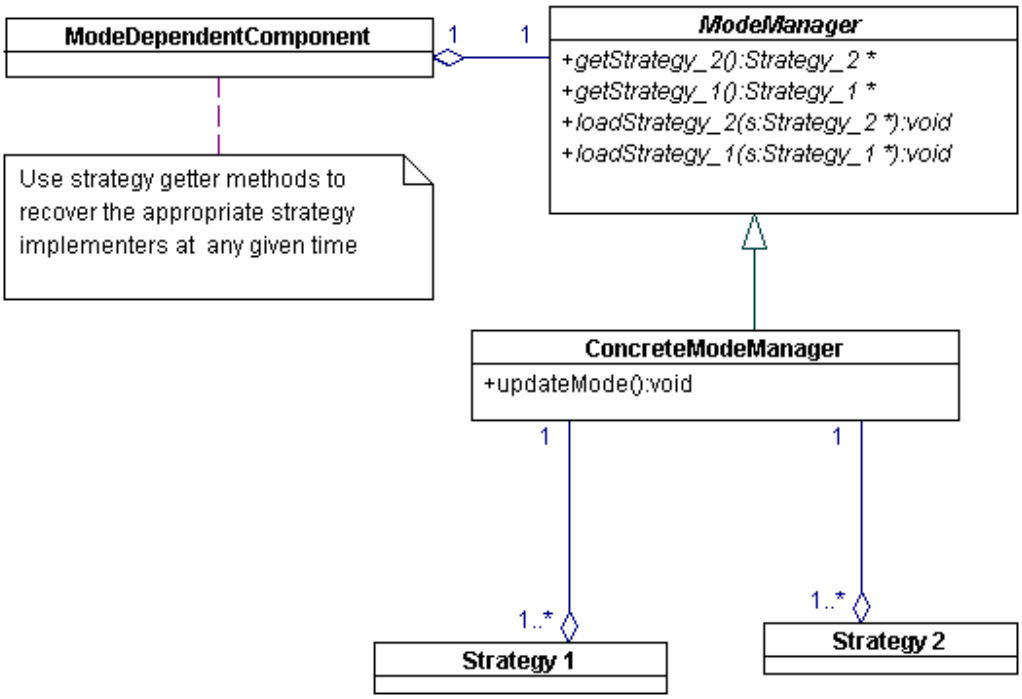
OPERATIONAL MODE MANAGEMENT FRAMELET
Framelet Design Pattern
Mode Management Pattern : pattern to endow components with mode-dependent behaviour
Framelet Core Components
`AocsMissionModeManager` : AOCS mission mode manager `ModeManager` : core mode manager component `ModeChangeAction` : encapsulation of a mode change action
Framelet Default Components
`CyclingModeManager` : cycling mode manager component `FollowerModeManager` : follower mode manager component `NullModeChangeAction` : default mode change action that does nothing

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

5 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` and `Strategy_2`. 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 (method `updateMode`). 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` and `Strategy_2`.

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.

Note that the term *strategy implementer* here is used in a very broad sense. It can refer to an object implementing a certain type of algorithm but it can more generally refer to an instance of a particular class of objects.

5.1 Mode Representation

A mode-dependent component is at any given time in a unique operational mode. The type of the mode indicator could be either an integer or an enumeration type. To each class of mode-dependent components, there may correspond a dedicated type for the mode indicator. Thus, operational mode indicators of different components may not be type-compatible.

To each operational mode, there corresponds a unique set of strategy implementations.

AOCS's sometimes have *submodes* as well as *modes*. The AOCS framework makes no specific provisions for handling submodes. This is primarily because submodes can easily be represented as distinct modes and secondarily because it is believed that submodes in current AOCS's are required because operational mode is treated as an attribute of the entire AOCS. It is postulated that, by treating mode as an attribute of individual components, the AOCS framework removes the need for submoding.

5.2 Mode Manager Attributes

A concrete mode manager is characterized by:

- *The set of modes*

  There are as many modes as there are different implementations that the mode manager can supply for each of the strategies for which it is responsible.

- *The number of strategies*

  This is the number of distinct strategies (as opposed to the number of distinct implementations of each strategy) that are managed by the mode manager. In the example of RD2, there is only one strategy (the computation of the control torque). In the example of the controller implementation case, there are instead three strategies (corresponding to the sensor, actuator and control law objects).

- *The references to the strategy implementations*

  The mode manager maintains references to the objects implementing the strategies for which it is responsible but is insulated from strategy implementation details.

- *The mode-switching logic*

This is the algorithm that is used to decide which implementation of the strategies should be returned at a certain time.

