TELECOMMAND 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 telecommand management framelet. This framelet defines an interface to which telecommands must conform and defines an architecture for the telecommand manager. The framelet enhances reusability because it decouples the task of managing the telecommands from the task of executing them.

Written By: A. Pasetti

Date: 30 April 2002

Issue: 2.3

Reference: SWE/99/AOCS/014

TABLE OF CONTENTS

1 REFERENCES

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

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

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

2 ACRONYMS

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

3 INTRODUCTION

This document describes the *AOCS telecommand 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 interface to which telecommands must conform and defines an architecture for the telecommand manager. The framelet enhances reusability because it decouples the task of *managing the telecommands* from the task of *executing them*.

3.1 Context

The context for the design of the framelet is described in RD2. The present document assumes that the reader is familiar with RD2 and in particular with the section dealing with telecommand management.

The architecture proposed here follows the general concept outlined in RD2. However, the telecommand loader provided with the framework prototype only provides data loading capability (ie. no code loading).

RD2 discussed the possibility of merging the telecommand and manoeuvre concepts by turning telecommands into special instances of AOCS manoeuvres. This option was rejected because the concepts of telecommand transaction does not have a counterpart in the manoeuvre management concept and because telecommands, unlike manoeuvres, are loaded dynamically and this gives rise to special problems that need special treatment.

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 telecommand event repository hot-spot (section 11.5) is not applicable to the Java framework. Event repositories are event listeners and can be linked to the telecommand manager through the associated `addListener` methods.

3.3 Notation

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

UML class diagrams were obtained with the *Together* tool (version 4.0).

4 FRAMELET CONSTRUCTS

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

TELECOMMAND MANAGEMENT FRAMELET
Framelet Design Patterns
Telecommand Transaction : design pattern to handle sequences of telecommands as a single entity
Framelet Interfaces
`TelecommandLoader` : interface for the telecommand loader
Framelet Core Components
`Telecommand` : base class for telecommands `TelecommandTransaction` : base class for transaction telecommands `TelecommandManager` : telecommand manager component
Framelet Default Components
`ModeChangeTelecommand` : simple telecommand to change the mode of a mode manager `ModeChangeTransactionTelecommand` : transaction telecommand to change the mode of a mode manager `TelemetryFormatTelecommand` : simple telecommand to change the format of a telemeterable object `ManoeuvreTelecommand` : simple telecommand to load a parameterless manoeuvre in the manoeuvre manager `AttitudeSlewTelecommand` : simple telecommand to configure and load an attitude slew manoeuvre `TelemetryFormatTransactionTelecommand` : transaction telecommand to change the format of a telemeterable object `ReconfigureTelecommand` : simple telecommand to command a reconfiguration to a reconfiguration manager `ReconfigureTransactionTelecommand` : transaction telecommand to command a

reconfiguration to a reconfiguration manager

`VsDmaTelecommandLoader` : DMA-based telecommand loader for the Visual Studio environment.

`Erc32DmaTelecommandLoader` : DMA-based telecommand loader for the ERC32 environment with the GNU compiler

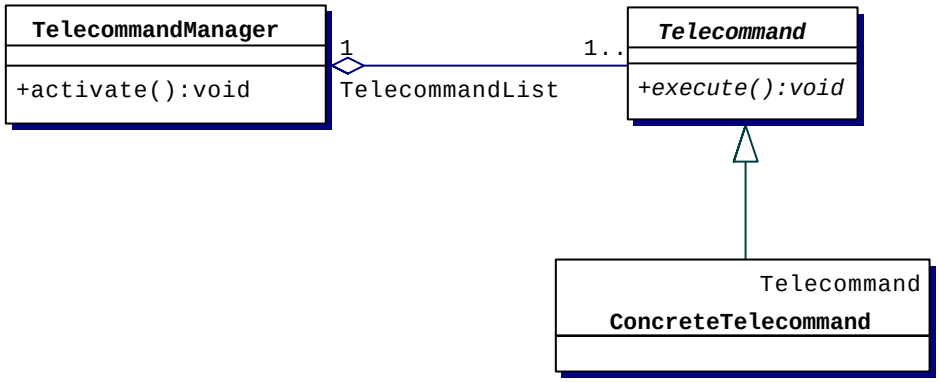
The constructs listed above are those provided by the framework prototype. More advanced versions of the framework may include additional constructs. In particular, they may include more telecommand and transaction telecommand classes.

