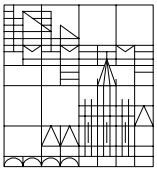


University of Constance
Department of Computer Science

OBJECT MONITORING 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 object monitoring framelet. This framelet proposes an architectural solution to the problem of monitoring an object and its properties. The framelet consists of application-specific design patterns and interfaces.

Written By: A. Pasetti

Date: 30 April 2002

Issue: 2.1

Reference: SWE/99/AOCS/008

TABLE OF CONTENTS

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

1 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

2 INTRODUCTION

This document describes the *object monitoring 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 monitoring an object and its properties. Object monitoring is useful for FDIR purposes, for managing operational mode changes and more generally for helping components synchronize their behaviour.

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

2.1 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 sections dealing with properties, change objects and monitoring.

RD2 outlines a baseline for the handling and monitoring of properties. The architectural solution proposed in the present document follows the conceptual guidelines of RD2.

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

2.2 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.

– Property monitoring is done in a slightly different way. The Java framework does not have property objects (see section 4.2). Monitorable components (components that expose properties that can be subjected to monitoring) are instead characterized by implementation of interface `Monitorable`.

2.3 Notation

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

The class diagrams use UML notation and were generated with the reverse engineering tool of Rational Rose.

3 FRAMELET CONSTRUCTS

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

OBJECT MONITORING FRAMELET
Design Patterns
Property Definition Pattern : pattern to define properties in objects and the methods to access them *Additional Properties Pattern :* pattern to add new properties to a component that is already packaged as a binary unit (not used in prototype framework) *Direct Monitoring Pattern :* pattern to directly monitor an object's property *Monitoring through Change Notification Pattern :* pattern to implement a notification mechanism when a property changes in a specified manner.
Framelet Interfaces and Abstract Base Classes
`ChangeObject` : abstract base class for object encapsulating a type of property change
Framelet Core Components
`Property` : encapsulation of property objects
Framelet Default Components
`SimpleChange` : implementation of interface `ChangeObject` encapsulating a simple change in a property value `OutOfRangeChange` : implementation of interface `ChangeObject` encapsulating an out-of-range change in a property value `DeltaChange` : implementation of interface `ChangeObject` encapsulating a delta change in a property value `SpikeFilteredDeltaChange` : implementation of interface `ChangeObject` encapsulating a delta change in a property value with spike filtering (not implemented in prototype framework)

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. In particular, they may offer a richer set of change objects.

4 PROPERTY MODEL

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

Properties can be accessed either *directly* or indirectly, through *property objects*.

4.1 Property Definition Design Pattern

To each property may be associated getter and setter methods to allow direct access to it. The getter and setter methods follow naming conventions. A property of name <`PropertyName`> and type <`PropertyType`> has getter and setter methods that conform to the following signature and naming pattern:

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

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

Direct access to a property only gives access to the *value* of the property. Controlled access to a *reference* can be obtained through *property objects*.

4.2 Property Objects

Property objects encapsulate a reference to a property and provide a way of identifying the property through a *property identifier*.

Property objects only give *read access* to the property. Some form of controlled write access (using perhaps dynamic access control) could be easily implemented but is regarded as unnecessary in the AOCS context.

Property objects can be implemented in two different manners: either as objects *external to the property owner* or as objects *internal to the property owner*. The framework uses the second option. However the first one is also discussed for future reference.

4.3 Property Objects as External Objects

An abstract base class `Property` is defined from which subclasses are derived to encapsulate specific properties. The definition of class `Property` could be:

```
class Property {
     AocsObject* propertyOwner;
public:
     Real getValue()=0;
}
```

Method `getValue` is implemented in subclasses to recover the property value from the property owner.

The advantage of this approach is that the property owner does not need to know whether its property will be accessed as a property object. It simply has to provide a `get` method and, if access through an object is required, this is arranged externally to the property owner by setting up a suitable property object of type `Property`.

The drawback of this approach is that a subclass of `Property` for each property object must be created. This could lead to a proliferation of small classes and for this reason this option was discarded.

4.4 Property Objects as Internal Objects

This is the option that was selected for the AOCS framework. It relies on the property owner itself returning an object encapsulating a reference to the property giving read access to the property itself. It might seem that `DataItemRead` objects can serve this purpose. However, as they stand, they are unsuitable to act as property objects because they are anonymous: there is no way to identify the property from its `DataItemRead`. Moreover, `DataItemRead` objects can only encapsulate references to `Real` variables and often properties can be integer or boolean.

