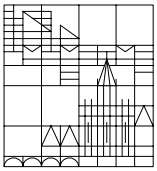


University of Constance
Dept. of Computer Science

FAILURE RECOVERY MANAGEMENT FRAMELET

Concept And Architecture Description

Abstract

This document was written as part of the study "Design and Prototyping of a Software Framework for the AOCS" done under contract Estec/13776/99/NL/MV for ESA-Estec. The purpose of the study is the development of a software framework for the Attitude and Orbit Control Subsystem (AOCS) of a satellite. The framework will be built as a collection of framelets. This document describes the failure recovery management framelet. This framelet defines an architecture to handle failure recovery tasks. The framelet enhances reusability because it decouples the task of managing the failure recovery function from the task of carrying out the failure recovery actions.

Written By: A. Pasetti

Date: 30 April 2002

Issue: 2.2

Reference: SWE/99/AOCS/011

TABLE OF CONTENTS

1 REFERENCES

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

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

RD3 W. Pree, A. Pasetti (2000), *Operational Mode Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/009

RD4 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 *failure recovery management framelet* for the AOCS framework. The framelet is described at both the framelet concept level and at the framelet architectural level.

This framelet defines an architecture to handle failure recovery tasks.

The framelet enhances reusability because it decouples the task of *managing the failure recovery function* from the task of *implementing failure recovery algorithms.*

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 sections dealing with failure recovery management and with the overall FDIR approach.

The architecture proposed here follows the concept outlined in RD2.

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

3.2 Applicability to Java Version

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

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

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

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

3.3 Notation

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

The class diagrams use UML notation generated with the reverse engineering tool of the *Together* tool.

4 FRAMELET CONSTRUCTS

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

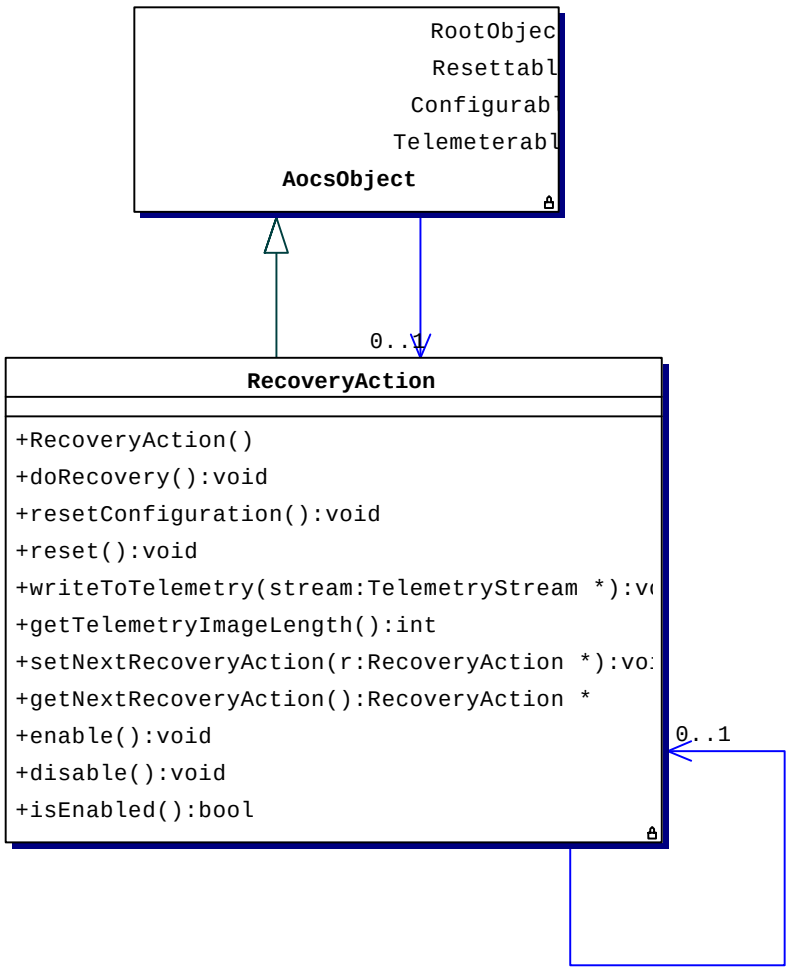
FAILURE RECOVERY MANAGEMENT FRAMELET
Design Pattern
Failure Recovery Pattern: design pattern to separate the management of failure recovery from the implementation of failure recovery strategies.
Framelet Interfaces and Base Abstract Classes
`RecoveryAction` : abstract base class for objects encapsulating recovery actions `RecoveryStrategy`: abstract base class for objects encapsulating failure handling strategies `FailureRecoveryModeManager` : interface for the operational mode manager for the failure detection manager.
Framelet Core Components
`FailureRecoveryManager` : failure recovery manager component (including mode manager)
Framelet Components
`SystemReset` : recovery action component encapsulating a system reset `SystemReboot` : recovery action component encapsulating a system reboot `ObjectReset` : recovery action component encapsulating a reset on a specific object `Reconfiguration` : recovery action component encapsulating a reconfiguration action `ModeChange` : recovery action component encapsulating a mode change action `NullRecoveryAction` : null recovery action `SystemResetOnTooManyFailures` : failure recovery strategy to command a system reset if too many failures are found in the failure recovery repository `LocalRecoveryActions` : failure recovery strategy to perform the recovery actions associated to each failure found in the failure recovery repository `FollowerFailureRecoveryModeManager` : failure recovery mode manager based on based on follower mechanism

The components listed above are those envisaged for the prototype version of the AOCS framework. Later versions may offer a richer set of default implementations of the framelet interfaces. In particular, more recovery action and recovery strategy components might be provided.

5 FAILURE RECOVERY ACTIONS

In general, the AOCS software can react to a specific failure event by performing one or more *failure recovery actions.* A failure recovery action therefore represents a *local* response to a failure. The response is said to be local because it is based on a single failure report. A *global* response would take account of sets of failure reports.

