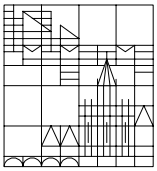


University of Constance
Dept. of Computer Science

INTER-COMPONENT COMMUNICATION 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 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.

Written By: A. Pasetti

Date: 30 April 2002

Issue: 2.3

Reference: SWE/99/AOCS/005

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 SWE/99/AOCS/004

RD3 A. Pasetti (2000), *AOCS Framework – Failure Detection Framelet*, AOCS Framework Document SWE/99/AOCS/010

RD4 A. Pasetti (2000), *AOCS Framework - Failure Recovery Framelet*, AOCS Framework Document SWE/99/AOCS/011

RD5 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 inter-component 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 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 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.

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 overview of the inter-component framelet.

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.
- Data items (section 8) are not used in the Java framework where their function is fulfilled by *data sinks* and *data sources.*
- The Illegal Access Recovery Action Plug-In hot-spot (section 11.8) does not exist in the Java framework since illegal accesses to data items inside AOCS Data are caught by the Java run-time exception mechanism.

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:

INTER-COMPONENT COMMUNICATION FRAMELET
Design Patterns
Shared Data Pattern : pattern to exchange data among components using shared data areas *Shared Event Pattern :* pattern to exchange events among components using shared data areas
Framelet Interfaces and Abstract Base Classes
`AocsEvent` : abstract base class for AOCS events `EventRepository` : abstract base class for event repositories `AocsData` : abstract base class for all AOCS data `DataPool` : abstract base class for AOCS data pools
Framelet Core Components
`TelecommandEvent` : telecommand event `ModeEvent` : mode change event `RecoveryEvent` : failure recovery event `FailureEvent` : failure event `ManoeuvreEvent` : manoeuvre event `ChangeEvent` : property change event `ConfigurationEvent` : configuration error event `SystemEvent` : system event `ReconfigurationEvent` : reconfiguration event `TelecommandEventRepository` : telecommand events repository `ModeEventRepository` : mode change events repository `RecoveryEventRepository` : failure recovery events repository `FailureEventRepository` : failure events repository `ManoeuvreEventRepository` : manoeuvre events repository `ChangeEventRepository` : property change events repository `ConfigurationEventRepository` : configuration error events repository `SystemEventRepository` : system events repository

`ReconfigurationEventRepository` : reconfiguration events repository

`Scalar` : scalar data
`TwoEulerAngles` : set of two Euler angles
`ThreeEulerAngles` : set of three Euler angles
Nvector : set of n elements treated as an n-vector

`AttitudeDataPool` : data pool for attitude data

`DataItemRead` : component encapsulating a read-only access to a data item

`DataItemWrite` : component encapsulating a read/write access to a data item

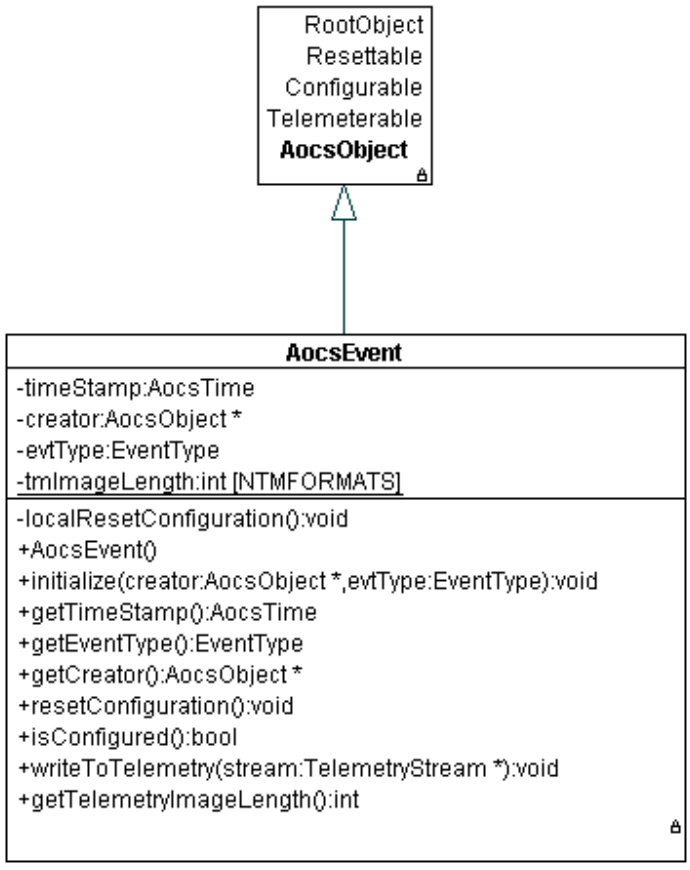
The components listed above are those offered by the prototype version of the AOCS framework. Later version may offer a richer set of default implementations of the framelet interfaces. In particular, they might offer a richer set of AOCS data types.

5 AOCS EVENTS

AOCS events represent asynchronous changes in the state of an object. Examples of meaningful events in an AOCS system include:

- the execution of a telecommand
- the reconfiguration of a unit
- the commencement of a manoeuvre
- the detection of an error
- the execution of a failure recovery action

Events are encapsulated in objects that are derived from the base class `AocsEvent`. Class `AocsEvent` presents the following interface:



Each `AocsEvent` has the following attributes:

- `timeStamp`: read-only attribute representing the time when the event was created.
- `creator`: reference to the object that created the event
- `eventType`: a code identifying the event type.

The event type codes are represented as an enumeration type `EventType`.

Apart from methods inherited from base classes, the `AocsEvent` class provides getter methods for the three above attributes. There is an additional method, `initialize`, that is used to set the creator and event type of an event. The time tag is automatically set by the event. Method `initialize` would typically be called only by the repository containing the event (see section 6).

5.1 Event Categories

The following categories of events are recognized in the AOCS framework:

- *Telecommand Events*

  Record reception of, and successful and unsuccessful execution of telecommands.

- *Failure Events*

  Record the occurrence of a failure detected by the AOCS software during normal operation (eg. failures detected by the failure detection manager).

- *Failure Recovery Events*

  Record the execution of a failure recovery action.

- *Manoeuvre Events*

  Record the loading of, beginning and termination of manoeuvres.

- *Property Change Events*

  Record the occurrence of a property change.

- *Mode Change Events*

  Record the occurrence of a change in the operational mode of a component.

- *Reconfiguration Events*

  Record the occurrence of a reconfiguration.

- *Configuration Error Events*

  Record the occurrence of an error during the configuration of a component.

- *System Events*

  Record the occurrence of a system management events.

5.2 Event Subclasses

Some of the event categories listed in the previous subsection need to store more information than just their own type. An event signaling the execution of a telecommand, for instance, will need to store an identifier of the telecommand. There are two solutions to this problem:

- *Add additional fields to class* `AocsEvent`.

  This solution has the drawback of burdening all events with the extra fields thus resulting in a waste of memory.