Hence a new class was created for property objects with the following definition:

```
class Property : public RootObject {

     Real* realDatum;
     int* intDatum;
     bool* boolDatum;
     int selector;      // holds the type of the property
     PropertyId propertyId;

public :

     Property(Real* datum, PropertyId p);
     Property(int* datum, PropertyId p);
```

```
    Property(bool* datum, PropertyId p);

    PropertyId getPropertyId();

    Real get();
};
```

The class can hold references to `Real`, integers and booleans. Three constructors are provided to accept these three types. However, the value of the property is always returned as a `Real` (operation `get` will, if required, perform a type conversion from `int` and `bool` to `Real`).

Consider now the case of a class `aClass` that has a property `prop` to which it wants to give access as a property object. Class `aClass` would implement the following methods:

```
    class aClass {

        Real prop;              // property
        PropertyId propId;      // property identifier

    public :

        // Provide direct access to the property
        Real getProp() { return prop; }

        // Provide access to the property as a "property object"
        Property getPropProperty() {
            return Property(&prop, propId);
        }

        . . .                   // other methods
    }
```

Thus, now the property owner exposes a method called `get<PropertyName>Property` that returns the property object as an instance of class `Property`.

The advantage of this approach is that it uses only existing classes. Its drawback is that it puts the onus of creating property objects on the property owner. This is not very flexible because it means that the decision as to which properties should be available as property objects must be made at design time.

Note that this approach involves associating to each property a *property identifier* that uniquely identifies the property. In practice, the property identifier is formed by combining the object identifier of the property owner with an identifier of the property within the object.

4.5 Baseline Selection

As already mentioned, the second of the two options outlined in the previous subsections is baselined for the AOCS framework. This choice is motivated by a desire to keep the total number of classes in the framework low and by a belief that it will be possible to decide at design time which properties should be available as property objects.

Properties that will normally be available also as property objects are:

- health status flags
- operational mode indicators
- data items in AOCS data variables
- outputs (both housekeeping and functional) of AOCS units

4.6 Additional Properties Pattern

The mechanism baselined for property objects works best when the decision as to which properties should be available as property objects is done at design time. However, the possibility should still remain to introduce new property objects even after a component has been packaged as binary unit. This can be done using the *additional properties pattern* described below.

Consider the following example class:

```
class aClass {

protected :

     Real prop;         // the property

public :

     // Method to provide direct access to the property
     Real getProp() { return prop; }

     . . .              // other methods

}
```

With the above class definition, only direct access to property `prop` is possible. If, however, access through a property object is desired, the class can be extended through inheritance as follows:

```
class aNewClass : public aClass {

     PropertyId propId;

public:

     // Provide new method to allow access to the property
     // as a "property object"
     Property getPropProperty() {
          return Property(&prop, propId);
     }
```

This extension is possible because `prop` was defined in the base class as `protected`.

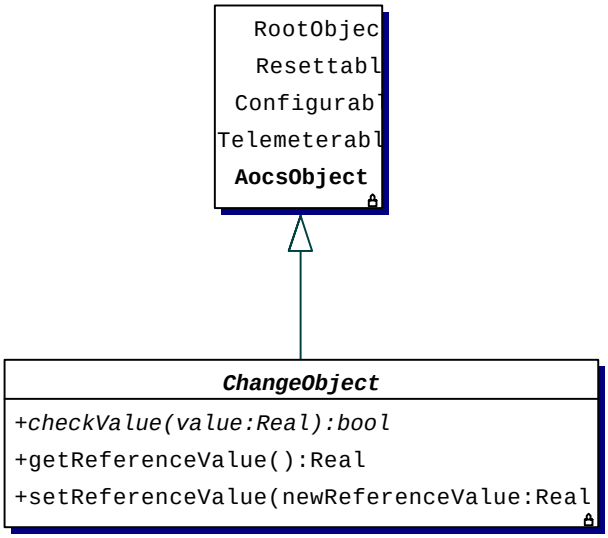
4.7 Summary

The following design rules are adopted in the AOCS framework

- a property <`property`> can be directly accessed through getter and setter (if write access is allowed) methods following the naming conventions of section 4.1.
- an object may make a property available as a property object by exposing a method called `get<property>Property` that returns a `Property` object encapsulating a reference to the property and the property identifier.
- in order to allow extension of property object facilities by inheritance, properties are declared as `protected` variables.
- only variables for which automatic conversion to type `Real` is possible, can be properties.

5 CHANGE OBJECTS

Change objects encapsulate a type of change in a property. The class diagram of change objects is:



`ChangeObject` is an abstract class from which concrete change objects can be constructed as subclasses.