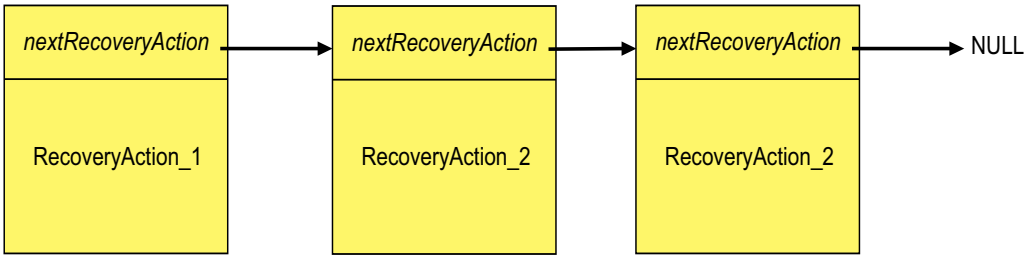
Failure recovery actions are encapsulated in objects derived from class `RecoveryAction`:



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

`doRecovery()`
Implement the recovery actions encapsulated by the component.
`enable(), disable(), isEnabled()`
Recovery actions can be enabled or disabled. These operations allow the enable status of the recovery action to be changed and to be queried. Execution of operation `doRecovery` on a disabled recovery action is equivalent to a no-op.
`getNextRecoveryAction(), setNextRecoveryAction()`
Recovery actions can be chained (see below). These are the getter and setter methods for the next recovery actions in the link chain.

Recovery actions can be linked in a linear chain as shown in the picture:



The implementation of `doRecovery` provided by the base class `RecoveryAction` is as follows:

```
if (nextRecoveryAction != NULL)
     nextRecoveryAction->doRecovery();
```

A typical implementation of `doRecovery` in a subclass of `RecoveryAction` would be as follows:

```
void doRecovery() {
     . . .              // perform class-specific recovery action
     RecoveryAction::doRecovery();
}
```

This means that execution of a recovery action will lead to execution of all recovery actions that are linked to it. Thus, a single recovery action can in fact represent any number of

recovery actions to be executed in sequence. In practice, this allows several recovery actions to be associated to the same failure event.

5.1 Types of Recovery Action

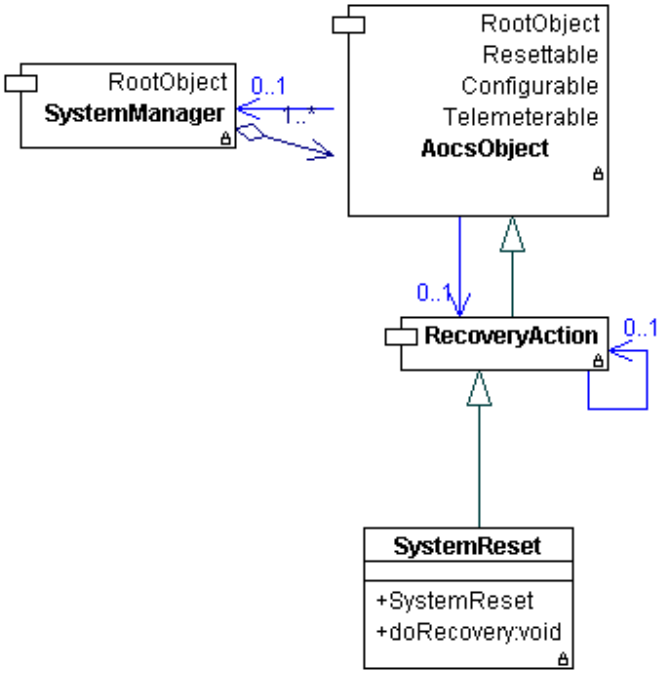
Class `RecoveryAction` is the base class for all recovery actions. By itself, it does not define any concrete action. Concrete recovery actions are defined by subclasses of `RecoveryAction`. Typical failure recovery actions would include:

- Reset of the AOCS software
- Reboot of the AOCS software
- Reset of one or more AOCS objects
- Reconfiguration of one or more units
- Fall-back to a lower operational mode

To each type of recovery action there corresponds a concrete class derived from `RecoveryAction`. More details are provided in the next sub-sections.

5.2 System Reset Recovery Action

This recovery action uses the services of the `SystemReset` object to perform a system reset of the AOCS. The UML diagram of this recovery action is:



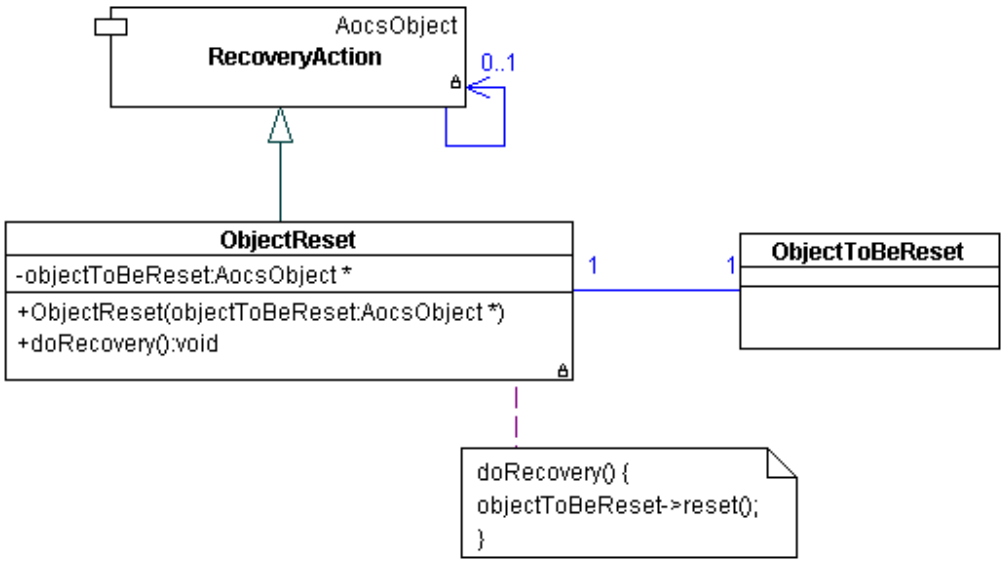
The system reset is performed by using the `systemReset` service offered by the `SystemManager` component. The recovery action retrieves the reference to the system manager using one of the services offered by its parent class `AocsObject`.