- *Create subclasses of* `AocsEvent` *to cater for event categories with greater storage requirements.*

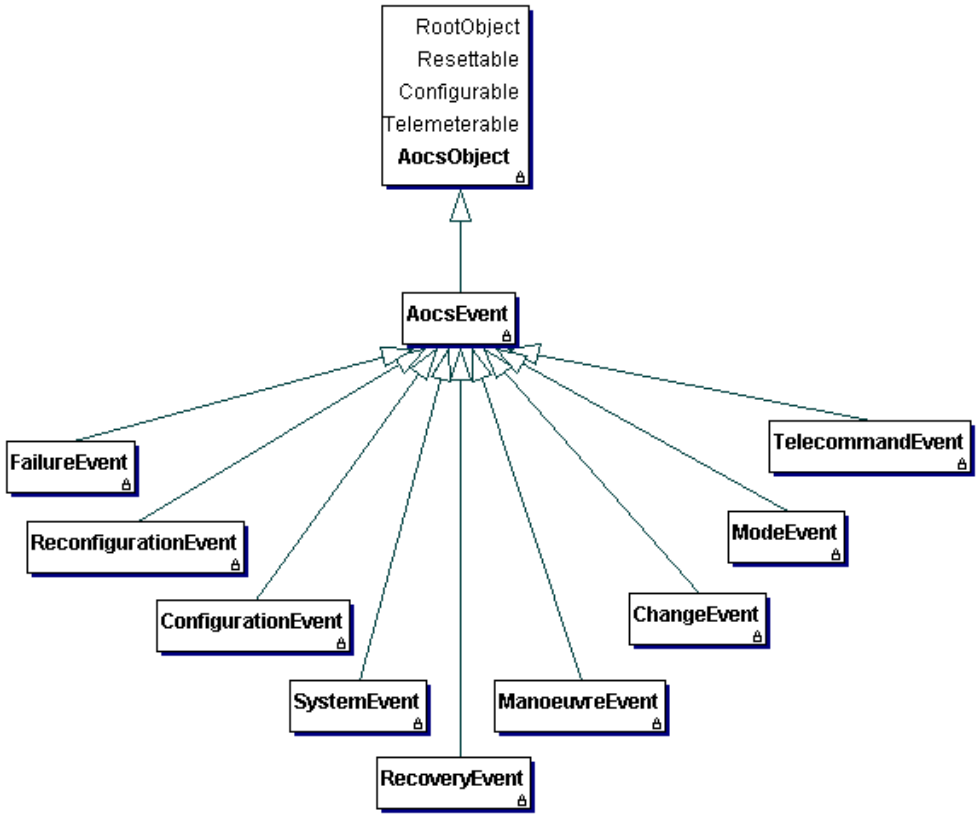
  This solution saves memory (only data that are actually used are stored in each event object) and has the further advantage of making it easier to have event repositories dedicated to each event class.

The second option was selected for the AOCS framework. The internal structure of the event subclasses is defined in the corresponding framelets documents.

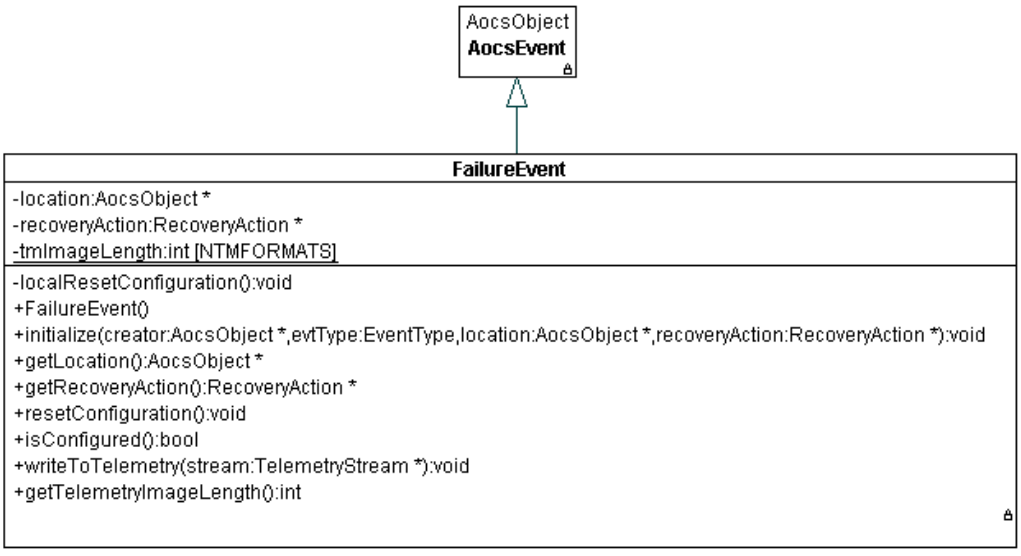
The event subclasses implemented by the framework are listed in the table:

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.

The event subclasses are shown in the following UML diagram:



As an example of an event subclass, consider the subclass for failure events:



Failure events add to basic events a reference to a recovery action associated to the failure and a reference to the object where the failure was detected (the *failure location*). Hence class `FailureEvent` differs from its base mainly for the presence of an `initialize` method that takes a different set of parameters. These parameters are those that are specific to failure events: they include the recovery action and the failure location reference as well as the reference to the event creator and the event type. Additionally, the subclass adds a getter method for the recovery action.

5.3 The Telemetry Interface

Event objects inherit from `AocsObject` the `telemeterable` interface and must therefore implement the corresponding methods.

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

TM Format	TM Data
Short	none
Normal	event type
Long	event type, time stamp
Debug	same as LongTm

5.4 The Reset and Configurable Interface

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

Method `reset` resets all event attributes to zero.

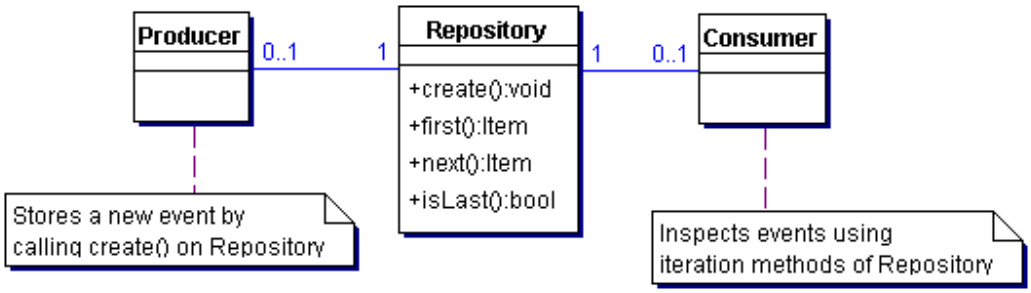
Events have no configuration data associated to them.

6 EVENT REPOSITORIES

Event data are shared among framework component. The shared areas through which events are shared are the *event repositories*.

6.1 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 use the iteration methods to retrieve all the events in the repository and process them as necessary.

6.2 Instantiation of Shared Event Pattern

The `EventRepository` class is introduced to act as the repository through which events are shared between their producers and consumers.

More specifically, an event repository is an object with a double role:

- it acts as a shared data area where events are stored, and
- it acts as a factory of new event objects.