5.3 Mode Switching Logic

The mode switching logic is highly component-specific. In general, mode switches occur in response to changes in the environment around the mode manager. Mode managers can monitor their environment in three manners:

- By directly monitoring properties exposed by other objects;
- By registering with other objects to be notified of specific changes in their properties;
- By inspecting the event repositories for the occurrence of specific events.

In order to allow coordination of mode changes among mode managers, the mode indicator is always exported by mode managers as a bound property.

An AOCS mission manager objects acts as a proxy for the ground (of for a higher level entity within the spacecraft like the OBDH computer) within the AOCS. Its operational mode – the mission mode – can be monitored by mode managers that need to take ground (or OBDH) state into account when performing mode switches.

5.4 Operational Mode Control

Mode switches nominally occur autonomously under the control of the mode manager. Mode managers, however, expose methods that to allow the autonomous mode switching logic to be overridden. In particular, methods are made available to:

- set the current mode
- force the mode to a fixed value
- to inhibit / enable transitions to a specific mode

Because of their potentially disruptive effect, such operations should be used with the utmost care.

5.5 Mode-Based Components in the AOCS Framework

The following objects are endowed with mode-dependent behaviour in the AOCS framework:

- The telemetry manager
- Attitude and orbit controllers
- The unit trigger objects

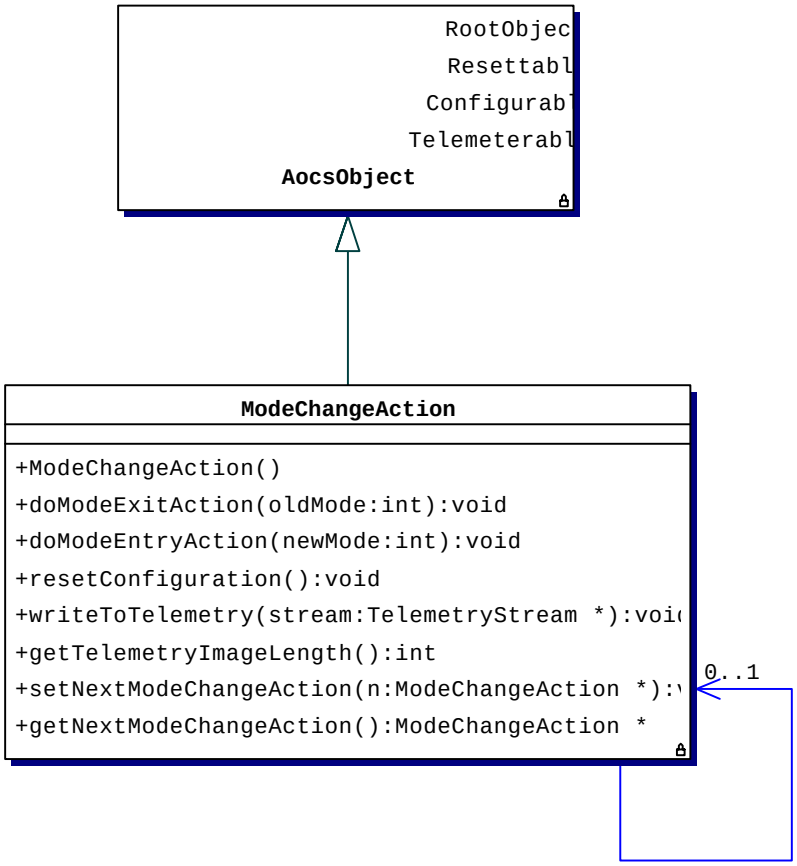
- The failure recovery manager
- The failure detection manager

Readers are referred to the framelet documentation for a description of how the mode management design pattern is instantiated in each case.

6 MODE CHANGE ACTIONS

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.

A second, more systematic approach, is to use *mode change action objects*, namely objects that are instantiated from classes derived from the following base class:



A mode change action object then encapsulates the actions that are associated to a mode transition. The semantics of the methods specific to class `ModeChangeAction` is:

`doModeExitAction(int oldMode)`
Performs the actions associated to a transition out of mode `oldMode`.
`doModeEntryAction(int newMode)`
Performs the actions associated to a transition into mode `newMode`.
`setNextModeChangeAction(ModeChangeAction* n), getNextModeChangeAction()`
Mode change actions can be linked together in a chain of responsibility (see below). These are the getter and setter methods for the next mode change action in the chain.

Mode change action objects should be associated to a mode manager and the mode manager should call method `doModeExitAction` when a mode is exited and method `doModeEntryAction` when a mode is entered. See section 8.3 for an example of how this is done in the case of the core mode manager.

