# User Manual 4.7 Events detection: Presentation

De Wiki
Aller à : navigation, rechercher
User Manual 4.7 Events detection: Presentation

## Sommaire

# Introduction

## Scope

This section describes :

- in a mono satellite context :
  - the events detection.
  - the "coded event" and "phenomenon" notions that extend event detection mechanism.
- in a multi satellite context :
  - the events detection.
  - the "coded event" and "phenomenon" notions, copied from the mono satellite mechanism, that extend Patrius's event detection mechanism for multi satellite propagation.

## Javadoc

The following packages are related to events detections in mono and multi propagation context:

| Library | Javadoc |
|---|---|
| Patrius | Package fr.cnes.sirius.patrius.propagation.event |
| Patrius | Package fr.cnes.sirius.patrius.math.ode.events |
| Patrius | Package fr.cnes.sirius.patrius.events |
| Patrius | Package fr.cnes.sirius.patrius.propagation.events.multi |

## Links

None as now.

## Useful Documents

None as of now.

## Package Overview

The following diagram represents the main architecture of the events detection for mono and multi satellite propagation:

The following diagram represents the mechanism used in the Patrius library in order to create the list of the events detected during the mono satellite propagation as well as the phenomena list :

The following classes are duplicated from mono satellite context in order to create a list of events detected :

- MultiCodedEventsLogger

- MultiCodingEventDetector
- MultiEventsLogger
- MultiGenericCodingEventDetector

# Features Description

## Event detection

The events detection process can be used during mono or multi propagation propagation, when we need to know if some events occur and at what time (for instance when the satellite will be in eclipse, or if at a given date it will be visible or not from a ground station).

In Patrius there is one class for every kind of event: the information related to an event is contained in this class, and it will be used by the mono satellite numerical or analytical propagator when extrapolating the orbit. The mechanism is similar for multi propagation.

When two events occur at the same date, Patrius detects both of them, but the order in which they are detected is the order in which they are added to the propagator.

As of PATRIUS 4.5, detectors can optionally take into account signal propagation delay. Javadoc of each event detector indicates if signal propagation delay can be taken into account through a setter `setPropagationDelayType()`. Delay type is of 2 types:

- `PropagationDelayType.INSTANTANEOUS`: signal propagation delay is not taken into account
- `PropagationDelayType.LIGHT_SPEED`: signal propagation delay is taken into account.

Signal propagation delay allows to detect events as it is exactly seen by a spacecraft, for instance a visibility between a satellite and a station. All detectors involving a signal propagation and or sensors can take signal propagation delay into account (BetaAngleDetector, SensorVisibilityDetector, TargetInFieldOfViewDetector, etc.):

By default signal propagation delay is not taken into account.

A new detector class should implements :

- EventDetector interface in case of mono satellite propagation or multi satellite propagation.
- MultiEventDetector interface in case of multi satellite propagation.

***Construction of a detector***
The action performed at event detection can be set by user at detector's construction. Several actions can be defined if several behavior are available (depending on state and/or on increasing/forward parameter). It is possible to specify at construction if the detector should be removed after event detection and thus not be detected any more. If a single action is available, a single boolean has to be provided by the user at construction. In the case where several actions are available, the same number of boolean has to be provided to determine if the detector has to be removed after event detection.
Focusing on the `NodeDetector` for instance, two actions are possible at event detection :

- action_at_ascending_node if an ascending node detection is done
- action_at_descending_node if a descending node detection is performed

If no actions are set, the user should be aware of the default implementation. By default, event

detectors are never removed.

***Initialisation of eventOccurred()***
The returned action of the function eventOccurred could be set by user or defined by default.

The function eventOccurred is common to all EventDetector classes (respectively MultiEventDetector) and its purpose is to handle an event and to choose what to do next ; this method is called when the integrator has accepted a step ending exactly on a sign change of the switching function, and it allows the user to choose which action will take place after the event (the simulation could be stopped, the propagator could reset its initial state or its initial derivative state ...). Plus, this method is up to determine if the detector used has to be removed after the event detection : in the above example of `NodeDetector` , the action taking place after the event is set to action_at_raising_node in the case of an ascending node detection. If one has decided to remove the detector at such detection, the attribute shouldBeRemoved will be set to remove_at_ascending_node which is true by construction, the case of descending node is similar.