In principle, one could have one single repository for all events, regardless of their event class. However, there are advantages to having dedicated repositories for each event class:

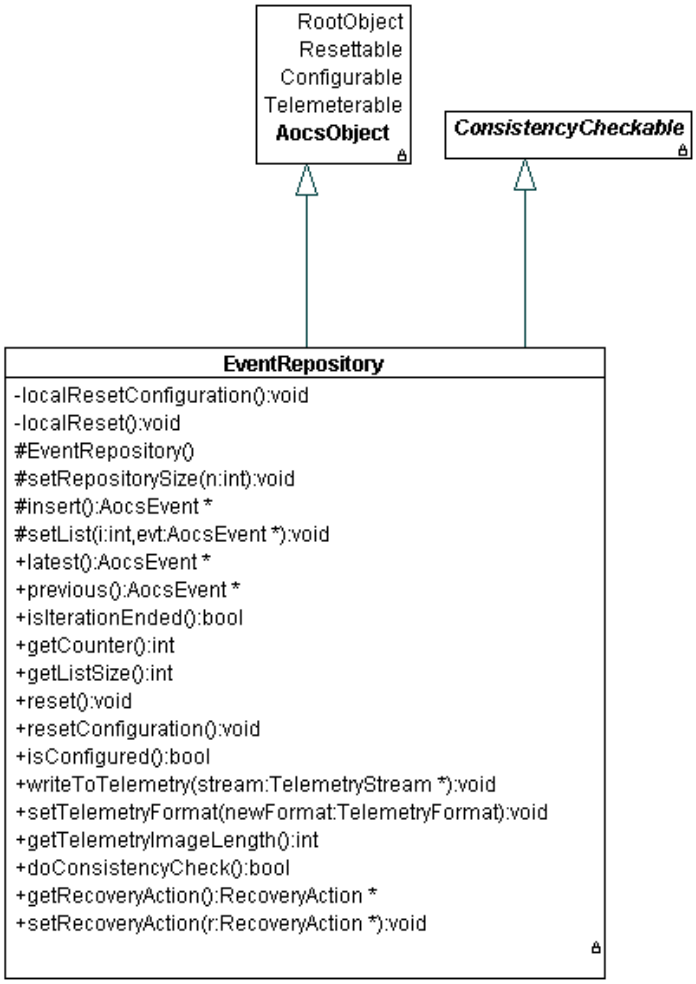
- the number and type of parameters required to create an event vary from class, it is therefore impossible to have a single `create` method to create all kinds of events.

- if dedicated repositories are available, it is easier to subject different classes of events to different treatment. For instance, if it is desired to store in telemetry failure events, and only failure event, this can be done by simply passing the reference to the failure event repository to the telemetry manager.

Hence, in the framework there is an event repository object for each class of events. The following naming convention is adopted: to event class <`eventClass`> there corresponds an event repository called <`eventClassRepository`>.

6.3 Repository Base Class

The base class for event repositories is:



Its public methods not inherited from base classes are described in the table:

`latest, previous, isIterationEnded`	
	Iteration methods that iterate through all the events in the repository starting from the one that was created last. Note that iteration is performed on events that have been *created* in the repository in the sense of section 6.4
`getCounter`	
	Returns the total number of events created in the repository since the last reset.
`getListSize`	
	Returns the capacity of the repository (the maximum number of events it can store).

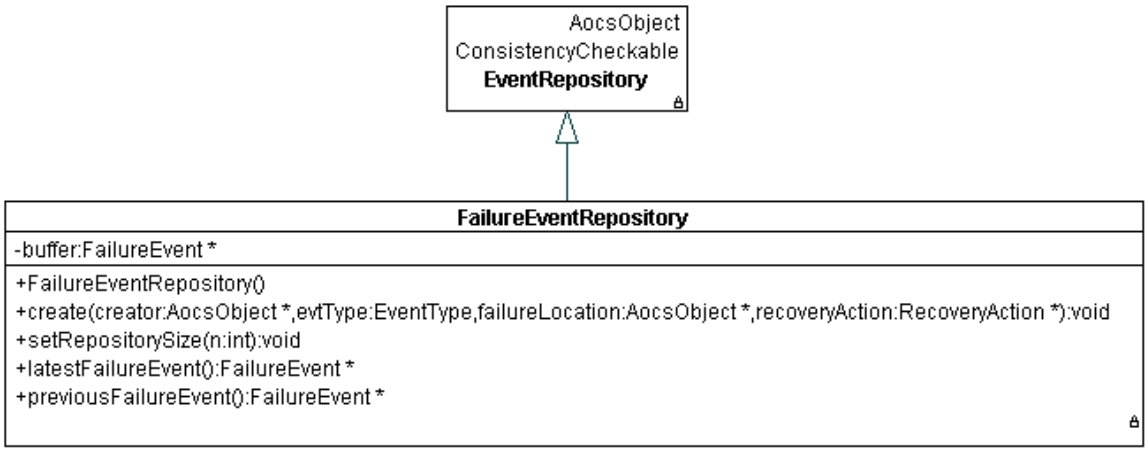
Note that repositories implement the `ConsistencyCheckable` interface (see section 6.7).

6.4 Event Creation

Events by their very nature must be created dynamically. However, in an embedded system, memory dynamically cannot be allocated dynamically. Hence, each event repository pre-allocates memory and exposes `create` methods to be called by clients that need a new event.

Events are created by passing the parameters that define the event. Since different event subclasses have different sets of parameters, the create method is not part of the interface of the base `EventRepository` class. Each repository subclass exposes a `create` method with a signature tailored to the events it contains.

As an example consider the repository for `FailureEvents`:



This class differs from its base mainly in the presence of a `create` method whose signature is tailored to the signature of the `initialize` method of the corresponding event subclass (see section 5.2). A call to this `create` method simply cause an event in the repository to be initialized with the parameters specified by method `create`.

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

The size of the event buffer maintained by each event repository is defined at initialization time by calling method `setRepositorySize`.

6.5 Repository Subclasses

Class `EventRepository` acts as a base class to more specific event repository subclasses. There is an event repository subclass for each event sub class.

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`).

Note that it is not possible to treat repositories as instantiation from templates because of differences in the signature of their `create` methods.

6.6 Use of Events

When a component needs to deposit a new event in a repository, it calls the `create` method of the corresponding repository. This method retrieves memory for the new event from a circular buffer. In practice, this means that it overwrites an existing – but old – event.

If components were allowed to hold references to events in repositories, they would have to be notified of the overwriting of events to ensure that their references are consistent. Such a notification mechanism is judged too complex. The selected approach is as follows:

- Event producers do not get a reference to the event they deposit in the repository: they can ask for the creation of an event of a certain type but the event itself is stored in the repository and they do not have direct access to it.
- Event consumers retrieve events by inspecting the event repositories.

In this manner, references to repository events are not directly passed from component to component and the consistency problem does not arise.

Note that components can still exchange references to events outside repositories as method parameters. This is allowed for the case where the event creator does not know what to do with the event (should it be stored in a repository? In which repository should it be stored? Etc.).

As an example of this situation, consider the case of a change object that detects a change. This situation is to be reported as an event however the change object itself does not know (and should not know) in which repository the event is to be stored. The solution adopted here is:

- The change object creates an event and returns it to the monitor that is responsible for performing the check;
- The monitor copies the event to the appropriate repository.

6.7 "Lost" Repository Events

Event producers create new events which are stored in an event repository. Consumers process events by periodically inspecting the events in the repository. Since the memory in the repository is finite, it may happen that an event is overwritten before its consumer gets to see it. The chances of this happening should be minimized by judicious selection of the repository size but, if it does happen, it should be recognized and flagged.

For this purpose, repositories are made to implement the `ConsistencyCheckable` interface The implementation of method `doConsistencyCheck` checks that the number of events created in between two successive calls to `doConsistencyCheck` are less than the repository capacity. If this is not the case, then a system event is created.

Method `doConsistencyCheck` is normally called by the failure detection manager. If this check is scheduled with appropriate frequency, then it can be used to notify the ground or

some error handler that one or more events have been "lost" because they have been overwritten before having been processed.