5.3 System Reboot Recovery Action

This recovery action performs a system reboot. No default implementation is provided by the framework since a system reboot requires interfacing to the low-level drivers of the operating system

5.4 Object Reset Recovery Action

Objects that are derived from `AocsObject` implement the `Resettable` interface and can thus be reset. Reset of a specific object can be a failure recovery action. The UML class diagram of the corresponding class is:

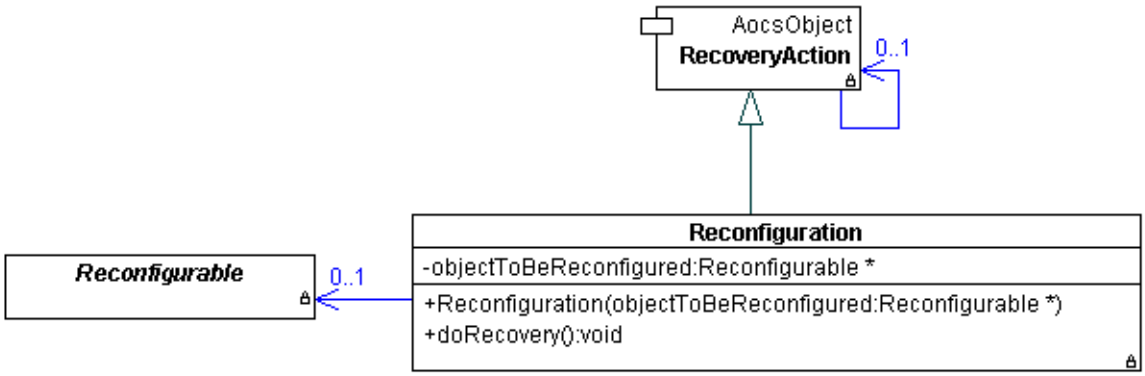


The object to be reset is passed as a parameter to the constructor.

If several objects have to be reset in response to the same failure event, their reset recovery actions can be strung together in a chain.

5.5 Reconfiguration Recovery Action

Reconfigurations are typical responses to failures. The corresponding recovery action object is shown in the figure:



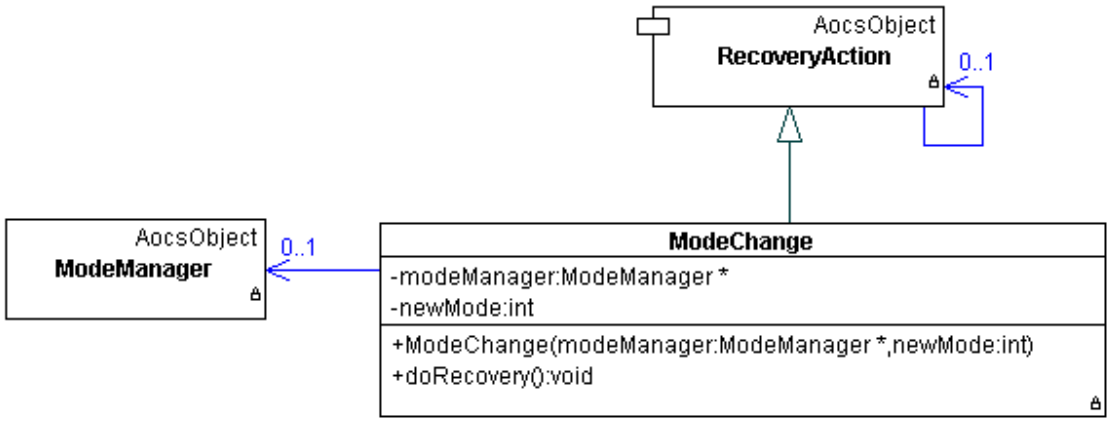
The recovery action holds a reference to an object of type `Reconfigurable`, namely to an object that can be reconfigured. The implementation of `doRecovery` calls the `reconfigure` method on it and causes the reconfiguration to be performed.

The object to be reconfigured is passed as a parameter to the constructor.

If several objects need to be reconfigured, this effect can be achieved by stringing together several reconfiguration actions.

5.6 Mode Change Recovery Action

Operational mode changes – including fall-backs to safe and survival modes – are another common response to failure events. Mode changes are handled by the following class:



Mode changes are performed by acting on mode managers. The mode manager to be acted upon and the target operational mode are passed as parameters to the constructor.

Chaining of recovery actions is again possible to perform several mode changes on several objects.

Note that in the AOCS framework, operational mode is a property of individual objects rather than of the AOCS as a whole. If it is desired to effect a mode change throughout the AOCS, the AOCS mission mode can be changed by acting on the AOCS mission manager object.

5.7 Null Recovery Action

The failure reporting mechanism requires that to each failure a recovery action be associated. Failure events that have been created without a recovery action will give rise to a failure when they are processed by the failure recovery manager.

Sometimes, however, it is not possible for the AOCS developer to associate a specific recovery action to certain failures. In order to cater for such contingencies, a *null failure recovery action* is predefined. This is a recovery action whose implementation of method `doRecovery` does nothing and simply hands over to the next recovery action in the chain (if one exists).

Null recovery actions are obtained as instance of class `NullRecoveryAction`.

5.8 Failures during Recoveries

Method `doRecovery` may call on other methods offered by other classes to implement its recovery action. These methods may in turn encounter failure situations that will cause the generation of failure events. Thus, the execution of recovery from a failure may itself give rise to the generation of a failure event. The latter may have its own recovery action associated to it. This situation can arise because the present version of the AOCS framework foresees a single level of failure detection and recovery and requires some care on the part of the application developed in the association of recovery actions to failures. See RD2 for a more detailed discussion of an alternative approach.

5.9 Recovery Actions with Memory

Recovery actions can be endowed with “memory”. Consider for instance the case of a Kalman Filter to which a recovery action is associated that is triggered when the filter diverges. The nominal recovery for a filter divergence may be a filer reset. However, the recovery action object may be made to remember the last time it was called and, if it finds that it is called too frequently, it can decide that there is a fundamental control failure and may react by commanding a mode fall-back.

Recovery actions with memory can be derived by extending through inheritance the basic components provided by the framework. As a simple example, consider the case of the recovery action for the Kalman filter divergence. An outline implementation could be:

```
class ObjectResetWithCheck : public ObjectReset {

    AocsTime timeOfLastReset;
    AocsTime threshold;
```

```
public :

     . . .

     void doRecovery() {
          if ( (current time – timeOfLastReset)>threshold )
               ObjectReste::doRecovery();
          else
          {    . . .     // failures were too close together, take
               . . .     // some more drastic recovery action
          }

          timeOfLastReset=current time;
     }

     . . .
}
```

Thus, the implementation of the `doRecovery` uses the implementation in the base class under nominal condition but introduces some special action in case two successive failures occur at too close an interval.