Since the default implementation of eventOccurred could unfit the user needs in terms of propagation behaviour, it is recommended to check the content of the detector before using them and to change it if necessary by using a specific constructor which take as parameter the expected action(s).

Note that the implementation of eventOccured must not contain any action. Even if the action does not impact the state vector, the action should be implemented in the resetState method.

***Initialisation of resetState() or resetStates()***
If the event have an action, this action should be implemented in the resetState() method. In this case, the eventOccured() of the detector concerned should return a RESET_STATE action.

***Convergence threshold parametrisation***
The convergence threshold represents the events detection precision; the following observations can help the user when choosing a value for this parameter:

1. if the convergence threshold increases, the execution time decreases (the precision becomes worst);
2. if the convergence threshold is bigger than the max integration step, the events detection will fail.

If the resetState(s) could modify others event detectors, the convergence threshold should be low in order to avoid any unpredictable behavior.

***Max check interval parametrisation***
The max check interval parameter represents the maximum interval in which the propagator checks for an event (i.e. for a sign change of the g switching function). When choosing this parameter, the user should keep in mind the following remarks:

1. if max check interval is smaller than the integration step, the execution time increases if the max check interval decreases (the events detection is more accurate);
2. if max check interval is bigger than the max integration step, it will be replaced by the max integration step during the events detection process; in this case the max check interval is a useless parameter that will never be used by the propagator.

In general, the user should always choose the max check interval value as a function of the integration step used during propagation.

***Initialisation of the event detection***
If an event occurs exactly at the propagation start date, i.e if g() equal to zero, the event is processed. This case is extremely rare. Note that in some case however, due to numerical quality rounds-off, an event may not be detected at the propagation start date although it's theoritically supposed to occur. For example, starting propagation at periapsis with a periapsis detector may not detect periapsis since the g() function of PeriapsisDetector is based on position- velocity scalar product and due to rounds-off errors, its values may be around 1E-7 which is not exactly zero. Also, if at the propagation start date the g() function is not continuous but changes its sign (for example value-1 before event date, value 1 after), the event will also not be detected since the g() function value is not zero.

***Ephemeris propagator event detection***
Sometimes the event detection is performed on a substep a little larger than the real one. Consequently, when propagating with a bounded propagator, if the last substep is larger than the real one then the real end date is exceeded, raising an exception. Therefore, in this case, it is recommended to **propagate over an interval shorter than the interval of validity of the bounded propagator**.

***Events occurring at the same date***
When two events occur at the same date, Patrius detects both of them, but the order in which they are processed is the order in which they are added to the propagator. If the detected event stops the propagation, the other events occuring at the same date are not processed.

***Events occurring at the end date***
If an event occurs exactly at the propagation end date, the event is not processed.

**Available event detectors**

Mono events detectors detect events applied on a specific state. These detectors can be used in mono or multi propagation case.

The available mono events detectors are presented in specific pages:

- [MIS_ORB_Home Orbit determination events]
- [MIS_SENSORS_Home Sensors events]
- [MIS_STASAT_Home Ground stations and satellites events]

All these event detectors can be used [MIS_EVT_Home#HEventDetection directly] or as [MIS_EVT_Home#HGeneratingcodedeventsandphenomena coded events].

It also exists multi events detectors applied on several states. These detectors can only be used in multi propagation case. Notice that some detectors that implies several satellite coul be used in multi or mono satellite context ; (SatToSatMutualVisibilityDetector, ThreeBodiesAngleDetector or ExtremaThreeBodiesAngleDetector)

All these event detectors can be used [MIS_EVT_Home#HMultiEventDetection directly] or as [MIS_EVT_Home#HGeneratingcodedeventsandphenomena coded events].

**Nth occurrence detector**

The NthOccurrenceDetector is a detector that detects the nth occurrence of an underlying event

detector.
However the `eventOccurred()` method is triggered at every event of the underlying detector. As a result, the behaviour of this detector is the following:

- Before and after the nth occurrence, the eventOccurred() method returns `Action.CONTINUE`.
- At the nth occurrence, the `eventOccurred()` method returns the user-provided action.

***Warning***: the `eventOccurred()` method is triggered at every occurrence of the underlying detector, not only at nth occurrence. Hence, overloading this detector should be performed carefully: in the overloaded `eventOccurred()` method, the check `getCurrentOccurence() == getOccurence()` should be performed first to ensure we are at nth occurrence before calling `super.eventOccurred()`.