6.8 The `ConsistencyCheckable` Interface

Event repository objects implement the `ConsistencyCheckable` and must therefore implement method `doConsistencyCheck`.

The consistency check on event repositories is described in section 6.7.

As usual, a recovery action object can be defined to specify the recovery action to be taken in case the consistency check fails.

6.9 The Telemetry Interface

Repositories inherit from `AocsObject` the `telemeterable` interface and must therefore implement the corresponding methods.

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

TM Format	TM Data
Short	identifier of new events (events not yet reported in telemetry)
Normal	calls `writeToTelemetry` on all new events in the repository
Long	calls `writeToTelemetry` on all new events in the repository
Debug	calls `writeToTelemetry` on all events (new and old) in the repository

6.10 The Reset and Configurable Interface

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

Method `reset` deallocates all events in the repository.

Method `resetConfiguration` clears the recovery action associated to the consistency checkable interface (see section 6.7).

7 PRELIMINARY CONCEPT FOR AOCS DATA

This section discusses a concept for AOCS data that is very attractive in its generality but that appears very difficult to realized and may be incompatible with the constraints imposed by an embedded environment.

7.1 Objective

The cyclical data in an AOCS can be of many concrete types (quaternions, scalars, vectors, etc). It is desirable to have components that process them in a manner that is independent of their concrete type. Thus, ideally, 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. The following subsections explore the feasibility of achieving this effect.

7.2 Replacement of Concrete Type by a Base Type

Consider a hypothetical `Integrator` class containing a function to implement an integrator. This is a very common type of component that will be found in many different AOCS systems. When it operates on `Real` data, a sample implementation of the function:

```
class Integrator_1 {

     Real y = 0;    // output of PI controller
     Real Dt;       // time step probably set by constructor

public:

     Real integrate(Real u) {
          y = y + u*Dt;
          return y;
     }

     // other methods as needed . . .
}
```

Assume now that there is a base type `AocsData` from which all concrete data types are derived and assume furthermore that the basic arithmetic operators have been overloaded for

this type to allow direct addition, multiplication, etc of `AocsData` variables. One naïve way of making `Integrator` type-independent would be as follows:

```
class Integrator_2 {

     AocsData y = 0;    // output of PI controller
     AocsData Dt;       // time step probably set by constructor

public:

     AocsData integrate(AocsData u) {
          y = y + u*Dt;
          return y;
     }

     // other methods as needed . . .
}
```

However, this solution will not work in practice. `AocsData` is a base class. If it is made abstract, the above code will not compile. If it is made a concrete class, the code will compile but will fail at run-time when method `integrate` is passed an argument which is one of the derived classes of `AocsData` (for instance, a `Vector` or a `Scalar`).

A second attempt at making `Integrator` type-independent would rely on the use of pointers rather than objects:

```
class Integrator_3 {

     AocsData* y;       // output of PI controller
     AocsData* Dt;      // time step probably set by constructor

public:

     AocsData* integrate(AocsData* u) {
          (*y) = (*y) + (*u) * (*Dt);
          return y;
     }

     // other methods as needed . . .
}
```

This version of `Integrator` will compile without errors but now problems arise in the overloading of the arithmetic operators for `AocsData`.

The definition of `AocsData` will look like this:

```
class AocsData {

     int format;   // identifier of the concrete data type
     . . .

     int getFormat() { return format;}

     virtual AocsData operator+(AocsData& d) {
          // just a placeholder, dummy definition
     }

     . . .
}
```

The methods in `AocsData` can be either pure virtual or (as in the example above) can contain dummy definitions to be overridden by the derived classes. One example of derived class encapsulating the scalar type is:

```
class Scalar : public AocsData {

     Real value;

     . . .

     AocsData operator+(AocsData& d) {
          . . . // definition of addition on scalars
     }

     . . .
}
```

The problem here is that a natural definition of `operator+` for `Scalar` would take a `Scalar` argument and return a `Scalar` result but in this case both argument and result must be of `AocsData` type to maintain compatibility with the base class.

One could imagine having conversion operators that go from `AocsData` to `Scalar` and vice-versa. The format field that identifies the data type could be used to perform up- and down-casts. The definition of `operator+` for class `Scalar` might then look like this:

```
class Scalar : public AocsData {
```

```
        Real value;

        . . .

        AocsData operator+(AocsData& d) {
             if (d.getScalar() == SCALAR)
                  return Scalar(value+d.value);
             else
                  . . . // error, create failure event
        }

        . . .
    }
```

This code assumes that the value returned by the method as a `Scalar` can be converted to an `AocsData`. Even if this code is accepted by the compiler, it will fail at run-time because the memory requirements of the base type `AocsData` are smaller than those of its derived concrete types.

Another option would be to have `operator+` return a pointer to the result of the addition (as opposed to returning the result itself). The method would then have the following signature:

```
        AocsData* operator+(AocsData& d);
```

Unfortunately, this will make it impossible to chain operations together as in: `(a+b)*c`

Moreover, returning a pointer to the result implies that memory must be dynamically allocated to store the result itself on the heap. This is usually not possible in embedded systems.

7.3 Use of Handles

An alternative solution to the problem of section 7.1 is to operate on a *handle* rather than on the data themselves.

The `AocsData` class is now defined to hold a handle to a datum:

```
    class AocsData {

        Datum* pDatum;     // pointer to concrete data item

    public:
```

```
        AocsData(Datum* p) {pDatum = p;}

        ~AocsData() { delete pDatum; }

        AocsData operator+(AocsData d) {
             return AocsData( (*pDatum)+(*d.pDatum) );
        }

        . . . // definition of other arithmetic operators

    }
```

An `AocsData` variable holds a pointer to a variable of type `Datum`. This is the base class for concrete types.

Users only see `AocsData` variables and can operate directly upon them. The implementation of `AocsData` arithmetic operators then delegates the operation to `Datum`.

`Datum` is defined as follows:

```
    class Datum {

        int format;   // identifier for the concrete type

    public:

        virtual operator new()=0;

        virtual operator delete()=0;

        int getFormat() {return format;}

        Datum* operator+(Datum d)=0;

        . . . // definition of other arithmetic operators
    }
```

Note that now the addition operator returns a *reference* to the result. As discussed above, this means that dynamic memory allocation is required and for this purpose the `new` and `delete` must be overloaded to make memory allocation efficient and predictable (and hence compatible with the constraints of embedded systems).

As an example consider the definition of the `Scalar` concrete type:

```
class Scalar : public Datum {
     Real value;

public:

     . . . // class-specific definition of ‘new’ and ‘delete’

     void set(Real v) {value = v;}

     DataItem* operator+(Datum d) {
       if (v.format == SCALAR)
          Scalar* res = new Scalar; // calls Scalar version of ‘new’
          res.set(value+(Scalar)d.value);
          return res;
       else
          . . . // error! create failure event
     }
}
```

Note that the arithmetic operators defined on type `Datum` return *references* whereas the arithmetic operators defined on type `AocsData` return *values*. Thus, while it is not possible to chain together operations performed on `Datum`, it is possible to do so for operations performed on type `AocsData`.

The memory allocated within `operator+` is released by the destructor of `AocsData`.

Defining new types is easy and can be done by simply adding new subclasses to `Datum`. There are no repercussions on either user's code (which operates entirely on `AocsData` variables) or on existing `Datum` subclasses.