5.10 Telemetry Interfaces

Failure recovery actions may in principle implement complex algorithms. The objects that represent them are therefore derived from `AocsObject` to make it possible to reset them and write their state to telemetry.

The data sent to the telemetry stream by an object of the base class `RecoveryAction` in each telemetry mode are summarized in the table:

TM Format	TM Data
Short	none
Normal	enabled status
Long	Normal TM + instance ID of `nextRecoveryAction`
Debug	same as Long TM

Class `SystemReset` does not add any class-specific telemetry data.

Class `ObjectReset` adds the following class-specific telemetry data:

TM Format	TM Data
Short	none
Normal	none
Long	instance ID of object to be reset
Debug	same as Long TM

Class `Reconfigurable` adds the following class-specific telemetry data:

TM Format	TM Data
Short	none
Normal	none
Long	instance ID of object to be reconfigured
Debug	same as Long TM

Class `ModeChange` adds the following class-specific telemetry data:

TM Format	TM Data
Short	none
Normal	none
Long	instance ID of mode manager and target mode
Debug	same as Long TM

In all cases, calls to telemetry methods are propagated along a chain of linked recovery actions.

5.11 The Reset and Configurable Interfaces

Recovery actions inherit from `AocsObject` the `Resettable` and `Configurable` interfaces and must therefore implement the corresponding method.

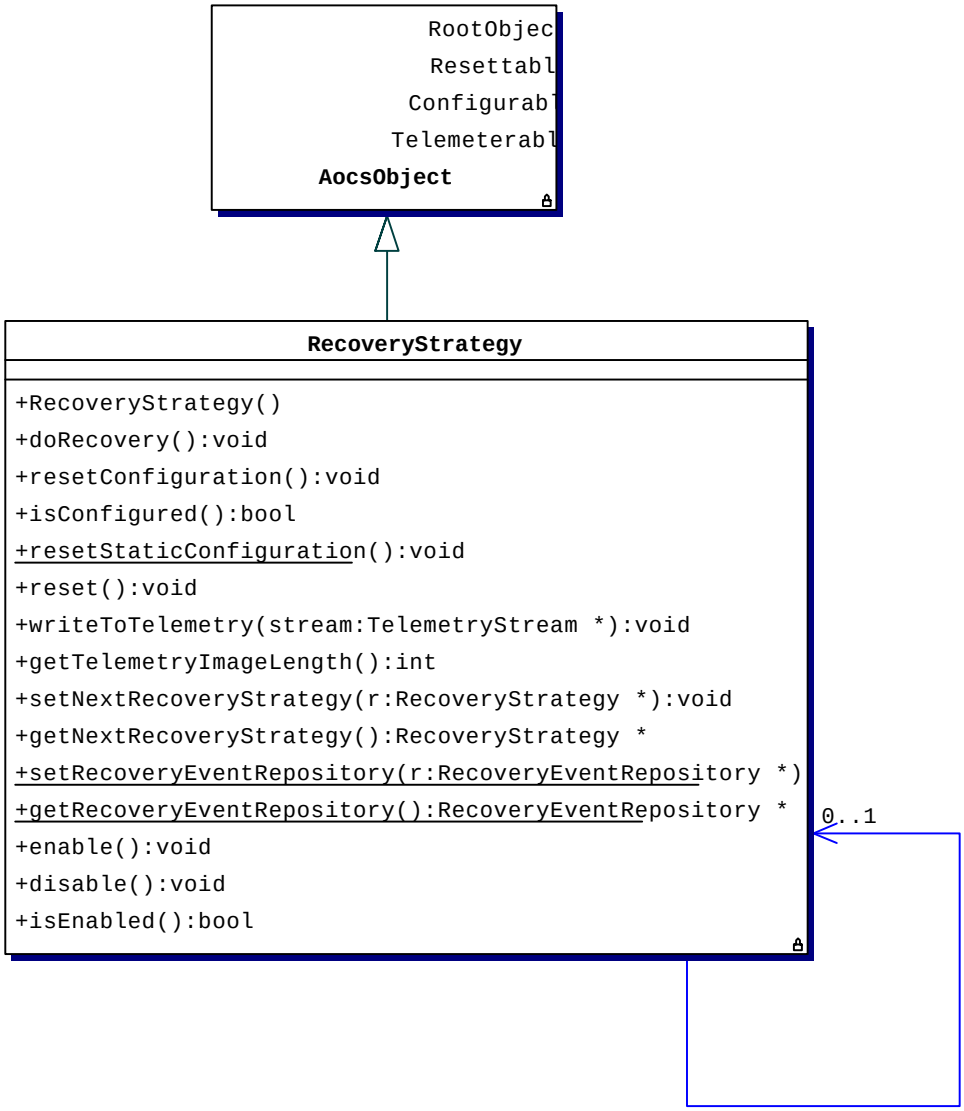
A call to method `reset` on a recovery action causes its enable status to be set to `Enabled`.

A call to method `resetConfiguration` unloads the next recovery action in the chain of linked recovery actions.

6 FAILURE STRATEGIES

A *failure strategy* is a set of coordinated responses to the failure events in the failure event repository.