5 THE TELECOMMAND MODEL

The telecommand model adopted by the AOCS framework is based on the telecommand design pattern described in the next subsection and on the concept of telecommand loading described in sub-section 5.3.

5.1 The Telecommand Management Design Pattern

The telecommand management design pattern is introduced to address the problem of separating telecommand management from telecommand implementation. It is closely based on the *command design pattern* from RD1 as illustrated in the figure:



The telecommand manager maintains a list of pending telecommands and, when it is activated, it goes through the list and executes all telecommands in sequence.

The telecommand pattern can also be seen as an instance of the manager meta-pattern where telecommand execution is the pattern functionality, the telecommand manager is the functionality manager, and the `Telecommand` class decouples the functionality management from the functionality implementation.

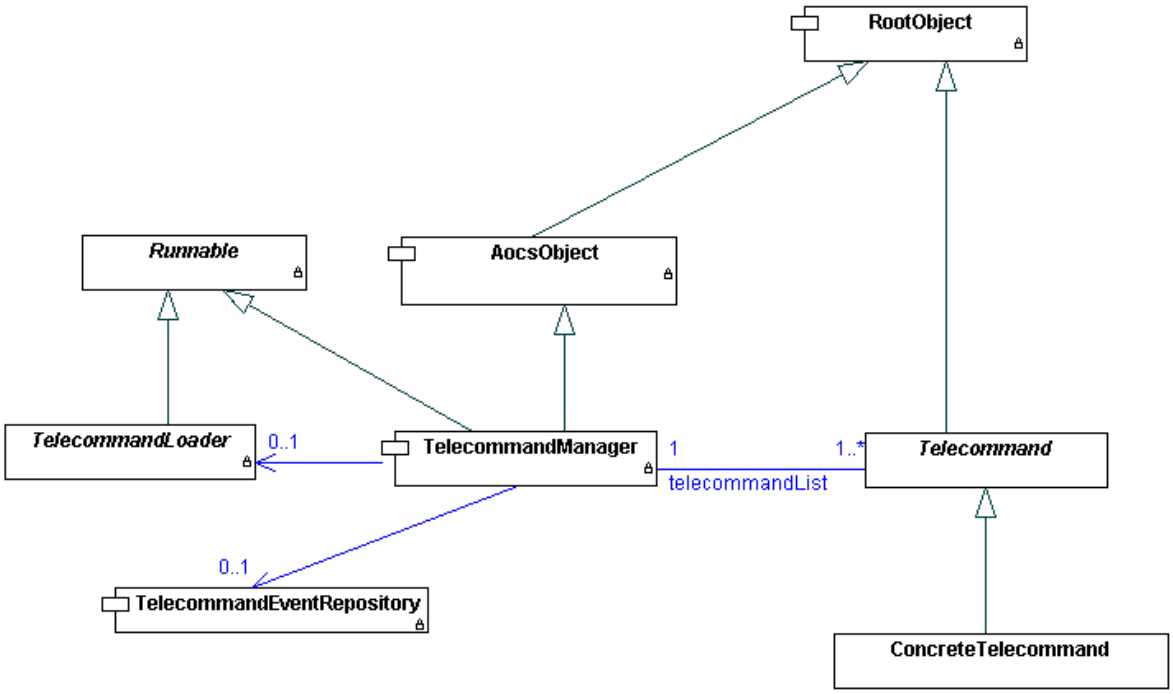
5.2 Telecommand Management Pattern Instantiation

The telecommand pattern is instantiated in the AOCS framework as follows:

- The telecommand manager is implemented as an active object and its `activate` method is the `run` method it inherits from interface `Runnable`.

- Telecommand events are created when telecommands are executed, when their execution fails, when they loaded in the telecommand manager, etc.
- The `Telecommand` interface is replaced by a concrete class with overridable methods. This allows it to encapsulate some functions that are common to all telecommands.
- The base class `Telecommand` is endowed with extra functions in addition to telecommand execution as described in section 6.
- A telecommand loader component is introduced to load telecommands into the telecommand manager (see section 10).