The basic method of class `ChangeObject` is `checkValue`. Clients that intend to check the value of a variable to verify y whether a particular type of change has occurred, hold a change object that encapsulates the desired type of change and periodically pass the variable as an argument to method `checkValue`.

The second and third methods defined by class `ChangeObject` are `getReferenceValue` and `setReferenceValue` that set and get the reference value (see next subsection) associated to the change object.

The following classes of concrete change objects are predefined in the AOCS framework and are made available to application developers as default components:

- *Simple change*: the change occurs when the monitored property changes its value.
- *Out-of-range change:* the change occurs whenever the monitored property moves outside a pre-defined range.
- *Delta change*: the change occurs whenever the monitored property changes by more than a pre-defined delta value.

- *Filtered delta change*: monitors are notified whenever the filtered value of the monitored property changes by more than a pre-defined threshold (not implemented in the AOCS framework).

More specific types of change objects can be added as required.

5.1 Reference Values of Change Objects

In order to characterize a change object, a *reference value* is attached to it. Its interpretation depends on the type of change object. For the basic change object classes defined in the previous section, its interpretation is as follows:

- for a simple change, it is the value of the property before the change was detected (ie. its "default" value)
- for an out-of-range change, it is the limit closest to the last value of the property (ie. the limit that was "violated" by the change)
- for a delta change, it is the maximum allowed variation for the property
- for a filtered delta change, it is the maximum allowed variation for the filtered property

For more complex types of change objects, it may be necessary to attach more than one reference value to each change object.

5.2 The Telemetry Interface

Change objects are telemetry objects because they inherits from `AocsObject` the `telemeterable` interface.

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

TM Format	TM Data
Short	none
Normal	reference value (default value of property)
Long	same as normal TM
Debug	same as normal TM

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

TM Format	TM Data
Short	none
Normal	reference value (delta limit for property change)
Long	normal TM + firstCheck flag (true if this is the first time the change object was triggered)
Debug	same as long TM

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

TM Format	TM Data
Short	none
Normal	lower and upper bounds defining allowed range
Long	normal TM + last value passed through change object
Debug	same as long TM

5.3 The Reset and Configurable Interface

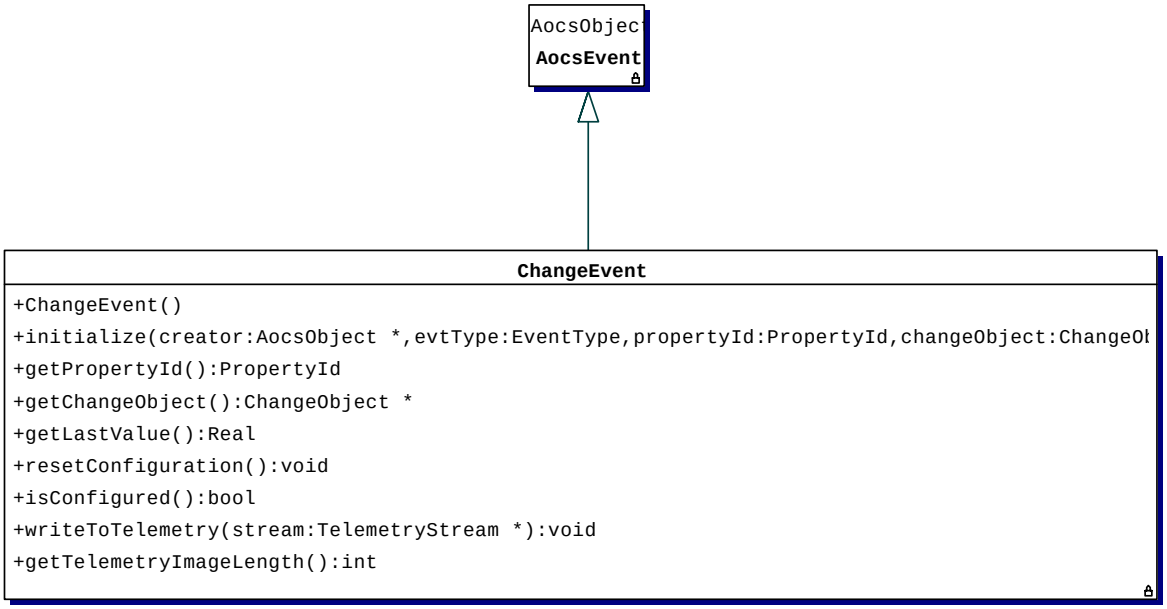
Change objects inherit from `ChangeObject` the `Resettable` and `Configurable` interfaces. The base class `ChangeObject` does not have any configuration or state data and therefore does not provide any class-specific implementation of the methods required by these interfaces.