### IntervalOccurenceDetector

This IntervalOccurenceDetector is able for any EventDetector. It detects the [math]n^{th}[/math] to [math]m^{th}[/math] event occurences by step of [math]p[/math] occurences. Otherwise, if j event occurence, the `IntervalOccurenceDetector` detects events as :

- [math](j - n) % p = 0[/math]
- [math]n \lt= j \lt= m[/math]

The g switching function is the function of the event to detect.

### Mono Events detection process description

#### Numerical propagator

When a numerical propagation is performed, the event detection is done at the math level since the integration is realized by one of the numerical integrators. However, the event detectors are instanciated by the user at the Patrius level. Hence the necessity of an adapter to wrap a Patrius event detector object into a math event handler object whose interfaces provide basically the same kind of services. Once the event detectors are given to the numerical propagator, they are transformed into event handlers in order to give them to the math numerical integrator.
The numerical integrator associates each event with another object which represents its state during the integration. At the end of each step, the method acceptStep() is systematically called and performs the event detection thanks to an interpolator, inside the current step. Thus the state of each event is updated. If an event is detected therefore the integration is either stopped or continued with or without state or derivative state resetting. It is also possible to remove the detector after the first event detection and the propagation is continued. An event is represented by a function, the function "g()": the event occurs when the function sign changes and NOT when this function equals to zero. To find this root, in addition to the interpolator, a solver is used. By default, this solver is the Brent solver. The exception being at the beginning of the propagation where events are detected when the g() function equals exactly zero.

#### Analytical propagator

When an analytical propagation is performed, the event detection is done at the Patrius level directly. The process is basically the same than its math counterpart: the detectors are given to the propagator and associated to their states. The Patrius propagator has a method acceptStep() which is similar to the one of the math package.

#### Ephemeris propagator

Sometimes the event detection is performed on a substep a little larger than the real one which could be knotty when it comes to propagate with a bounded propagator. Indeed, if the last substep is larger than the real one then the real end date is exceeded. If the end date is equal to the ephemeris max date, an exception is raised when the interpolator sets the current date to a date exceeding the ephemeris max date. Therefore, in this case, **it is recommended to propagate over an interval shorter than the interval of validity of the bounded propagator**.

### Multi Events detection process description

#### Numerical propagator

The event detector process for multi satellite numerical propagation is equivalent to the process performed with single satellite numerical propagation. The multi event detection is wrapped into a math event handler object. The detector could be applied on a single satellite or on several satellites.

## Coded events and phenomena

### Mono propagation

#### The coded event

Patrius does detect events. This means :

- Patrius triggers a method call on the corresponding EventDetector when the event occurs.
- Patrius logs the SpacecraftState and g function slope sign for the occuring event (through the EventsLogger).

But this mechanism alone doesn't provide a programmatic representation for an "event", which may be processed after the propagation.

A new way of handling events as individual instances was thus designed : the "atomic event" or "coded event".

A "coded event" has the following attributes :

- an event "code" : a String identifying the category of the event, as chosen by the library user (no actual code is provided yet).
- an event "comment" : also a String that can be chosen by the library user.
- an AbsoluteDate : the date at which the event occurred.
- a boolean, true when the event represents a "starting" event, or false when it represents an "ending" event (relative to a phenomenon, see below).

The code and comment formats are free, they depend on the CodingEventDetector implementation (see below).

A CodedEvent is supposed to be built by the CodedEventsLogger (see below).

Moreover, when one of the boundaries of the phenomenon is ill-defined (e.g. unknown due to computational limits), it is an instance of CodedEvent, with the code "UNDEFINED_EVENT".

#### The CodingEventDetector and MultiCodingEventDetector

The CodingEventDetector (or MultiCodingEventDetector in MultiPropagator case) is an extension of the EventDetector type(respectively MultiEventDetector). A CodingEventDetector (or MultiCodingEventDetector) has the following requirements :

- Providing a CodedEvent builder method, that creates a CodedEvent instance appropriate for the event (this is how the code and comment are determined).
- Providing a Phenomenon code if a Phenomenon makes sense in the context of the event, otherwise return null (see below).
- Providing a boolean telling which sign of the g() method means the Phenomenon is active (see below).