Mode change actions can be linked together in a chain of responsibility (in the sense of RD1). When the mode manager performs a mode transition, it calls methods `doModeExitAction` and `doModeEntryAction` on a `ModeChangeAction` object but the method calls might actually be passed along a chain of linked mode change action objects. In a typical configuration, each mode change action object in the chain specializes in one the transition into or out of one particular mode.

6.1 Recursion

Use of the chain of responsibility pattern introduces the possibility of recursion in the calls to methods `doModeExitAction` and `doModeEntryAction`. The maximum depth of the recursion is given by the maximum number of mode chain actions that are chained together.

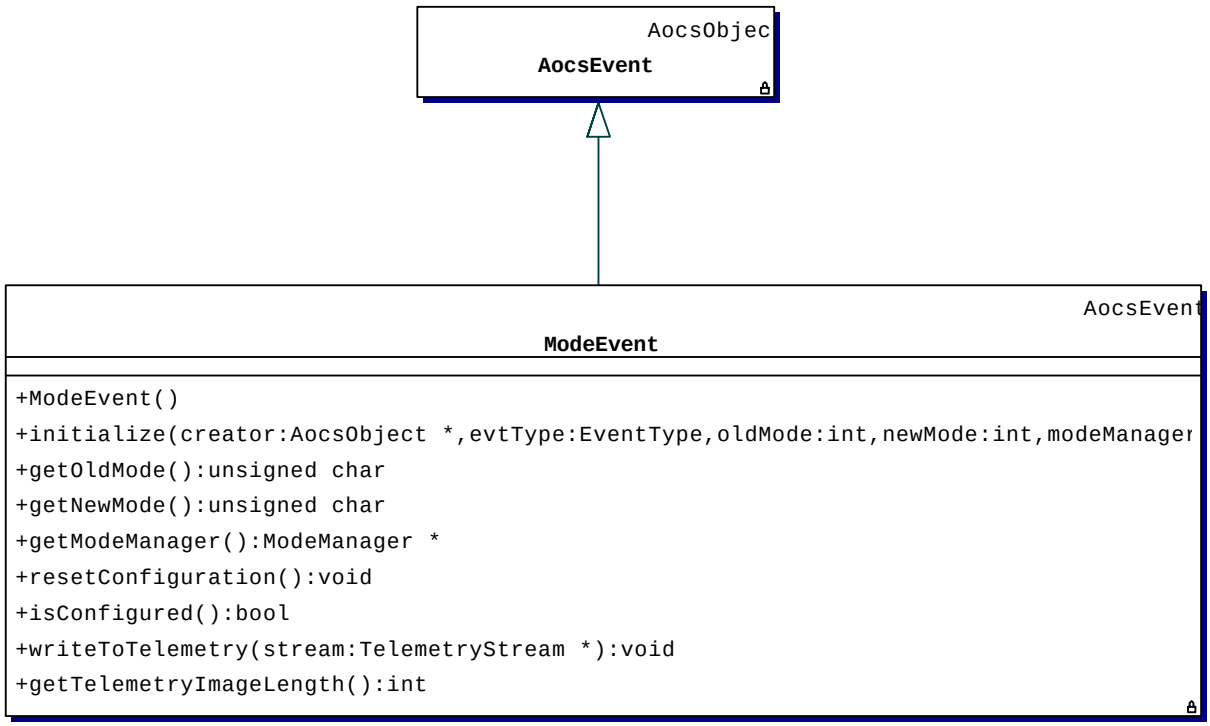
6.2 Default Mode Change Actions

Mode change actions are obviously application-specific (they are one of the hot-spots of the AOCS framework) and therefore the only default mode change action class provided by the AOCS framework is `NullModeChangeAction` that defines a mode change action object that does not do anything. This object to configure mode managers where no special actions need to be taken when a mode transition occurs.

7 MODE CHANGE EVENTS

Changes in the operational mode of a component are recorded as events of type `ModeEvent`. Mode events are stored in a dedicated event repository instantiated from class `ModeEventRepository`.

The class diagram for the mode event class is shown in the next UML diagram:



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

- `oldMode` : the mode before the mode transition.
- `newMode`: the mode after the mode transition
- a reference to the mode manager that underwent the mode transition

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

Mode changes events are created for reporting purposes only. They provide a vehicle through which mode changes can be recorded for possible reporting to the ground in the telemetry stream. Components that need to observe mode changes should do so through the property monitoring mechanism, *not* through inspection of the mode change repository.