7.4 Assessment

The solution proposed in the last sub-section achieves the stated goal of allowing uniform treatment of data. Users can perform their manipulation on `AocsData` while remaining oblivious to the specific type on which they operate. An integrator can be defined that will work equally well on scalars or on vectors of any dimension.

There are, however, three serious drawbacks to the proposed solution. Firstly, it is clear that not all types can be mixed together in all operations. It is for instance possible to multiply a vector by a scalar but it does not make sense to add a vector to a quaternion. However, correctness of argument matching cannot be checked statically because all operations take arguments of the same, generic, `AocsData` type. Type mismatches can only be discovered at

run-time (the `if` statement in the definition of `operator+` in class `Scalar`). This, in an embedded environment, can be a serious problem because it allows unrecoverable errors to remain undetected until run-time.

This problem could be mitigated by connecting components to their inputs as part of their configuration. Each component would then check that it is connected to an input of the appropriate type. Thus, one would still have to wait until run-time to discover type mismatches but one would also know that their existence would be flagged during initialization and would not unpredictably crash the system during normal operation.

The second drawback of this solution is the overhead due to: indirection introduced by the handle mechanism; need to perform type checking at every operator call; dynamic memory management. Some of this overhead is, however, inevitable: generality of treatment is almost always realized by adding one layer of indirection and is therefore almost always bought at the expenses of greater execution inefficiencies.

The third drawback is the recourse to dynamic memory management. This is usually frowned upon in real-time system because of its unpredictability and because of the danger of memory leaks. The first danger can be overcome by suitably overloading the `new` and `delete` operators but the second one cannot be avoided.

These drawbacks are judged to be unacceptable and the solution proposed here is consequently rejected. This unfortunately means that the objective of generality of treatment of AOCS data without regard to their exact concrete type cannot be achieved in full. Section 9 proposes an alternative architecture that achieves this objective only in part. This is the selected baseline for the AOCS framework.

8 THE DATA ITEM CONCEPT

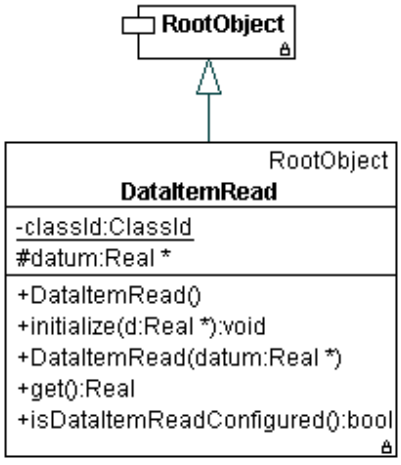
As discussed in the previous section, it is not possible to define a generic `AocsData` type on which arithmetic operations can be directly performed. Hence, components in the AOCS framework that need to perform operations on each other's data have to do so at the level of atomic variables of primitive type.

The term *data item* is used to designate an atomic variable of primitive type that 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 providing respectively read-only and read-write access. The two wrappers are represented by classes `DataItemRead` and `DataItemWrite` presented in the next two sub-sections.

8.1 The `DataItemRead` Class

Class `DataItemRead` encapsulates a reference to a data item and gives read-access to it. Its UML diagram is:



The data item object is constructed around a reference to a datum. The object only provide read access to the datum.

Its non-trivial methods are described in the following table:

`DataItemRead(&datum)`
Constructor that creates the data item object to encapsulate the reference to datum.
`get()`
Returns the value of the datum.
`initialize(Real* d)`
Initializes the reference encapsulated by the `DataItemRead` object. This is useful when the default constructor is used to create the object (eg. when arrays of DataItemRead objects are created).
`isDataItemConfigured()`
Returns true if the data item object contains a non-null reference to a datum. This method allows to identify data items that are "empty" (in the sense that they do not refer to any datum).

Note that it is assumed that the data item is of basic type `Real`. It is used for any data item that either is a `Real` or can be converted to a `Real`.

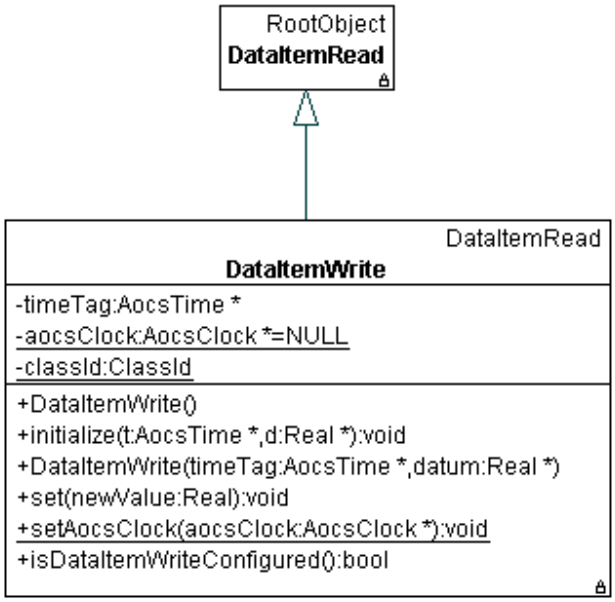
`DataItemRead` objects can be used to establish links between producers and consumers of data. Suppose for instance that component `A` uses a data item `d` from component `B` as an input for its operations. Component `B` will then expose a method `getDataItemRead()` to allow `A` to get a `DataItemRead` object that encapsulate `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. In particular, it neither has nor inherits virtual methods which means that there is no virtual pointer table associated to it.

8.2 The `DataItemWrite` Class

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

Class `DataItemWrite` is defined as shown in the following UML diagram:



Basically, this class adds to the `DataItemRead` class: a reference to the time tag of the data item; a reference to the AOCS clock; and methods to set the encapsulated datum and read its time tag. The time tag is automatically set by the `set` method that uses the AOCS clock to retrieve the current time.

Its non-trivial methods of are described in the following table:

`DataItemWrite(&t, &datum)`
Constructor that creates the data item object to encapsulate the reference to datum and its time tag.
`set(value)`
Sets the new value of the datum. The time tag is automatically set by this method.
`initialize(Real* d, AocsTime* timeTag)`
Initializes the reference encapsulated by the `DataItemWrite` object. This is useful when the default constructor is used to create the object (eg. when arrays of `DataItemWrite` objects are created).
`setAocsClock(&aocsClock)`
Data item write objects needs to have access to the clock so as to be able to set the

	time tag of the datum they encapsulate. This method is used to set the reference to the AOCS clock.
`isDataItemConfigured()`	
	Returns true if the data item object contains a non-null reference to a datum. This method allows to identify data items that are “empty” (in the sense that they do not refer to any datum).

`DataItemWrite` instances are vehicles for setting the value of an internal variable of some object. Access to a data write object gives write access to the underlying datum.

9 BASELINE CONCEPT FOR AOCS DATA

This section presents a concept for AOCS data that does not achieve the generality of that presented in section 7 but which is baselined for its greater efficiency of implementation.