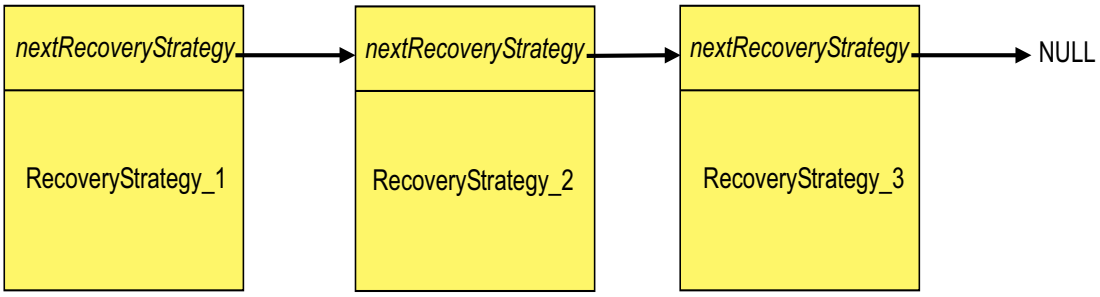
Failure strategies are encapsulated in objects instantiated from subclasses of the base class `FailureStrategy`:



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

`doRecovery()`
Implement the recovery strategy encapsulated by the component.
`enable(), disable(), isEnabled()`
Recovery strategies can be enabled or disabled. These operations allow the enable status of the recovery strategy to be changed and to be queried. Execution of operation `doRecovery` on a disabled recovery strategy is equivalent to a no-op.
`getNextRecoveryAction(), setNextRecoveryAction()`
Recovery strategies can be chained (see below). These are the getter and setter methods for the next recovery actions in the link chain.
`getRecoveryEventRepository(), setRecoveryEventRepository ()`
Recovery strategies can generate recovery events. These are the getter and setter methods for the recovery event repository.

Recovery strategies can be linked in a linear chain as shown in the picture:



The implementation of `doRecovery` provided by the base class `RecoveryAction` is as follows:

```
if (nextRecoveryAction != NULL)
     nextRecoveryAction->doRecovery();
```

A recovery strategy may or may not hand over to the next recovery strategy in the chain depending on some internally-determined condition. Thus, for instance, recovery strategy `SystemResetOnTooManyFailures` (see section 6.1 below) first checks the number of

failure events in the event repository. If this above a certain threshold, it commands a system reset and then returns. If, however, the number of failure events is below the threshold, then the failure strategy hands over to the next failure strategy in the chain.

A typical implementation of `doRecovery` in a subclass of `RecoveryStrategy` implementing a hand-over to the next recovery strategy in the chain is as follows:

```
void doRecovery() {
     . . .                          //perform class-specific recovery strategy
     if appropriate
          RecoveryAction::doRecovery(); //hand-over to next strategy in chain
}
```

Thus, a recovery strategy has the option either to hand over to the next strategy in the chain or to interrupt the recovery process.Note that this means that the *order* in which recovery strategies are linked together is significant.

6.1 Types of Failure Strategies

`FailureStrategy` is the base class for all recovery strategies. By itself, it does not define any concrete action. Concrete recovery actions are defined by subclasses of `RecoveryAction`. Typical failure recovery strategies would include:

- *Sequence of local recovery actions*

  This strategy retrieves from the event repository the failure events generated since the last call to `doRecovery` and performs the recovery actions associated to each event.

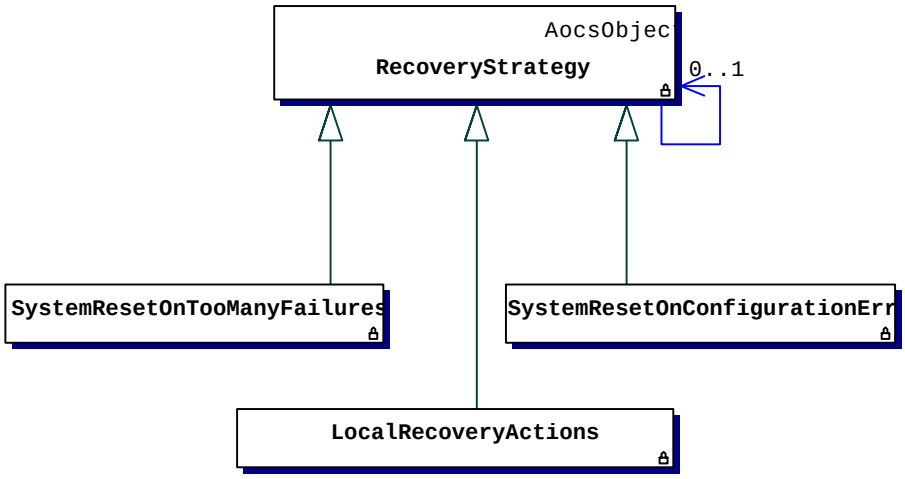
- *System reset on too many failures*

  This strategy checks the number of failure events in the repository generated since the last call to `doRecovery` and if it finds that it exceeds a predefined threshold, it commands a system reset. The system reset is performed as a service request to object `SystemReset`.

- *System reset on configuration error*

  This strategy checks the number of configuration events in the repository generated since the last call to `doRecovery` and if it finds that any configuration errors have occurred, it commands a system reset. The system reset is performed as a service request to object `SystemReset`.

The class diagram for these concrete recovery strategies is shown below:



When objects instantiated from class `LocalRecoveryActions` find a failure event without its associated recovery action, they generate a failure event. Getter and setter methods for the associated recovery action are offered by the class.

6.2 Telemetry Interfaces

Failure recovery strategies may in principle implement complex algorithms. The objects that represent them are therefore derived from `AocsObject` to make it possible to reset them and write their state to telemetry.

The data sent to the telemetry stream by an object of the base class `RecoveryStrategy` in each telemetry mode are summarized in the table:

TM Format	TM Data
Short	none
Normal	enabled status
Long	Normal TM + instance ID of `nextRecoveryStrategy`
Debug	same as Long TM

Class `LocalRecoveryActions` adds the following class-specific telemetry data:

TM Format	TM Data
Short	none

Normal	none
Long	instance ID of recovery action
Debug	Long TM + failure event counter

Class `SystemResetOnTooManyFailures` adds the following class-specific telemetry data:

TM Format	TM Data
Short	none
Normal	none
Long	Threshold for the number of failure events triggering a system reset
Debug	Long TM + failure event counter

In all cases, calls to telemetry methods are propagated along a chain of linked recovery strategies.

6.3 The Reset and Configurable Interfaces

Recovery strategies inherit from `AocsObject` the `Resettable` and `Configurable` interfaces and must therefore implement the corresponding method.

A call to method `reset` on a recovery strategy causes its enable status to be set to `Enabled`.

The recovery strategies defined in section 6.1 act on the number of failure events generates since the last time `doRecovery` was called. They therefore need to keep track of the failure events counter across activations of `doRecovery` (variable `lastEventCounter`). Calls to their methods `reset` causes this variable to be set to the current value of the failure event counter.