7.1 The Telemetry Interface

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

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

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

7.2 The Reset and Configurable Interface

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

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

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

8 THE CORE MODE MANAGER

It is not possible to give a generic implementation of a mode manager for two reasons. Firstly, the mode switching logic is highly mission specific (although one common pattern can be recognized and encapsulated in reusable objects – see section 10). Secondly, the number and type of strategies varies from mode manager to mode manager making general treatment impossible[1].

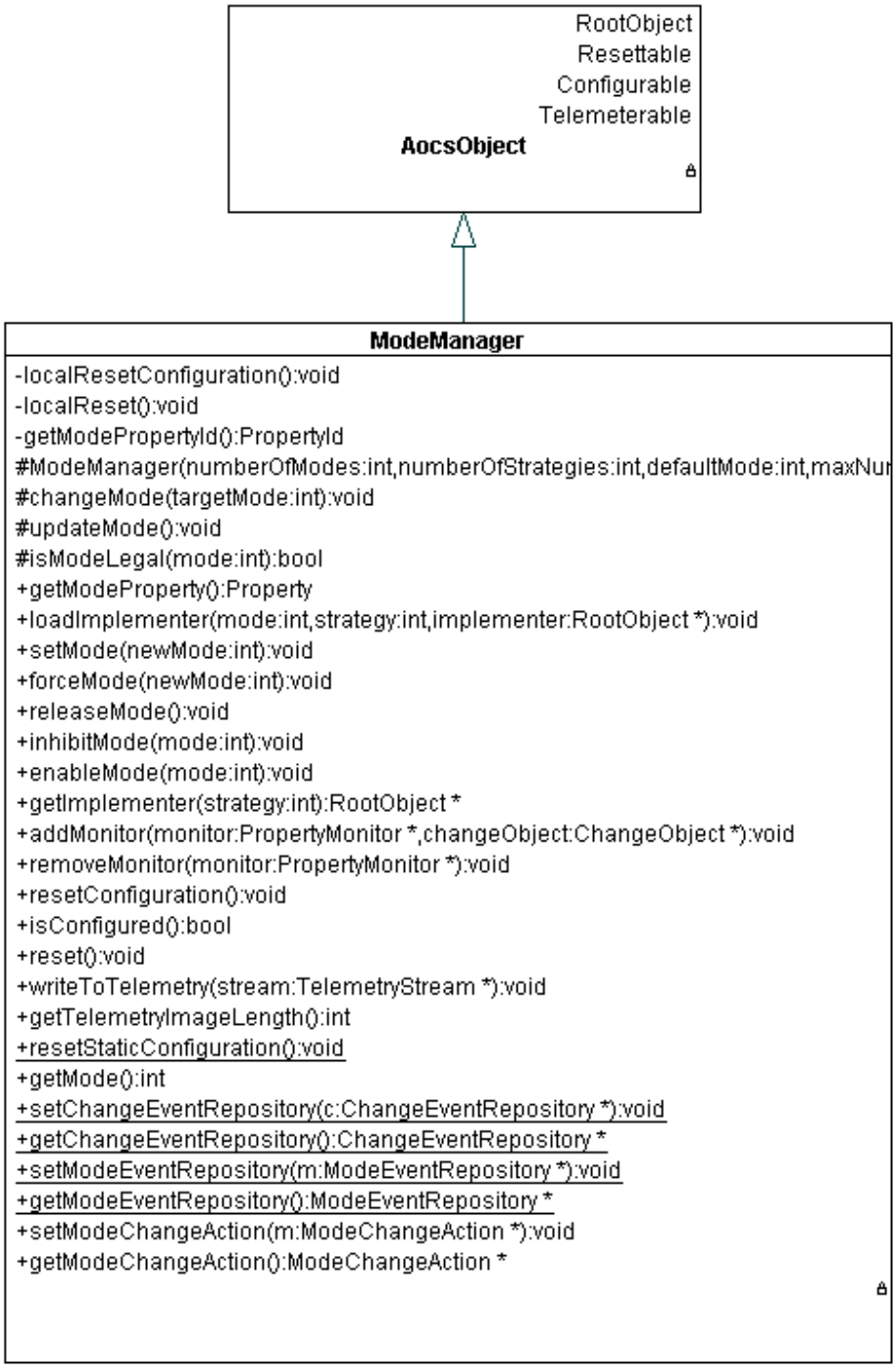
However, there are some functionalities that are common to all mode managers and that can therefore be gathered in a class acting as base class for all mode managers. More specific behaviour can then be implemented by overriding existing methods or adding new ones. The base class for mode managers is called `ModeManager`. It models a *core mode manager* that has no mode switching logic (mode changes occur only in response to external commands) and that handles strategies as references to the basic type `RootObject`.