Subclasses may provide class-specific implementations are required by their semantics. Among the classes predefined by the prototype framework, only `DeltaChange` has a class-specific implementation of method reset.

Finally, note that, for the default change objects provided by the framework, the characteristics of the change object (eg. the delta threshold for a `DeltaChange` object) are not part of the configurable state and hence cannot be changed dynamically (ie. they are set once and for all when the object is constructed).

6 PROPERTY CHANGE EVENTS

The detection of a change by a change object can be signaled by an event of type `ChangeEvent`. The class diagram for this class is shown in the next UML diagram:



Reference to the parameters of the `initialize` method shows that the change event adds the following attributes to those defined by the base class `AocsEvent`:

- `lastValue` : the value that triggered the change (this is the argument passed in the last call to the `checkValue` method of the change object).
- the identifier of the property object that underwent the change
- a reference to the change object that caught the change

The reference to the property object is non-null only when the change occurred in a property (as opposed to an ordinary variable). Getter methods are offered for all these attributes. Note that through the change object it is possible to retrieve the reference value associated to the change object.

To the change event is, as usual, associated an event repository class `ChangeEventRepository`.

6.1 The Telemetry Interface

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

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

TM Format	TM Data
Short	none
Normal	property identifier, change object class identifier (this identifies the type of change that occurred)
Long	property identifier, change object class identifier, last value
Debug	property identifier, change object class identifier, last value

6.2 The Reset and Configurable Interface

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

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

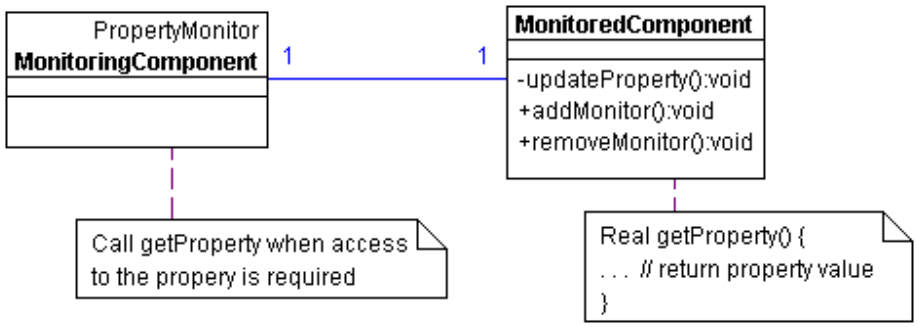
Change events define a class-specific `resetConfiguration` method that resets all event attributes to zero. Method `isConfigured` returns true if the change object reference is different from NULL.

7 PROPERTY MONITORING

The term *monitoring* refers to the observation of a change over time in the value of a property. Monitoring of an object can either be done *directly*, through *property objects*, or by *change notification*. Each of these monitoring types is covered by a design pattern as described in the three following sub-sections.

7.1 Design Pattern for Direct Monitoring

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



7.2 Instantiation of Direct Monitoring Pattern

An object that is interested in monitoring property <property> belonging to object <object> will periodically access it by calling method get<property> on <object>.

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

```
output = aSunSensor.getXoutput();
if (aOutOfRangeChange.checkValue(output))
     . . .     // change has been detected
else
     . . .     // change has not been detected
```

Object aOutOfRangeChange encapsulates an out-of-range change check.

The direct monitoring pattern can also be instantiated by having the getter method in the monitored component return a property object. Consider again the example of the previous sub-section and assume that aSunSensor exposes its X output as a property object. Direct monitoring can also be performed as follows:

```
// holder for property object
Property XoutputProperty;
. . .
// retrieve the property object
XoutputProperty = aSunSensor.getXoutputProperty();
. . .
// perform the test on the property value
if (aOutOfRangeChange.checkValue(XoutputProperty.get()))
     . . .     // change has been detected
else
     . . .     // change has not been detected
```

The result of the monitoring test is the same as before but the test is now performed on a property object. The advantage of this approach is that the entity doing the test need not have a reference to the sun sensor object, all it needs is the property object represented by the data item. This allows systematic monitoring of lists of property objects.