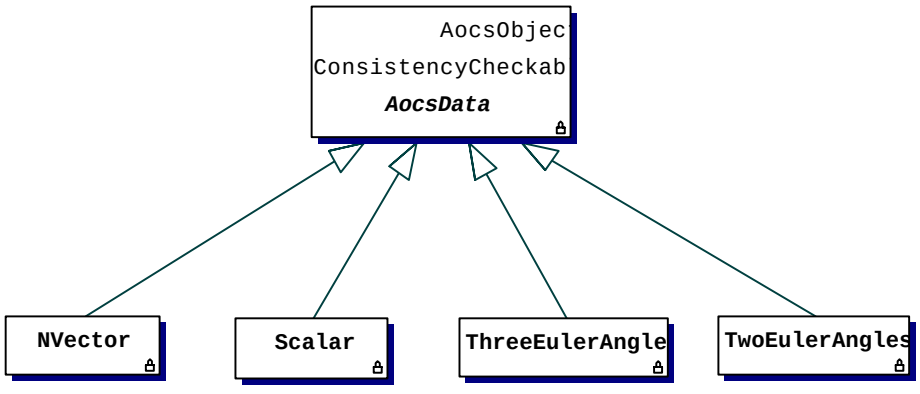
AOCS data present two "faces" to the rest of the AOCS software that reflect the two main purposes for which AOCS components may need to access them. Some components access AOCS data for *housekeeping purposes* like integrity checking, telemetry reporting, failure investigations, etc. Other components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

Section 7 showed that it is not possible to create an abstract computational interface for AOCS data: arithmetic operations must be done on concrete types. They cannot be efficiently done on the abstract type `AocsData` which must therefore be limited to encapsulating housekeeping functionalities.

Consequently, the objective of section 7.1 can only be partially achieved: it is possible to have a interface to AOCS data that is general with respect to housekeeping access but not with respect to computational access. The latter must be done on concrete data items.

9.1 Housekeeping Access to AOCS Data

The class hierarchy for the AOCS data types offered by the prototype framework is shown in the following UML diagram:



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. Obviously, specific missions may add further concrete data types as required.

The `AocsData` base class is:

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

`Distance, close, size, setClosenessThreshold`
Metrics methods, see section 9.2.
`getEarliestTimaTag, getLatestTimeTag, getTimeTAG(i)`
Time tag methods (see section 9.4) returning, respectively, the earliest time tag, the latest time tag and the time tag of the i-th data item.
`normalize, setNormalizationThreshold`
Normalization methods (see sections 9.3 and 9.9).
`getFormat`
Return the AOCS data format (see section 9.7).
`getDataItemRead(i)`
Returns the data item object for the i-th element of the AOCS data.
`getProperty(i)`
Returns the property object for the i-th element of the AOCS data (see section 9.5).
`getDataItemWrite(i)`
Provide write access to the i-th element of the AOCS data.
`getIllegalActionRecoveryAction, setIllegalActionRecoveryAction`
Getter and setter method for recovery action for illegal access failures (see section 9.6).

Note that, except for the data item setter and getter methods, all other methods imply a pure read-only access to the datum and do not require any knowledge of its concrete type. They represent the housekeeping access to the datum and realize the objective of generality of treatment laid down in section 7.1.

The data item methods are used for the computational access to the AOCS data which is discussed in section 9.7.

9.2 Metrics Methods

The AOCS data class defines four methods: `distance`, `equal`, `size` and `close`, that are *metrics* methods. They assume that the concrete data types exist in a space in which a metric can be defined. The basic method is `distance` that computes the distance between two items. In the case of scalars, this distance is simply the absolute value of the difference between the two items. In the case of two equal size vectors, the Euclidean distance is computed. In other cases, type-specific notions of distance may 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 items not of the same concrete type will result in some default 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.

9.3 AOCS Data Normalization

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 as discussed in section 9.9.

9.4 AOCS Data Time Tags

It is generally not possible to associate a single time tag to an `AocsData` since 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.

9.5 AOCS Data Properties

Property objects are associated to each element in an `AocsData`. Methods `getProperty(i)` returns the property object associated to the i-th element.

9.6 Recovery Actions for Illegal Accesses

When methods `getDataItemRead`, `getProperty`, `getDataItemWrite`, or `setDataItemWriteOwner` are called with an illegal argument, a failure event is raised. It is not possible to associate a specific recovery action to each one of these failures. Instead, a generic recovery action is defined for this contingency that can be set with method `setIllegalActionRecoveryAction`.

9.7 Computational Access to AOCS Data

AOCS data are retrieved from their data pool as references to the `AocsData` abstract type. However, as discussed in section 7, computational access to them cannot be done directly on this type. Two alternatives are instead open.

The first and most straightforward option is to use method `getFormat` to obtain the identifier of the underlying concrete type and use it to perform a downcast to the concrete type. Arithmetic operations may be defined on the concrete types (eg. vector operations upon vectors) that allow their manipulation for computational purposes.

This option should be used sparingly both because of the intrinsic danger of downcast and because it circumvents the access control mechanism described below for individual data item.

The second option is to use the data item mechanism to access individual elements in the AOCS data structure.

A component that needs read access to the j-th element of an `AocsData` object can obtain it by calling method `getDataItemRead(j)` that returns a `DataItemRead` variable that encapsulates the desired element.

Similarly, a component that needs read-write access to the j-th element of an `AocsData` object can obtain it by calling method `getDataItemWrite(this, j)` that returns a `DataItemWrite` variable that encapsulates the desired element.

As an example of the access mechanisms based on the data item concept, consider the `ThreeVector` class modelling 3-dimensional vectors. An incomplete implementation for the class is:

```
class ThreeVector : public AocsData {

     Real v[3];                  // elements of 3-vector
     AocsTime timeTag[3];        // element time tags
     . . .

public:

     DataItemRead getDataItemRead(int j) {
          if (j<3)
               return DataItemRead(&v[j]);
          else
               . . .             // failure event
     }

     DataItemWrite getDataItemWrite(AocsObject* caller, int j) {
          if (j<3)
               return DataItemWrite(&timeTag[j], &v[j]);
          else
               . . .             // failure event
     }

     . . .                       // other methods
```

Thus, clients that need to perform computations on the vector obtain access to its elements by calling `getDataItemRead` and `getDataItemWrite`. These methods return instances of `DataItemRead` and `DataItemWrite` that encapsulate references to a vector elements. The data item objects can then be used to establish a permanent link between a component and a location in a data pool.

9.8 Concrete Data Types

The concrete data types provided by the framework prototype are shown in the first figure of section 9.1. Their names are self-explanatory. More concrete types can be added as required.

9.9 The `ConsistencyCheckable` Interface

AOCS data objects implement the `ConsistencyCheckable` and must therefore implement method `doConsistencyCheck`.

A consistency check on an AOCS data fails if the datum is not properly normalized. The check consists in calling method `normalize` and verifying that the return value is smaller than a pre-defined normalization threshold.

Thus, for instance, call of method normalize on a quaternion returns the quadratic sum of the quaternion data members. If it was found that this quadratic on a quaternion representing the satellite attitude after integration of the Euler equations had a value 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 that determines whether or not an AOCS datum is normalized is set by calling method `setNormalizeThreshold`.

As usual, a recovery action object can be defined to specify the recovery action to be taken in case the consistency check fails.

9.10 The Telemetry Interface

`AocsData` variables inherit the `telemeterable` interface from `AocsObject` and must therefore implement the corresponding methods.

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

TM Format	TM Data
Short	none
Normal	values of data items
Long	values of data items and their time stamps
Debug	values of data items and their time stamps

9.11 The Reset and Configurable Interface

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

Method `reset` resets all data items values to zero.