The core mode manager implements the following functionalities:

- control of mode changes (overriding of autonomous mode changes, constraining of autonomous mode changes, etc)
- management of the mode property as a bound property
- management of the implementers for the mode strategies

Its class diagram is:

[1] General treatment remains impossible even using templates because the number of strategies is not fixed.


RootObject
Resettable
Configurable
Telemeterable
AocsObject
ModeManager
-localResetConfiguration():void
-localReset():void
-getModePropertyId():PropertyId
#ModeManager(numberOfModes:int,numberOfStrategies:int,defaultMode:int,maxNur
#changeMode(targetMode:int):void
#updateMode():void
#isModeLegal(mode:int):bool
+getModeProperty():Property
+loadImplementer(mode:int,strategy:int,implementer:RootObject *):void
+setMode(newMode:int):void
+forceMode(newMode:int):void
+releaseMode():void
+inhibitMode(mode:int):void
+enableMode(mode:int):void
+getImplementer(strategy:int):RootObject *
+addMonitor(monitor:PropertyMonitor *,changeObject:ChangeObject *):void
+removeMonitor(monitor:PropertyMonitor *):void
+resetConfiguration():void
+isConfigured():bool
+reset():void
+writeToTelemetry(stream:TelemetryStream *):void
+getTelemetryImageLength():int
+resetStaticConfiguration():void
+getMode():int
+setChangeEventRepository(c:ChangeEventRepository *):void
+getChangeEventRepository():ChangeEventRepository *
+setModeEventRepository(m:ModeEventRepository *):void
+getModeEventRepository():ModeEventRepository *
+setModeChangeAction(m:ModeChangeAction *):void
+getModeChangeAction():ModeChangeAction *

The public methods specific to the `ModeManager` class (ie not inherited from base classes) are described in the table:

`getModeProperty`	
	Returns the current mode encapsulated as a property object.
`loadImplementer(i,j,&object)`	
	Loads the implementer for the j-th strategy in the i-th mode. The implementer is treated as a reference to an object of generic type `RootObject`.
`setMode(i)`	
	Set the current mode to the i-th mode but does not inhibt further autonomous mode changes.
`forceMode(i), release`	
	Set the current mode to the i-th mode and inhibits further autonomous mode changes. The inhibition is lifted with a call to `release`.
`inhibit(i), enable(i)`	
	Inhibit and enable transition into the i-th mode.
`getImplementer(i)`	
	Return the implementer corresponding to the current mode for the i-th strategy.
`addMonitor(&monitor,&changeObject), removeMonitor(&monitor)`	
	Add and remove the object `monitor` to the list of those monitoring the mode property. Object `changeObject` defines the type of change that triggers a call-back to the monitoring object.
`getModeChangeAction, setModeChangeAction`	
	Setter and getter methods for the mode change action associated to the mode manager. See section 8.3.
`getModeEventRepository, setModeEventRepository`	
	Static setter and getter methods for the mode event repository. Mode changes are logged as mode events in the mode event repository.
`getChangeEventRepository, setChangeEventRepository`	

	Static setter and getter methods for the change event repository. Change events are created when the mode, as a property object, is being monitored and the mode has changed.
`getIllegalModeRecoveryAction, setIllegalModeRecoveryAction`	
	Attempts to operate on a non-existent mode (mode indicator negative or greater than `numberOfMode`-1) give rise to a failure event. These are the setter and getter methods for the recovery action associated to it.
`getIllegalStrategyRecoveryAction, setIllegalStrategyRecoveryAction`	
	Attempts to operate on a non-existent strategy (strategy indicator negative or greater than `numberOfStrategies`-1) give rise to failure event. These are the setter and getter methods for the recovery action associated to it.

Instances of `ModeManager` *cannot* be used as mode managers for framework components because they do not have any mode switching logic and *should not* be used for this purpose because they handle implementers as objects of generic type `RootObject` which would require users to perform potentially dangerous downcasts to the actual type of the implementers.

8.1 Concrete Mode Managers

The most straightforward manner to obtain a concrete mode manager is to subclass `ModeManager` and add the following:

- mode switching logic
- casting operations to allow the user of the mode managers to see the type of the implementers

The above two points are discussed separately in the next subsections.

8.2 Mode Switching Logic

Mode switches can be either triggered by the external commands (through method `setMode`) or they can be initiated autonomously by the mode manager itself.

The logic for autonomous mode switches is contained in method `updateMode`. This is a virtual method offered by class `ModeManager`. Class `ModeManager` provides a default implementation that does not do anything. Derived classes can override this method to provide class-specific mode switching logic.