#### The phenomenon

The EventDetector provides a g() method whose sign change triggers an event. Sometimes the g() method can also be seen as conveying a "state" i.e. when g() is positive (resp. negative), a "phenomenon" is going on. The clearest example would be the EclipseDetector :

- g() going from positive to negative means that the spacecraft has entered the eclipse zone.
- g() going from negative to positive means that the spacecraft has exited the eclipse zone.

The associated phenomenon would then be called "inside the eclipse zone", bounded by the starting event "enter eclipse zone" and the ending event "exit eclipse zone".

This library provides a "phenomenon" abstraction, as a lightweight representation.

A Phenomenon instance is an immutable object with the following attributes :

- a phenomenon "code" : a String identifying the category of the phenomenon, as chosen by the library user (no actual code is provided yet).
- a phenomenon "comment" : also a String that can be chosen by the library user.
- two "coded event" instances : the atomic events that represent the start and the end of the phenomenon.

Also :

- it tells whether the start and end are "well-defined" : a well-defined boundary has a real event backing it. A not well-defined boundary is a boundary because of computational limits : it happens when, during a propagation, the ending event occurs, but the starting event did not (or, the opposite) because it was outside the boundaries of the propagation. In this case, an "UNDEFINED_EVENT" is given as the missing boundary (see above).

A Phenomenon is built by the CodedEventsLogger, when the CodingEventDetector provides a non-null Phenomenon string code. The existence of this code proves the Phenomenon makes senses in the context of the CodingEventDetector. For instance :

- for a "CodingEclipseDetector" built upon the EclipseDetector, a Phenomenon has meaning (the Phenomenon being "inside the eclipse").
- for a "CodingApsideDetector" built upen the ApsideDetector, a Phenomenon has no meaning, since the events are : "the spacecraft is at the periapsis" and "the spacecraft is at the apoapsis".

### Multi propagation

The same process exists. The corresponding classes are localized in the Patrius library.

### Setting up a CodedEventsLogger or a MultiCodedEventsLogger

The CodedEvent and Phenomenon instances are not meant to be produced by the CodingEventDetectors (or directly MultiCodedEventsLogger in MultiPropagator case) (but it could be possible if needed).

Instead, the (Multi)CodedEventsLogger was created for this purpose.

At first glance, a (Multi)CodedEventsLogger instance is very similar to the (Multi)EventsLogger. The main difference is that the (Multi)EventsLogger instances have no inner representation of events, while the (Multi)CodedEventsLogger produces CodedEvent and Phenomenon lists.

Here's how one uses the (Multi)CodedEventsLogger :

- create a (Multi)CodedEventsLogger instance.
- create at least one (Multi)CodingEventProvider instance.
- call the `monitorDetector()` method on the (Multi)CodedEventsLogger : it produces a "wrapper" of the (Multi)CodingEventProvider exposed as a regular EventProvider.
- pass this "wrapper" to a propagator (any kind works : those who do compute a propagation and those who replay an ephemeris).
- run the propagator.

While the propagator runs, the "wrapper" calls both the (Multi)CodingEventProvider instance and the (Multi)CodedEventsLogger on each event. The (Multi)CodedEventsLogger instance uses the (Multi)CodingEventProvider to build and store all CodedEvent instances during the propagation.

### Getting the full coded events list

Call the method `getCodedEventsList()`. It provides a list of all CodedEvents (for all the CodingEventDetectors passed to the propagator, without distinction).

### Getting the events list per (Multi)CodingEventProvider

Call the method `buildCodedEventListMap()`.
It builds a `java.util.Map`, with CodingEventProvider instances as keys, and a CodedEventsList as value, this list only containing the events generated from the key.
It is, therfore, the same data as the full events' list, but rearranged per CodingEventProvider instance.

### Getting the phenomena per (Multi)CodingEventProvider

Call the method `buildPhenomenaListMap()`.
It builds a `java.util.Map`, with CodingEventProvider instances as keys, and a PhenomenaList as value, this list only containing the Phenomenon instances generated from the key. The(Multi) CodingEventProvider instance provides the Phenomenon code.

The (Multi)CodedEventsLogger only uses the (Multi)CodingEventProvider instances that provide a

non-null Phenomenon code (null indicates that a Phenomenon has no meaning for this detector).