Method `resetConfiguration` perform the following actions:

- clear the recovery action associated to the consistency checkable interface
- set the closeness and normalization thresholds to the default value of 1

9.12 Alternative Implementation

The data item interface in `AocsData` may seem superfluous. One could endow `AocsData` with the following methods:

```
void set(AocsObject* owner, 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.

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 items.

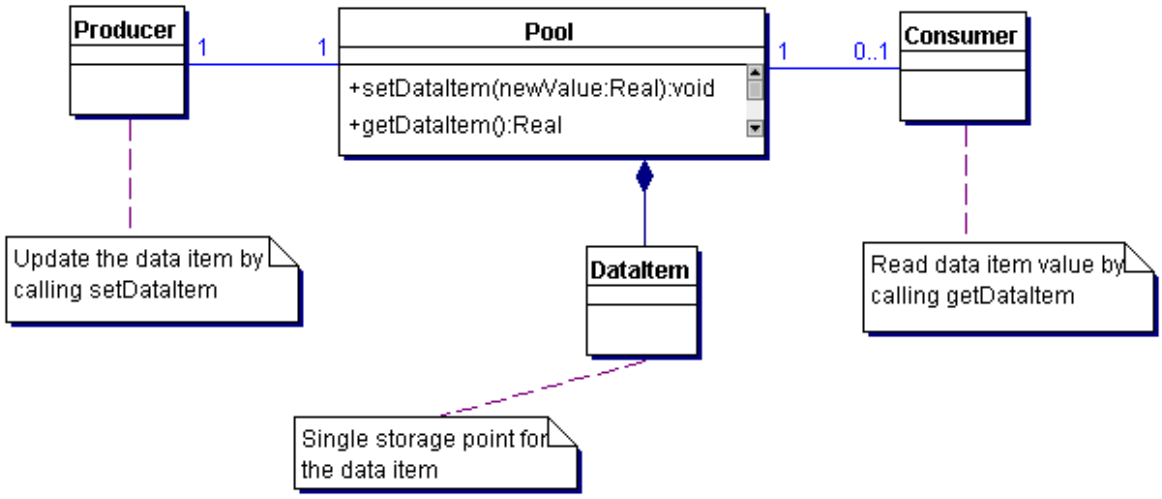
The type `AocsData` is a composite type that gathers together the individual elements of the concrete types it represents. These individual elements may have individual properties. For instance, each element may have its own time tag. For this reason, too, is encapsulation of individual data items useful: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

10 DATA POOLS

Data exchanges among components in the AOCS framework are done through shared data areas called *data pools*.

10.1 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:

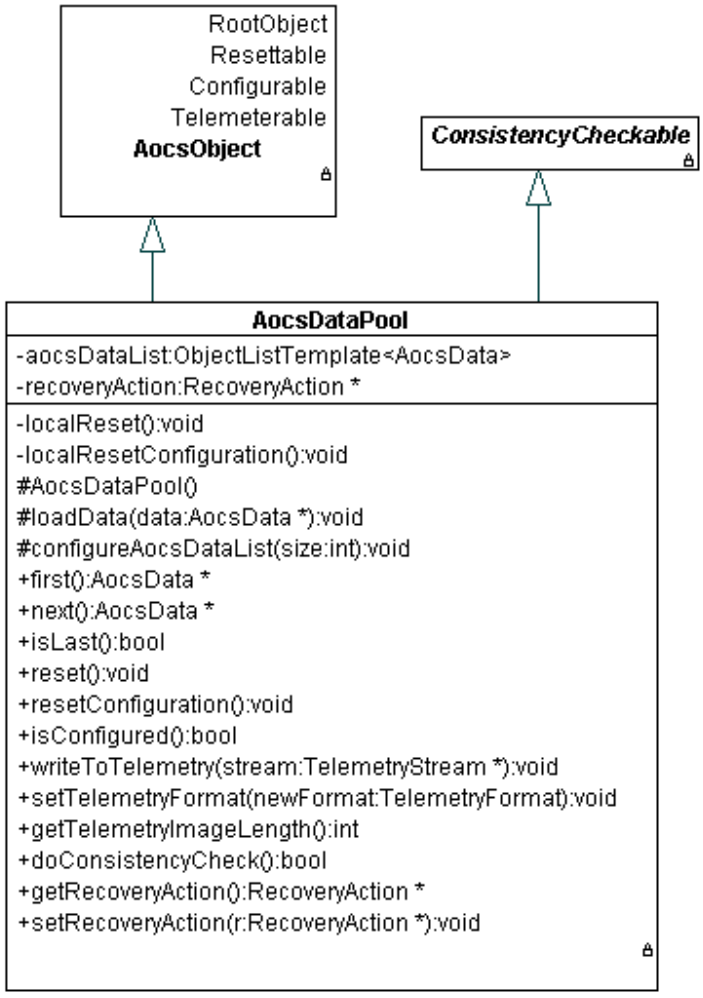


The data pool component contains a single instance of the shared datum. The datum producer (of which there should be only one: the datum owner) calls the setter method to set the datum parameters. The datum consumers can use the getter methods to retrieve the attributes of the shared datum and, if necessary,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

10.2 Instantiation of Shared Data Pattern

In principle, one could have one single data pool for all shared AOCS data. However, the alternative approach of having several data pools grouping together logically related data is preferred as it makes it easier to subject different classes of data to different treatment. For instance, if it is desired to store attitude data in telemetry, this can be done by simply passing the reference to the attitude data pool to the telemetry manager.

Data pools are derived from the following abstract base class:



The public operations that are specific to this class (ie not inherited from any base class) are shown in the table:

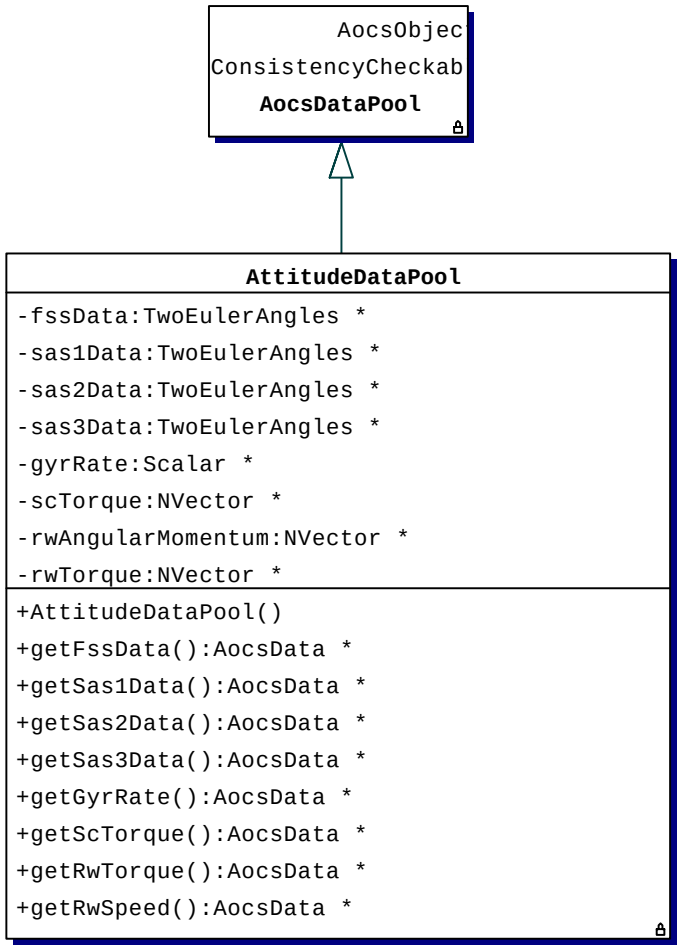
`first, next isLast`
Iteration methods that iterate through the AOCS data in the data pool.

Data pools contain a set of objects that represent the AOCS data in the pool. These data can only be accessed by outside components through the methods exposed by the data pool.

The number and structure of data pools will vary from mission to mission. Different projects will want to store different objects in data pools and will want to group them in different manners. Typical data pools that might be found in an AOCS include:

- `AttitudeDataPool` : contains all the data relative to the spacecraft attitude, including attitude sensor and actuator data.
- `OrbitDataPool` : contains all the data relative to the spacecraft orbit, including orbit sensor and actuator data.
- `SpacecraftDataPool` : contains the spacecraft data base with data such as the spacecraft inertia tensor, its mass, flexible appendage parameters and other physical characteristics

The UML diagram below shows, as an example, one segment of the attitude data pool used for the prototype AOCS instantiated from the prototype AOCS framework:



The concrete data pool adds to the `DataPool` base class getter methods for each AOCS datum contained in the class. Note that the AOCS data are contained in the data pools where they are declared with their concrete type. However, the are accessed through getter method that return references to the generic `AocsData` type.

10.3 The Telemetry Interface

Data pools inherit the `telemeterable` interface from `AocsObject` and must therefore implement the corresponding methods.

Method `writeToTelemetry` on data pools simply iterates on the same method on all AOCS data contained in the data pool.

10.4 The Reset and Configurable Interface

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

Method `reset` on data pools simply iterates on the same method on all AOCS data contained in the data pool. Method `resetConfiguration` does the same and additionally clears the recovery action associated to the `consistencyCheckable` interface.

10.5 The `ConsistencyCheckable` Interface

`DataPool` variables inherit the `ConsistencyCheckable` interface from their base class and must therefore implement method `doConsistencyCheck`.

The method iterates on all the `AocsData` variables in the pool and calls the `doConsistencyCheck` method on each (see section 9.9). If any of these calls reports a consistency check failure, then the consistency check on the data pool is also deemed to have failed.

A recovery action object must be defined to specify the recovery action to be taken in case the consistency check fails.

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 RD5.

11.1 Event Subclass Hot-Spot

Name: **Event Subclass Definition**
Visibility Level: framelet-level
Adaptation Time: compile-time
Adaptation Method: derivation from base classes `AocsEvent` and `EventRespository`
Pre-defined Options: event and event repository components exported by the framelet (see section 4)
Related Hot-Spots: none