Class `ModeManager` does not define when `updateMode` is called. In general, three possibilities can be recognized:

- *On-Call Mode Update*

  Method `updateMode` is called whenever `getImplementer` is called. In this case the default implementation of `getImplementer` must be overridden as follows:

  ```
  RootObject* CyclingModeManager::getImplementer(int strategy)
  {
      updateMode();
      return ModeManager::getImplementer(strategy);
  }
  ```

  Thus, mode switches only take place when the client of the mode manager requires a new implementation for one of its strategies.

- *Reactive Mode Update*

  The mode manager acts as a monitor for external properties and changes its mode in response to changes in the external properties. In this case the mode manager must implement interface `PropertyMonitor` and the mode switching logic is entirely implemented inside method `propertyChange`. In this case, method `updateMode` may be unnecessary and may be implemented as a non-op.

- *Periodic Mode Update*

  The mode switching logic is triggered at regular intervals. This is best realized by making the mode manager an active object. Method updateMode is then called from within method `run`.

The mode management framelet provides two components encapsulating default mode switching logic. They are described in section 10.

8.3 Mode Change Action

The core mode manager also takes care of the management of mode change actions. It provides getter and setter methods for the mode change action and it calls its mode entry and mode exit methods upon a mode transition.

8.4 Casting of Implementer Objects

The core mode manager treats implementers as objects of type `RootObject`. Concrete mode managers instead see implementers of specific types. Concrete mode managers therefore

implement dedicated versions of methods `loadImplementer` and `getImplementer` that handle implementers of the appropriate type.

Consider, for instance, a concrete mode manager that uses two strategies with implementers of type `Type1` and `Type2` for, respectively, strategy 1 and strategy 2. Its `getImplementer` methods will be:

```
Type1* getImplementer1()
{
     return (Type1*)ModeManager::getImplementer(0);
}
Type2* getImplementer2()
{
     return (Type2*)ModeManager::getImplementer(1);
}
```

The implementation of the `loadImplementer` methods will be similar.

8.5 Telemetry Interface

The telemetry manager is itself a telemetry object because it inherits (indirectly, through `AocsObject`) the `telemeterable` interface.

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

TM Format	TM Data
Short	none
Normal	current mode indicator
Long	same as normal TM
Debug	same as long TM

8.6 The Reset and Configurable Interfaces

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

Method `reset` resets the current mode to the initial default mode and clears any flags inhibiting mode transitions.

Method `resetConfiguration` unloads all recovery actions and clears all the implementer references.

Method `isConfigured` returns true if implementers for all modes and for all strategies have been loaded.

9 THE AOCS MISSION MODE MANAGER

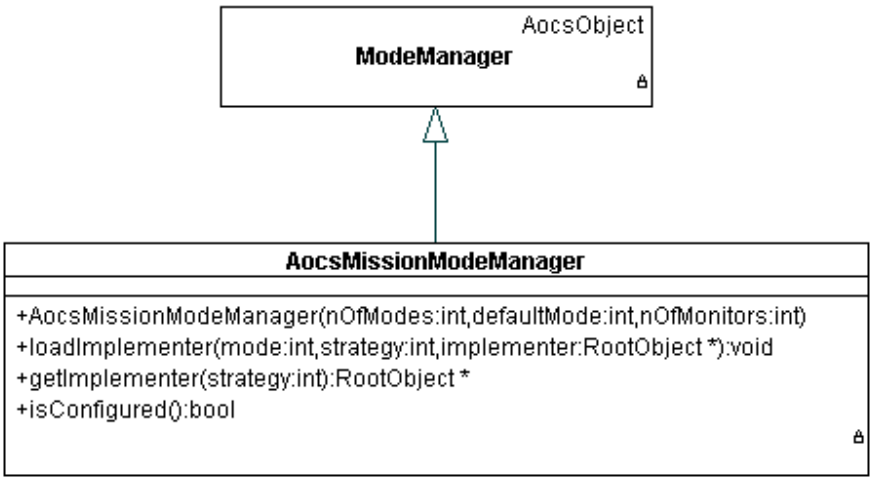
In the model proposed here, components are responsible for managing their own operational mode. They decide on mode changes based on their observation of the external environment. Part of this information can come from monitoring the state of other AOCS objects but part must come from outside the AOCS.

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 Mode Manager* – is provided to supply this information to the rest of the AOCS software. This object encapsulates 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.

Components whose mode is affected by the agents outside the AOCS software should register their interest in mission mode property exposed by the mission mode manager.

The mission mode manager is implemented as a subclass of class `ModeManager` (see previous section):



The mission mode manager basically is a mode manager with no switching logic and without any implementers. It accordingly redefines the methods to load and get the implementers

(methods `loadImplementer` and `getImplementer`) to be dummy methods that return without doing anything.