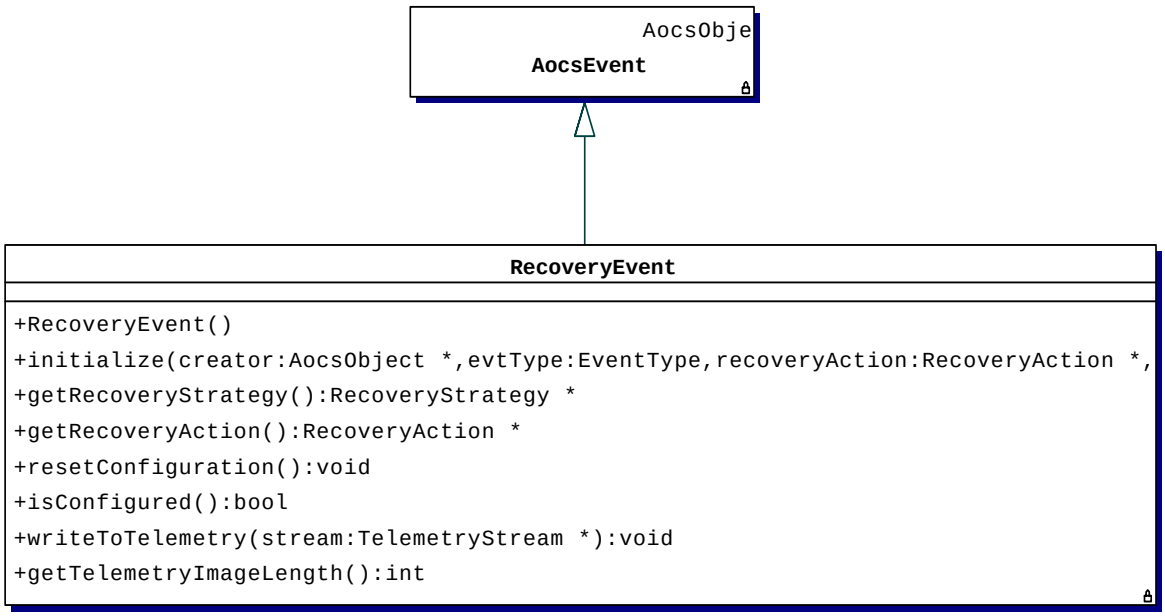
A call to method `resetConfiguration` on class `RecoveryStrategy` unloads the next recovery strategy in the chain of linked recovery strategies.

A call to method `resetConfiguration` on class `SystemResetOnTooManyFailures` causes the failure event threshold to be reset to zero.

A call to method `resetConfiguration` on class `LocalRecoveryActions` causes the recovery action associated to this class to be unloaded.

7 FAILURE RECOVERY EVENT

The execution of a failure recovery action or of a failure strategy strategy triggers the creation of an event of type `RecoveryEvent`. The class diagram for its class is:



Thus, recovery events add the following attributes to those defined by the base class `AocsEvent`:

- `recoveryStrategy`: pointer to recovery strategy object.
- `recoveryAction`: pointer to recovery action object

The creation of the recovery events is the responsibilities of the recovery strategies. Neither the recovery actions nor the recovery manager create any recovery events.

7.1 The Telemetry Interface

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

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

TM Format	TM Data

Short	none
Normal	none
Long	instance identifier of recovery strategy and recovery action
Debug	same as long TM

7.2 The Reset and Configurable Interface

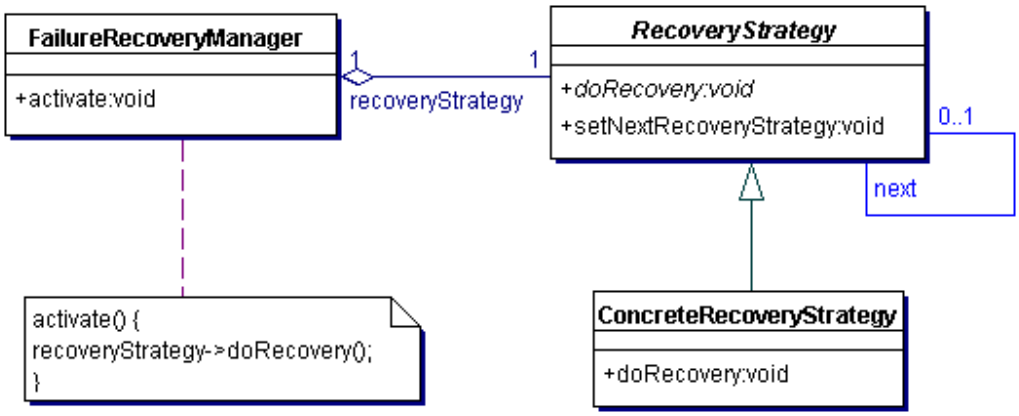
Recovery event objects inherit from `AocsObject` the `Resettable` and `Configurable` interfaces and must therefore implement the corresponding method.

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

Recovery events define a class-specific `resetConfiguration` method that resets all event attributes to zero. Method `isConfigured` returns true if the reference to the recovery strategy is non-NULL.

8 THE FAILURE RECOVERY DESIGN PATTERN

This design pattern is introduced to address the problem of separating the management of failure recovery from the implementation of failure recovery actions and strategies. The design pattern is illustrated in the figure:



The failure recovery manager is essentially based on the *chain of responsibility* design pattern from RD1 but it can also be seen as an instance of the manager meta-pattern of RD2 where the list of functionality implementers only contains one element.

In the classical version of the chain of responsibility pattern, the client's request (in this case, the request to perform a recovery) is passed along the chain of handlers (the recovery strategies) until one is found who is able to handle it. Each request is intended to be handled by only one handler. In the application of the pattern to failure recovery, however, a recovery strategy when it receives a recovery requests performs the following actions:

- it handles the recovery request, and
- it checks whether the recovery request should be passed on to the next recovery strategy or whether recovery processing should terminate.

The recovery strategies are therefore executed in sequence but every recovery strategy has the chance to interrupt the chain. This incidentally means that the *order* in which the recovery strategies are linked in the list is important.