#### N-th occurrence of a coded event, or a delayed event

There are two ways to generate the n-th occurrence of an event or a delayed coded event from a detected event: the first one is using the post-processing classes, and the second one is to configure the CodingEventDetector in order to create the delayed or filtered coded events during the propagation (in addition to the "standard" coded events). While the former can only be done after the propagation, the latter has to be set before the propagation and is possible thanks to the following methods in the class CodingEventDetector:

- `buildCodedEvent(SpacecraftState, boolean)` to generate the standard CodedEvent
- `buildDelayedCodedEvent(SpacecraftState, boolean)` to generate the delayed CodedEvent
- `buildOccurrenceCodedEvent(SpacecraftState, boolean)` to generate the n-th occurrence CodedEvent (n will be a parameter chosen by the user)

In order to use this feature, the delay value and/or the occurrence number must be given as input parameters of the constructor of the class implementing the CodingEventDetector class (for instance GenericCodingEventDetector). The CodedEventsLogger will then be able to read these parameters and handle the generation of standard and not-standard coded events.

Note, in the GenericCodingEventDetector constructor (EventDetector, String, String, boolean, String, double, int), if both parameters delayIn and occurrenceIn are other than 0, the event is saved as a delayed event, not a "Nth occurence of".

### When to use Coded events and phenomena

- To build easier time lines of events
- To manipulate easier phenomena (for example between "starting" and "ending" events) and associate to them specific computations (for example the evolution of the illumination during an eclipse)
- To apply some [MIS_POSTP_Home post-processing filters]

## Getting Started

### Using available events detectors

#### Nth occurrence detector

Example of propagation stopping at 3rd detected node with event detector overloading:

```
// Initialization
final AbsoluteDate date = new AbsoluteDate(2003, 1, 1,
TimeScalesFactory.getTAI());
final Orbit orbit = new KeplerianOrbit(7000000, 0.01, 0.1, 0.2, 0.3, 0.4,
PositionAngle.TRUE, FramesFactory.getGCRF(), date, Constants.EGM96_EARTH_MU);

// Node detector
final EventDetector nodeDetector = new NodeDetector(orbit,
FramesFactory.getGCRF(), NodeDetector.ASCENDING_DESCENDING);
```

```
// Stop at 3rd detected node
final NthOccurrenceDetector nthOccurrenceDetector = new
NthOccurrenceDetector(nodeDetector, 3, Action.STOP) {
    @Override
    public Action eventOccurred(SpacecraftState s, boolean increasing,
boolean forward) throws PatriusException {
        super.eventOccurred(s, increasing, forward);
        if (getCurrentOccurence() == getOccurence()) {
            return Action.STOP;
        } else {
            return Action.CONTINUE;
        }
    }
};

// Propagation
final KeplerianPropagator propagator = new KeplerianPropagator(orbit);
propagator.addEventDetector(nthOccurrenceDetector);
propagator.propagate(orbit.getDate().shiftedBy(86400.));
```

## Generating coded events and phenomena

### Examples

The CodedEvent instances and Phenomenon instances are meant to be produced using :
- CodingEventDetector implementations that provide information about events and phenomena,
- a CodingEventsLogger, that uses the CodingEventDetector instances to build the events and phenomena.

For example, if we want to create a list of CodedEvent objects using a detector of nodes (ascending and descending orbital nodes):

1. Create a new EventDetector (in this case a NodeDetector):

```
  // Set up the node detector:
  AbsoluteDate date = new AbsoluteDate("2000-01-01T12:00:00Z",
TimeScalesFactory.getTT());
  AbsoluteDate dateF = date.shiftedBy(2* period);
  Orbit orbit = new KeplerianOrbit(7e6, 0, 0.2, 0, 0, 0.2,
PositionAngle.MEAN, FramesFactory.getGCRF(), date, Constants.EGM96_EARTH_MU);
  NodeDetector node = new NodeDetector(orbit, FramesFactory.getEME2000(),
NodeDetector.ASCENDING_DESCENDING);
```