Since the mission mode manager has no implementers, it has no configuration information and method `isConfigured` is redefined to return `true`.

10 DEFAULT MODE SWITCHING IMPLEMENTATION

Concrete mode managers are derived by inheritance from the core mode manager. One crucial functionality that they must provide is the mode switching logic, namely a mechanism that defines when a mode transition occurs. This logic will often be application-specific. However, there are two types of mechanisms that are likely to recur in many AOCS's and that therefore deserve to be provided as default framelet components. They are described in the next two sub-sections.

10.1 The Cycling Mode Manager

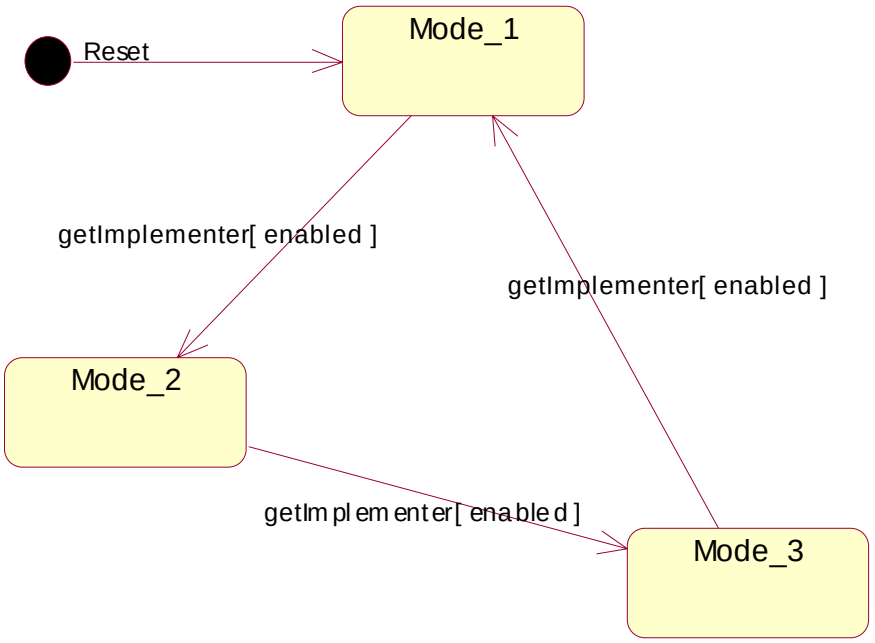
This mode manager is encapsulated by class `CyclingModeManager`. It uses the *on-call mode update* mechanism to cycle through its modes in a fixed sequence. Its implementation of method `updateMode` is:

```
void CyclingModeManager::updateMode()
{
     if (currentMode==(nModes-1))
          . . . // change to mode 0
     else
          . . . // change to mode (currentMode+1)
}
```

where variable `currentMode` is the indicator for the current mode and `nModes` is the number of modes.

Cycling mode managers have only one strategy. A mode transition is triggered whenever the implementer is retrieved (ie. their `updateMode` method is called whenever `getImplemeter` is called). Thus, the cycling mode manager returns its implementers one after the other in a fixed sequence.

A UML state transition diagram for a cycling mode manager with three states is:



The diagram does not show forced transitions (ie transitions commanded by calling `setMode`).

10.2 The Follower Mode Manager

This mode manager is encapsulated by class `FollowerModeManager`. It uses the *reactive mode update* mechanism: its mode is the same as the mode of a master mode manager.

The constructor links the mode manager to the master mode manager. Henceforth, any change in the operational mode of the master will be mirrored in a corresponding change in operational mode of the follower.

The follower mode manager implements interface `PropertyMonitor` and its mode switching logic is contained in method `propertyChange`:

```
void FollowerModeManager::propertyChange(ChangeEvent evt)
{
     int newMode = (int)(evt.getLastValue()));
     . . . // change mode to newMode
}
```

Obviously, master and follower mode managers should have the same set of modes.

10.3 Concrete Mode Managers

Concrete mode managers that use either the follower of cycling mechanisms for mode updates can be derived by inheritance from the default mode managers presented in the two previous subsections. Their implementation only needs to provide the casting operations to convert the references to the implementers to their appropriate types (see section 8.4).

11 FRAMELET HOT-SPOTS

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

11.1 Mode Manager Hot-Spot

Name: **Mode Manager Hot-Spot**
Visibility Level: framelet-level
Adaptation Time: compile-time
Adaptation Method: derivation from base classes `ModeManager`
Pre-defined Options: none
Related Hot-Spots: Cycling Mode Manager Hot Spot, Follower Mode Manager Hot Spot
Description Class `ModeManager` defines the core behaviour of a mode manager but leaves the mode switching logic and the type of the implementers undefined. Concrete mode managers are derived by inheritance from this class. Usually, the methods to be overridden are `updateMode` and the getter and loader methods for the implementers (see sections 8.1 to 8.4).