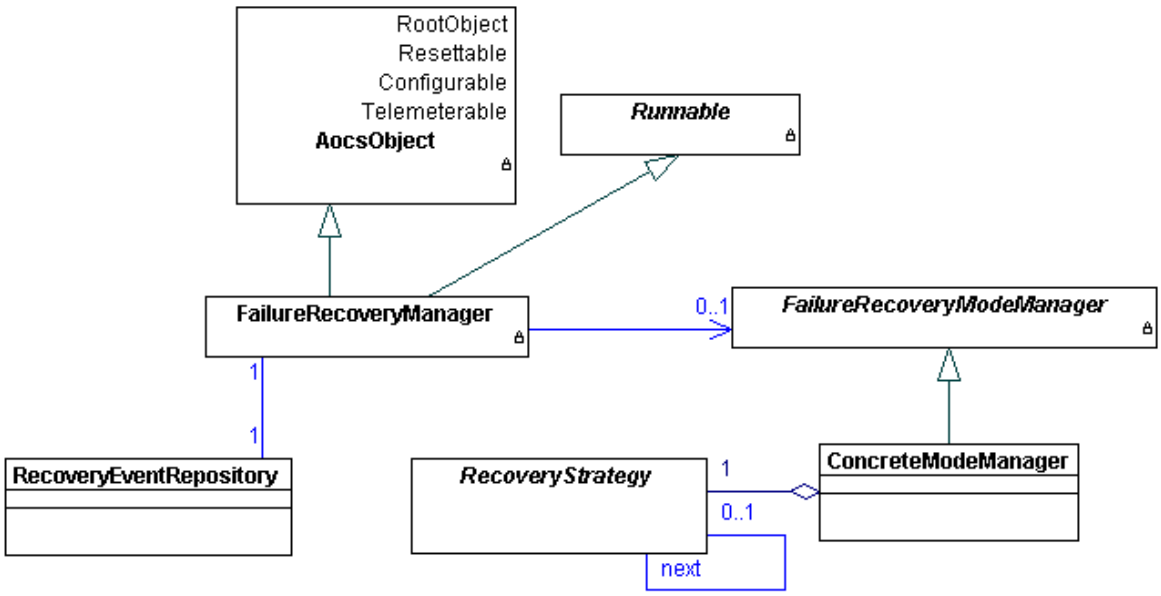
It would have been possible to implement failure recovery using a more straightforward version of the manager meta-pattern where the recovery strategies are arranged in a list and the recovery manager, when it is activated, goes through the list and executes each strategy in sequence. This architecture, however, would have made it more awkward to give each a recovery strategy the option to interrupt the recovery process.

8.1 Instantiation of Failure Recovery Pattern

The failure recovery pattern is instantiated as follows for the framework:

- the failure recovery manager is an active object and its `activate` method is the `run` method declared by interface `Runnable`.
- Recovery events are created for each recovery strategy and recovery action that is executed.
- In most cases, the recovery strategy to be executed depends on operational conditions. This is taken into account by making the failure recovery manager mode-dependent (see section 8.3). The failure recovery mode manager then manages a single strategy corresponding to the recovery strategy to be supplied to the recovery manager.

The resulting class diagram is:



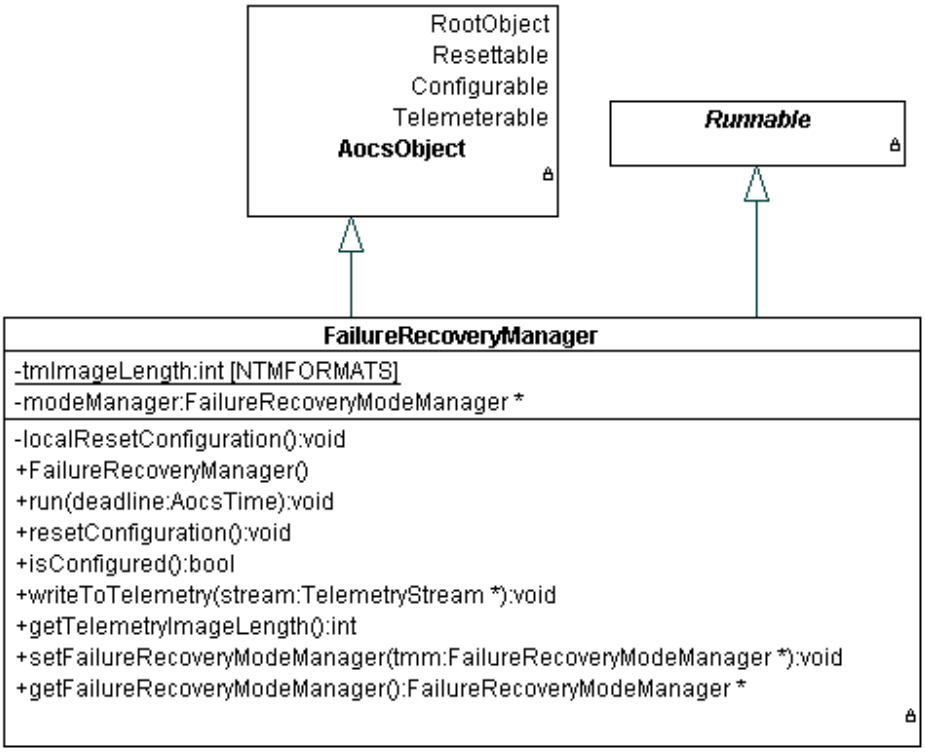
The mode manager is characterized by a dedicated abstract interface as discussed in section 8.3.

8.2 The Failure Recovery Manager

The failure recovery manager is the active component that is responsible for processing the events in the failure event repository and responding to them with appropriate failure recovery strategies.

The failure recovery manager only sees recovery strategies. Failure recovery actions are, where required, managed and implemented by the recovery strategies.

The failure recovery manager is instantiated from class `FailureRecoveryManager` shown in the next figure:



The failure recovery manager is derived from `AocsObject` and implements interface `Runnable` to signify that it is an active object.

The failure recovery manager sees the recovery strategy base class but, behind it, there may be a string of concrete failure recovery strategies.

As discussed in section 8.3, the failure recovery manager obtains the failure recovery strategy appropriate to current operating conditions from a *failure recovery mode manager*.

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

`setFailureRecoveryModeManager, getFailureRecoveryModeManager`
Setter and getter methods for the failure recovery mode manager.