2. Create a new generic coding event detector:

```
  // standard event
  GenericCodingEventDetector nodeDet =
    new GenericCodingEventDetector(node, "Ascending node", "Descending node",
true, "Nodes");
  // third occurence of event
```

```
  GenericCodingEventDetector nodeDetOcc =
    new GenericCodingEventDetector(node, "Ascending node", "Descending node",
true, "Nodes", 0., 3);
  // delayed event (10 seconds here)
  GenericCodingEventDetector nodeDetDel =
    new GenericCodingEventDetector(node, "Ascending node", "Descending node",
true, "Nodes", 10., 0);
```

3. Create a new CodedEventsLogger and use the `monitorDetector` function on the new CodedEventsLogger:

```
  CodedEventsLogger logger = new CodedEventsLogger();
  // Create an EventDetector object so that event detection functions could
be used:
  EventDetector d = logger.monitorDetector(nodeDet);
  EventDetector dOcc = logger.monitorDetector(nodeDetOcc);
  EventDetector dDel = logger.monitorDetector(nodeDetDel);
```

4. Add the EventDetector to the propagator:

```
  propagator.addEventDetector(d);
  propagator.addEventDetector(dOcc);
  propagator.addEventDetector(dDel);
```

5. Start the propagation:

```
  // Set the propagator:
  propagator.setEphemerisMode();
  propagator.resetInitialState(new SpacecraftState(orbit));
  // Propagate:
  propagator.propagate(date0, dateF);
```

## Combination of boolean phenomena

Sometimes an event (a zero of a g function) can be seen as a begin (passing zero from positive to negative or invert) or a end ( other way ) of a phenomena. For example the eclipse event can be the begin (detected when g function goes from positive to negative) or the end (when g function goes from negative to positive) of an eclipse. (cf. [MIS_ORB_Home#HEclipseDetectorandGenericEclipseDetector Eclipse detector].)

One can choose to create an event as a combination of these phenomena, for example a station visibility outside an interference period. In this case a user can create a detector as a combination of existing detectors.

The user simply has to provide the following parameters :

1. First detector and the way to use it (g is positive or negative during the phenomena to combine)
2. Second detector and the way to use it

3. How to combine the phenomenom : both are during phenomena (will be detected as the minimum of the two g function) or at least one (will be detected as the maximum of the two g function) is during phenomena

Once a new detector have been created in this way, it is possible to combine it with another one and so on.

For example the [MIS_SENSORS_Home#HCentralBodyMaskCircularFOVDetector CentralBodyMaskCircularFOVDetector] detects when a target is in a circular field of view and outside of an eclipse, that is translated by :

- Outside of eclipse : when EllipsoidEclipseDetecor g function is positive
- Inside circular field of view : when CircularFieldOfView g function is positive

So the following code example :

```
// EclipseDetector
final EclipseDetector eed = new EclipseDetector(targetPVP, targetRadius,
spheroEarth, 1., MAXCHK, THRS);

// CircularFOVDetector
final CircularFieldOfViewDetector cfvd = new
CircularFieldOfViewDetector(targetPVP, fovDir, halfAp, MAXCHK, THRS);

// Detector in field of view and outside eclipse
final CombinedPhenomenaDetector cbmcfovd2 = new
CombinedPhenomenaDetector(eed, true, cfvd, true, true);
```

is equivalent to CentralBodyMaskCircularFOVDetector :

```
// Detector in field of view and outside eclipse
final CentralBodyMaskCircularFOVDetector cbmcfovd =
new CentralBodyMaskCircularFOVDetector(targetPVP, targetRadius, spheroEarth,
false, fovDir, halfAp, MAXCHK, THRS);
```

# Contents

## Interfaces

### Interfaces events detection

| Interface | Summary | Javadoc |
|---|---|---|
| EventDetector | This interface represents an event finder. | EventDetector |
| CodingEventDetector | This interface is an extended event finder, that is needed to create CodedEvent and Phenomenon instances. | CodingEventDetector |
| MultiEventDetector | This interface represents an event finder in multi propagation case. | MultiEventDetector |

| | | |
|---|---|---|
| MultiCodingEventDetector | This interface is an extended event finder, that is needed to create CodedEvent and Phenomenon instances in multi propagation case. | MultiCodingEventDetector |