11.2 Cycling Mode Manager Hot-Spot

Name: Cycling **Mode Manager Hot-Spot**
Visibility Level: framelet-level
Adaptation Time: compile-time
Adaptation Method: derivation from base classes `CyclingModeManager`
Pre-defined Options: none
Related Hot-Spots: Mode Manager Hot Spot, Follower Mode Manager Hot Spot
Description Class `CyclingModeManager` defines a mode manager that uses the *on-call mode update*

mechanism for mode switching. Concrete mode managers that use this same mechanism can be derived by derivation from it. Usually, the methods to be overridden are only the getter and loader methods for the implementers (see section 8.4).

11.3 Follower Mode Manager Hot-Spot

Name: **Follower Mode Manager Hot-Spot**
Visibility Level: framelet-level
Adaptation Time: compile-time
Adaptation Method: derivation from base classes `FollowerModeManager`
Pre-defined Options: none
Related Hot-Spots: Mode Manager Hot Spot, Cycling Mode Manager Hot Spot
Description Class `FollowerModeManager` defines a mode manager that uses the *reactive mode update* mechanism for mode switching. Concrete mode managers that use this same mechanism can be derived by derivation from it. Usually, the methods to be overridden are only the getter and loader methods for the implementers (see section 8.4).

11.4 Recovery Action Plug-In for Illegal Mode

Name: **Recovery Action Plug-In for Illegal Mode**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `ModeManager` class (method `setIllegalStrategyRecoveryAction`)
Pre-defined Options: no recovery action is defined by default
Related Hot-Spots: none
Description

When an operation is attempted on a mode manager using an illegal value of mode indicator, then a failure event is generated. A recovery action can be associated to this failure. This hot-spot allows this recovery action to be loaded.

11.5 Recovery Action Plug -In for Illegal Strategy

Name: **Recovery Action Plug-In for Illegal Strategy**

Visibility Level: framework-level

Adaptation Time: run-time

Adaptation Method: plug-in component in `ModeManager` class (method `setIllegalModeRecoveryAction`)

Pre-defined Options: no recovery action is defined by default

Related Hot-Spots: none

Description

When an operation is attempted on a mode manager using an illegal value of strategy indicator, then a failure event is generated. A recovery action should be associated to this failure. This hot-spot allows this recovery action to be loaded.

11.6 Mode Event Repository Plug-In

Name: **Mode Event Repository Plug-In**

Visibility Level: framelet-level

Adaptation Time: run-time

Adaptation Method: plug-in component in `ModeManager` class (method `setModeEventRepository`)

Pre-defined Options: `ModeEventRepository` component exported by inter-component communication framelet.

Related Hot-Spots: none

Description

Mode managers log mode changes as events stored in the mode event repository. This hot-spot allows the mode event repository component to be loaded. Note that this component is loaded as a `static` reference.

11.7 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 `ModeManager` class (method `setChangeEventRepository`)
Pre-defined Options: `ChangeEventRepository` component exported by inter-component communication framelet.
Related Hot-Spots: none
Description Mode managers whose mode is being monitored by other components log changes in their mode property as events stored in the change event repository. This hot-spot allows the change event repository component to be loaded. Note that this component is loaded as a `static` reference.

11.8 Monitor Hot-Spot

Name: **Monitor Hot-Spot**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in components in `ModeManager` class (method `addMonitor` and `removeMonitor`)
Pre-defined Options: none.
Related Hot-Spots: none

Description Mode managers expose their mode as a bound property. This hot-spot allows other components to register and un-register their interest in the changes in the operational mode of the mode manager using the *Monitoring through Change Notification* design pattern.

11.9 Mode Implementer Hot-Spot

Name: **Mode Implementer Hot-Spot**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `ModeManager` class (method `loadImplementer`)
Pre-defined Options: none
Related Hot-Spots: none
Description Mode managers must be loaded with the implementers corresponding to each (mode, strategy) pair. This hot-spot allows the implementer components to be loaded.

11.10 Mode Change Action Hot-Spot

Name: **Mode Change Action Hot-Spot**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `ModeManager` class (method `setModeChangeAction`)
Pre-defined Options: the framework provides a null mode change action object
Related Hot-Spots: none
Description Mode managers must be loaded with the mode change action object encapsulating the actions to

be taken upon a mode transition.