The basic implementation of method `run` (the entry point for the task associated to the failure recovery manager) is very simple and is outlined in the pseudo-code below:

```
void FailureRecoveryManager::run(AocsTime deadline)
{
     // Load the recovery strategy
     RecoveryStrategy* recStr = modeManager->getRecoveryStrategy();

     // Implement the recovery strategy
     recStr->doRecovery();
}
```

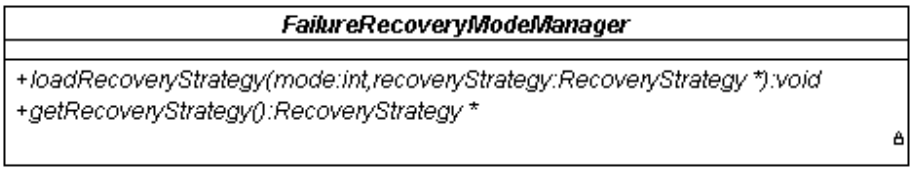
Thus, the recovery manager gets the recovery strategy from its mode manager and then executes it. Obviously, the failure recovery strategy may actually consist of a string of linked recovery strategies.

8.3 Failure Recovery Mode Manager

The type of failure recovery strategy may depend on operational conditions. This dependency is modelled by endowing the failure recovery manager with operational mode.

The mode manager is constructed in accordance with the mode management pattern prescribed in RD3 as an interface and a concrete class.

The failure detection mode manager must be able to supply to the failure recovery manager the failure recovery strategy. It is accordingly characterized by the following interface:

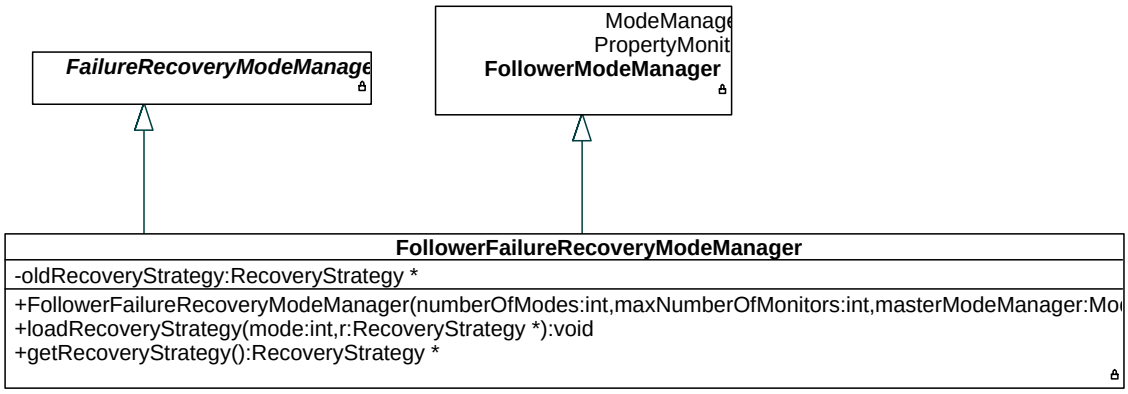


The semantics of the operations defined by this interface are summarized in the following table:

`getRecoveryStrategy()`
This method is called by the failure recovery manager to retrieve the currently valid recovery strategy.
`loadRecoveryStrategy (int i, RecoveryStrategy* recStr)`
This method is used to configure the failure recovery mode manager. It associates the recovery strategy `recStr` to operational mode i.

Concrete failure recovery mode managers are defined by the mechanism that they use to decide which particular recovery strategy should be returned at any given point in time.

The prototype framework provides a default failure detection mode manager that is based on the follower mode manager. This default failure detection mode manager is instantiated from the following class `FollowerFailureRecoveryModeManager`:



Thus, the default failure recovery mode manager uses the services offered by the generic follower mode manager component exported by the operational mode framelet.

8.4 Recursion

Use of the chain of responsibility design pattern introduces the possibility of recursion. A call to method `RecoveryStrategy::doRecovery` can be recursive if several recovery strategies are linked together. The maximum depth of the recursion is given by the maximum number of recovery strategies that are linked together.

Recursion can also arise because of the way recovery actions are linked together. A call to method `RecoveryAction::doRecovery` can be recursive if several recovery strategies are linked together. The maximum depth of the recursion is given by the maximum number of recovery actions that are linked together.

9 FRAMELET HOT-SPOTS

This section classifies the framelet hot-spots defined in the previous sections of this document. The classification is as described in RD4.

9.1 Failure Recovery Mode Manager Plug-In

Name: Failure Recovery Mode Manager **Plug-In**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `FailureRecoveryManager` class (method `setFailureRecoveryModeManager`)
Pre-defined Options: `FollowerFailureRecoveryModeManager` component exported by this framelet.
Related Hot-Spots: none
Description Failure recovery managers need a mode manager to supply them with a recovery strategy. This hot-spot allows the failure recovery mode manager to be loaded in the failure recovery manager.

9.2 Recovery Action Hot-Spot

Name: Recovery Action Hot-Spot
Visibility Level: framework -level
Adaptation Time: compile-time
Adaptation Method: overriding of method `doRecovery` in class `RecoveryAction`
Pre-defined Options: recovery action components exported by this framelet
Related Hot-Spots: none
Description Responses to individual failure events are encapsulated in instance of subclasses of

`RecoveryAction` with each subclass representing one type of response. Derivation of new subclasses usually requires on method `doRecovery` to be overridden.

9.3 Recovery Strategy Hot-Spot

Name: Recovery Strategy Hot-Spot
Visibility Level: framework -level
Adaptation Time: compile-time
Adaptation Method: overriding of method `doRecovery` in class `RecoveryStrategy`
Pre-defined Options: recovery strategy components exported by this framelet
Related Hot-Spots: none
Description Responses to the set of failure events in the failure event repository are encapsulated in instance of subclasses of `RecoveryStrategy` with each subclass representing one type of response. Derivation of new subclasses usually requires on method `doRecovery` to be overridden.

9.4 Recovery Event Repository Plug-In

Name: **Recovery Event Repository Plug-In**
Visibility Level: framelet-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `RecoveryStrategy` class (method `setRecoveryEventRepository`)
Pre-defined Options: `RecoveryEventRepository` component exported by inter-component communication framelet.
Related Hot-Spots: none
Description Recovery strategy objects log execution of their strategies as events stored in the recovery event

repository. This hot-spot allows the recovery event repository component to be loaded. Note that this component is loaded as a `static` reference.