## Classes

### Classes containing the general functions to detect events

| Class | Summary | Javadoc |
|---|---|---|
| AbstractDetector | This class contains the methods shared by all events detectors. | AbstractDetector |
| MultiAbstractDetector | This class contains the methods shared by all events detectors in multi propagation case. | MultiAbstractDetector |
| AdaptedEventDetector | This class adapts Patrius event detectors to math EventHandler object. | AdaptedEventDetector |
| AdaptedMonoEventDetector | This class adapts Patrius mono event detectors to math EventHandler object in multi propagation case. | AdaptedEventDetector |
| AdaptedMultiEventDetector | This class adapts Patrius multi event detectors to math EventHandler object in multi propagation case. | AdaptedEventDetector |
| CombinedPhenomenaDetector | This class handles events representing the combination of two boolean phenomena. | CombinedPhenomenaDetector |
| NthOccurrenceDetector | This class represents an event detector that will detect a certain occurrence of a specified event. | NthOccurrenceDetector |
| IntervalOccurrenceDetector | This class represents an event detector that will detect a interval occurrence of a specified event. | IntervalOccurrenceDetector |

### Classes for the generation of lists of events and phenomena

| Class | Summary | Javadoc |
|---|---|---|
| CodedEvent | This class implements coded events (a light and simple representation of events as generated by Patrius event detectors ). | CodedEvent |
| CodedEventsList | This class implements a list of coded events. | CodedEventsList |
| Phenomenon | This class implements the notion of a phenomenon : a state defined by a starting event and an ending event. | Phenomenon |
| PhenomenaList | This class implements a list of coded events. | PhenomenaList |

| | | |
|---|---|---|
| CodedEventsLogger | This class builds instances of CodedEventsList and PhenomenaList during a propagation. | CodedEventsLogger |
| MultiCodedEventsLogger | This class builds instances of CodedEventsList and PhenomenaList during a propagation in multi propagation case. | MultiCodedEventsLogger |
| GenericCodingEventDetector | This class is a generic implementation of the CodingEventDetector interface, mainly for testing purposes - but it is usable as is. | GenericCodingEventDetector |
| MultiGenericCodingEventDetector | This class is a generic implementation of the CodingEventDetector interface in multi propagation case, mainly for testing purposes - but it is usable as is. | MultiGenericCodingEventDetector |


Catégorie :

- User Manual 4.7 Mission

# Menu de navigation

## Outils personnels

- 3.15.34.50
- Discussion avec cette adresse IP
- Créer un compte
- Se connecter

## Espaces de noms

- Page
- Discussion

## Variantes

## Affichages

- Lire
- Voir le texte source
- Historique
- Exporter en PDF

## Plus

## Rechercher

Rechercher Lire

## PATRIUS

- Welcome

## Evolutions

- Main differences between V4.14 and V4.13
- Main differences between V4.13 and V4.12
- Main differences between V4.12 and V4.11
- Main differences between V4.11 and V4.10
- Main differences between V4.10 and V4.9
- Main differences between V4.9 and V4.8
- Main differences between V4.8 and V4.7
- Main differences between V4.7 and V4.6.1
- Main differences between V4.6.1 and V4.5.1
- Main differences between V4.5.1 and V4.4
- Main differences between V4.4 and V4.3
- Main differences between V4.3 and V4.2
- Main differences between V4.2 and V4.1.1
- Main differences between V4.1.1 and V4.1
- Main differences between V4.1 and V4.0
- Main differences between V4.0 and V3.4.1

## User Manual

- User Manual 4.14
- User Manual 4.13
- User Manual 4.12
- User Manual 4.11
- User Manual 4.10
- User Manual 4.9

- User Manual 4.8
- User Manual 4.7
- User Manual 4.6.1
- User Manual 4.5.1
- User Manual 4.4
- User Manual 4.3
- User Manual 4.2
- User Manual 4.1
- User Manual 4.0
- User Manual 3.4.1
- User Manual 3.3

## Tutorials

- Tutorials 4.14
- Tutorials 4.13.5
- Tutorials 4.12.1
- Tutorials 4.8.1
- Tutorials 4.5.1
- Tutorials 4.4
- Tutorials 4.1
- Tutorials 4.0

## Links

- CNES freeware server

## Navigation

- Accueil
- Modifications récentes
- Page au hasard
- Aide

## Outils

- Pages liées
- Suivi des pages liées
- Pages spéciales
- Adresse de cette version
- Information sur la page
- Citer cette page

- Dernière modification de cette page le 19 décembre 2023 à 16:18.

- Politique de confidentialité
- À propos de Wiki

- Avertissements
-