The class diagram of the instantiated telecommand management design pattern is:



Telecommand management in the AOCS framework is based on the command pattern of RD1. Telecommands are packaged as objects exposing a method `execute`. A call to `execute` by the telecommand manager causes the actions associated to the telecommand to be executed.

5.3 Telecommand Loading

Telecommands are loaded dynamically into the AOCS computer memory. The loading mechanism is application-dependent. Two common mechanisms are:

- the telecommand is loaded via DMA under the control of hardware that is external to the AOCS computer. The completion of a load operation is indicated by an interrupt to the AOCS software.
- the telecommand words or bytes are loaded by the AOCS software via I/O commands. The arrival of a new word or byte is indicated by an interrupt to the software.

A telecommand is an object made up of data and code. The link between the data and the code is provided by a virtual function table. Two types of telecommand loading are foreseen:

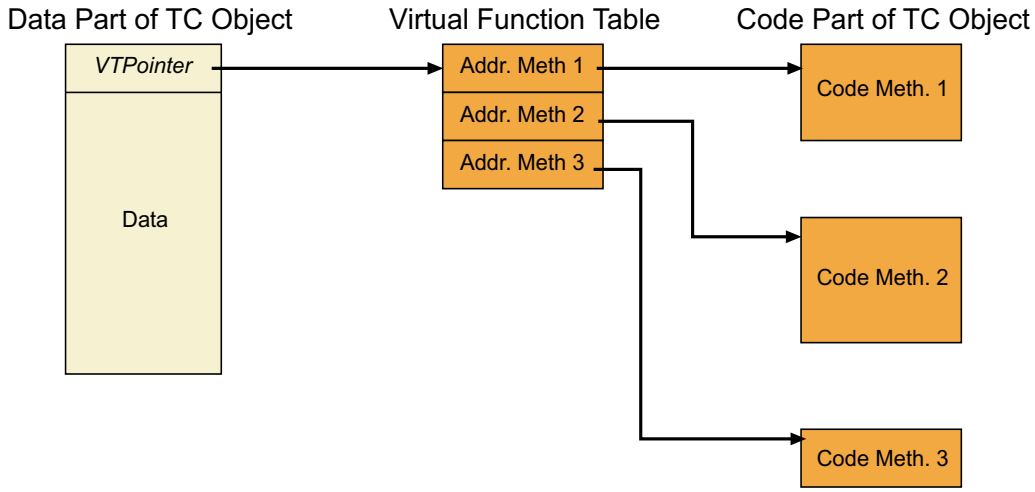
- *Data Load*

  The code for the telecommand is already present in the AOCS software and only the data part of the telecommand object is physically loaded into memory.

- *Full Load*

  Both the code and the data for the telecommand object are loaded.

The two telecommand load types are illustrated in the figure:



In a data load, only the yellow part of the figure is physically loaded onto the AOCS computer memory. In the full load case, both the yellow and red parts are loaded. The example in the figure assumes that the code for all methods required by the telecommand

must be loaded. In practice, some methods will be able to rely on code already present in the AOCS computer (typically, this is because such methods are inherited without changes from a superclass of the concrete telecommand class that is already present in memory).

Normally, an AOCS application will include some predefined telecommand classes implementing common telecommand actions. For such telecommands, only the data need to be loaded. Unusual or unforeseen telecommands need to be loaded in full.

The telecommand loaders components provided by the prototype framework only implement the data load mechanism. The code loaded mechanism is coded for information only (it was not tested).

5.4 Telecommand Management

Telecommand management is performed by two objects:

- *The Telecommand Loader*

  This object is responsible for loading telecommands from the DMA area or from the I/O port and for assembling them as objects of type `Telecommand` that are then passed to the telecommand manager for execution.