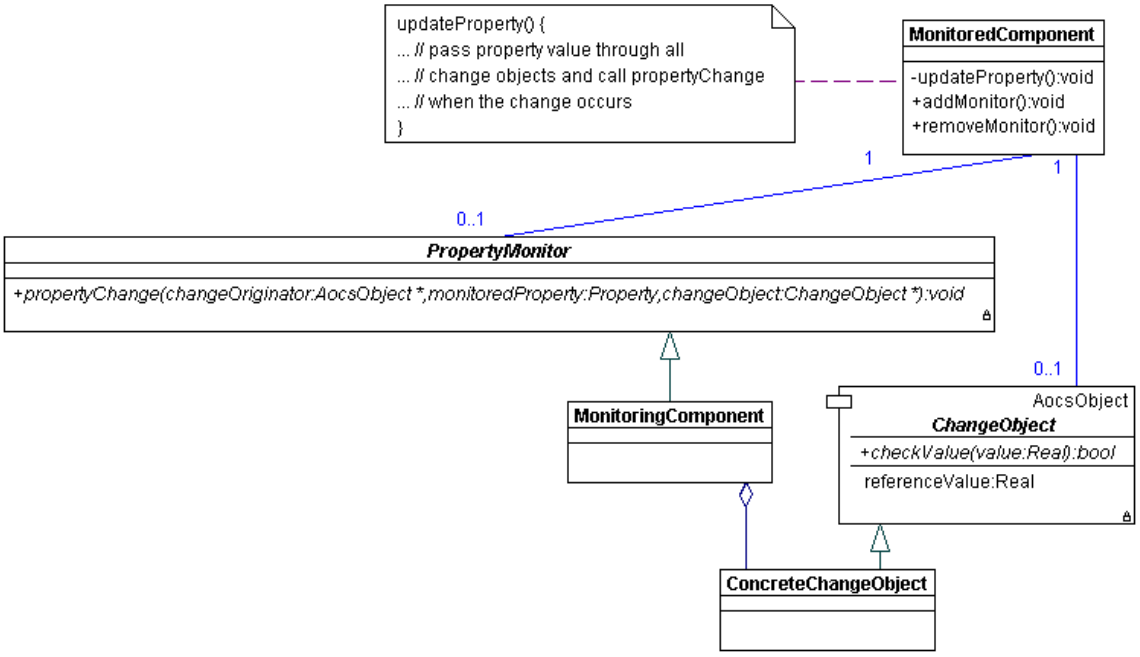
7.3 Design Pattern for Monitoring Through Change Notification

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

With this mechanism, the property owner allows monitors to register their interest in change of a certain type in its property and notifies them when the change occurs.

A property for which monitoring through change notification is possible is called a *bound property*.

This design pattern is modeled on the JavaBeans property mechanism and on the observer pattern of RD1. Its UML diagram is:



7.4 Instantiation of Monitoring with Change Notification Pattern

The property owner allows monitors to register and unregister their interest in a property. It does so by exposing methods with the following signatures:

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

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

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

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

Note that the property owner only receives a *reference* to the change object. This means that responsibility for ensuring that the change object is used only by a single monitored component (change objects have state!) rests with the monitor component.

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

```
class PropertyMonitor {
     void propertyChange(AocsObject* changeOriginator,
                              Property monitoredProperty,
                                   ChangeObject* changeObject) = 0;
}
```

`ChangeOriginator` is the property owner, `monitoredProperty` is a copy of the property being monitored, and `changeObject` is a reference to the change object associated to the monitoring action.

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

Sample code for updating the value of property `property` looks like this:

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

The following are examples of variables that are treated as bound properties in the AOCS framework:

- health status of real AOCS units
- execution status of manoeuvres
- operational mode indicators of mode managers

Note that monitoring through change notification must be used sparingly because the monitor has to sort incoming change events and it can only do this efficiently if the number of changes to which it is listening is limited.

8 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.

8.1 Change Object Hot-Spot

Name: **Change Object Hot-Spot**
Visibility Level: framework-level
Adaptation Time: compile-time
Adaptation Method: derivation from base classes `ChangeObject`
Pre-defined Options: change object components exported by the framelet (see section 3)
Related Hot-Spots: none
Description A change object subclass has to be constructed for each type of change with respect to which monitoring is desired. The derived class only has to provide an implementation for method `checkValue`.

8.2 Property Change Hot-Spot

Name: Property **Change Hot-Spot**
Visibility Level: framelet-level
Adaptation Time: compile-time
Adaptation Method: implementation of method `propertyChange`
Pre-defined Options: none
Related Hot-Spots: none
Description Some of the predefined components in the AOCS framework act as property monitors that perform monitoring through change notification. These components must implement interface `Monitor`

and must provide implementations for method `propertyChange`.