Description The base class `AocsEvent` allows only basic attributes to be attached to an event. If more specific data are required for certain event classes, then dedicated subclasses of `AocsEvent` need to be created. A new subclass will generally require definition of getter methods for the class-specific attributes. There should be no need to override any of the methods in the base `AocsEvent` class beside the standard telemetry and reset methods. To each event class, there must correspond an event repository class. Hence, when a new subclass of `AocsEvent` is created, a corresponding subclass of `EventRepository` must also be created. Definition of the new repository subclass will generally require definition of a new `initialize` method. There should be no need to override any of the methods in the base `EventRepository` class beside the standard telemetry and reset methods.

11.2 AOCS Data Subclass Hot-Spot

Name: **AOCS Data Subclass Definition**
Visibility Level: framelet-level
Adaptation Time: compile-time
Adaptation Method: derivation from base class `AocsData`

Pre-defined Options: AOCS data types exported as components by the framelet (see section 4)
Related Hot-Spots: none
Description To each category of AOCS data there should correspond a subclass of `AocsData`. A new subclass will typically require only the protected abstract method `getNumberOfItems` to be defined. In some cases, it may be necessary to override also method `distance` (its default implementation computes the Euclidean distance). Standard telemetry and reset methods must be redefined as usual when subclassing.

11.3 AOCS Clock plug-In for Data Items Hot-Spot

Name: **AOCS Clock Plug-In for Data Items**
*Visibility Level**:*** framelet-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `DataItemWrite` class
Pre-defined Options: none
Related Hot-Spots: AOCS clock plug-in for `AocsObject` class
Description `DataItemWrite` objects attach a time tag to each value they write. They therefore need a clock to provide them with the time. Class `DataItemWrite` offers method `setAocsClock` to define the plug-in clock that is used for this purpose. Note that the link to the clock object is static and hence the clock only needs to be plugged in one instance of class `DataItemWrite`.

11.4 Data Pool Subclass Hot-Spot

Name: **Data Pool Subclass Definition**
*Visibility Level**:*** framework-level
*Adaptation Time: c*ompile-time

Adaptation Method: derivation from base classes `DataPool`
Pre-defined Options: none
Related Hot-Spots: none
Description AOCS data objects are grouped in data pools whose content is highly mission specific. 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.). The derived class does not override any of the methods of its superclass besides the standard telemetry and reset methods and it adds to its base getter methods for all the AOCS data objects it contains.

11.5 Repository Size Hot Spot

Name: Repository Size Hot-Spot
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: setter method for initialization parameter in event repository sub-classes
Pre-defined Options: none
Related Hot-Spots: none
Description Event repositories maintain a pre-allocated buffer of events. The size of this buffer is set when the event repository is initialized by calling method `setRepositorySize` on the repository object. This method can only be called once. Attempts to call it more than once will results in a configuration error event being raised.

11.6 Recovery Action plug-In for Event Repositories

Name: Recovery Action **Plug-In for Event Repositories**
Visibility Level: framework-level

Adaptation Time: run-time
Adaptation Method: plug-in component in `AocsData` class (method `setRecoveryAction`)
Pre-defined Options: no recovery action is defined by default. Standard recovery action components are exported by the Failure Recovery Framelet.
Related Hot-Spots: none
Description Event repository implements the `ConsistencyCheckable` interface. Hence they must provide a plug-in for the recovery action object defining the action to be taken if the consistency check fails.

11.7 Recovery Action plug-In for AOCS Data

Name: Recovery Action **Plug-In for AOCS Data**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `AocsData` class (method `setRecoveryAction`)
Pre-defined Options: no recovery action is defined by default. Standard recovery action components are exported by the Failure Recovery Framelet.
Related Hot-Spots: none
Description AOCS data implement the `ConsistencyCheckable` interface. Hence they must provide a plug-in for the recovery action object defining the action to be taken if the consistency check fails.

11.8 Illegal Access Recovery Action Plug-In for AOCS Data

Name: Illegal Access Recovery Action **Plug-In for AOCS Data**
Visibility Level: framework-level
Adaptation Time: run-time

Adaptation Method: plug-in component in `AocsData` class (method `setIllegalRecoveryAction`)
Pre-defined Options: no recovery action is defined by default. Standard recovery action components are exported by the Failure Recovery Framelet.
Related Hot-Spots: none
Description A failure is reported if an attempt is made to access a non-existent data item or if incompatible data types mixed in calls to metrics method. This hot-spot allows the recovery action associated to this failure event to be defined.

11.9 Recovery Action plug-In for Data Pools

Name: Recovery Action **Plug-In for Data Pools**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `AocsData` class (method `setRecoveryAction`)
Pre-defined Options: no recovery action is defined by default. Standard recovery action components are exported by the Failure Recovery Framelet.
Related Hot-Spots: none
Description AOCS data pools implement the `ConsistencyCheckable` interface. Hence they must provide a plug-in for the recovery action object defining the action to be taken if the consistency check fails.