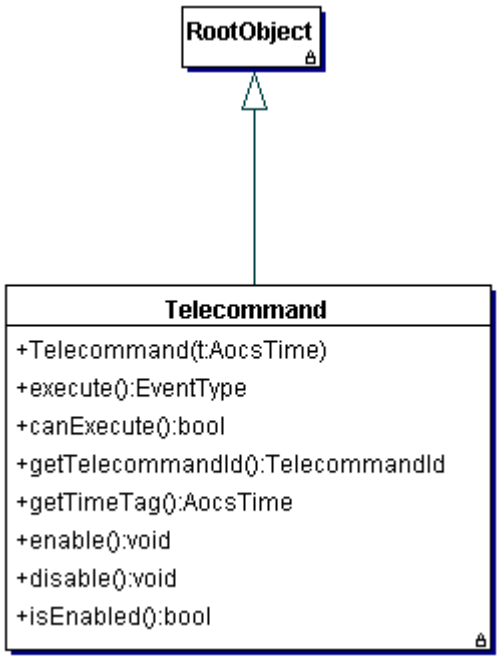
- *The Telecommand Manager*

  This object is responsible for executing the telecommands. It maintains lists of pointers to objects of type `Telecommand` and periodically goes through the list and executes the telecommands.

  Note that the telecommand manager only sees simple telecommands. It has no knowledge of the distinction between simple and transaction telecommands. This distinction is internal to the transaction telecommand class.

6 THE TELECOMMAND CLASS

All telecommands are ultimately instances of the following base class:



Class telecommand is a concrete class with overridable methods and should conceptually be seen as an interface class. The implementations of its methods are trivial default that are intended to be overridden by concrete telecommand classes. Their semantics are as follows:

`Telecommand(t)`
Constructor that sets the telecommand time tag t.
`execute()`
A call to this method causes the actions associated to the telecommand to be executed. The method returns a code indicating the completion status of the telecommand (see section 6.3).
`canExecute()`
Performs the execution check (see section 6.2) and returns `true` if the operational conditions are appropriate to the telecommand execution..

`getTelecommandId()`
Telecommands have an identifier associated to them. This method returns the telecommand identifier. Note that telecommand transactions have a single identifier associated to them: ie all telecommands in a transaction have the same identifier.
`getTimeTag()`
Telecommands have a time tag associated to them. Telecommands are only executed after their time tag is passed. Note that telecommand transactions have a single time tag associated to them: ie all telecommands in a transaction have the same time tag.
`enable(), disable(), isEnabled()`
Telecommands can be enabled and disabled. They are enabled by default. These methods report and set the enable status.

In the present implementation, the telecommand identifier is the same as the instance identifier inherited from `RootObject`. In the case of telecommand transactions, the telecommand identifier is the instance identifier of the first telecommand in the transaction.

6.1 Inheritance Tree Structure

Concrete telecommand classes are derived from class `RootObject` and from a single inheritance trees. They therefore have a single virtual table pointer associated to them which simplifies the implementation of the telecommand loading mechanism and keeps the telecommand size small. This consideration is important for the organization of telecommand loading (see section 5.3).

6.2 Execution Check

Some telecommands should only be executed if certain conditions hold. For instance, a telecommand to perform a reaction wheel unloading in some systems should only be executed if the AOCS is in a thruster-based more.

For this purpose, telecommands expose a `canExecute` method which performs a check that the operational conditions are appropriate to the execution of the telecommand.

6.3 Handling of Telecommand Execution Status

Telecommands are intended to be light-weight objects since they may have to be uplinked with their code. For this reason, they are derived from `RootObject` rather than from `AocsObject` and hence they do not have a default reference to the failure detection repository allowing them to create failure events in response to the detection of errors. Some concrete telecommand classes may be endowed with such a reference and may thus take care of their own error reporting but in general telecommands report their execution status through the return code of method execute. The following return codes are foreseen at present:

TC_SUCCESS	A simple telecommand executed successfully
TC_FAILURE	Simple telecommand failure
TC_TRANS_SUCCESS	A telecommand transaction executed successfully
TC_TRANS_FAILURE	A telecommand transaction failed but the unexecute action was successful and the system was returned to the state in which it was prior to the execution of the telecommand
TC_TRANS_UNDO_FAILURE	A telecommand transaction failed. Unexecution was attempted but failed. The system is in an undefined state.

It is then the responsibility of the telecommand manager to create telecommand events to record the execution status of a telecommand (see section 8).

6.4 Default Telecommands

The framework prototype provides three default telecommands encapsulated by the classes listed in the table:

Telecommand Class	Telecommand Function
`ChangeModeTelecommand`	Telecommand to change the operational mode of a mode manager component
`TelemetryFormatTelecommand`	Telecommand to change the telemetry format of a telemeterable component

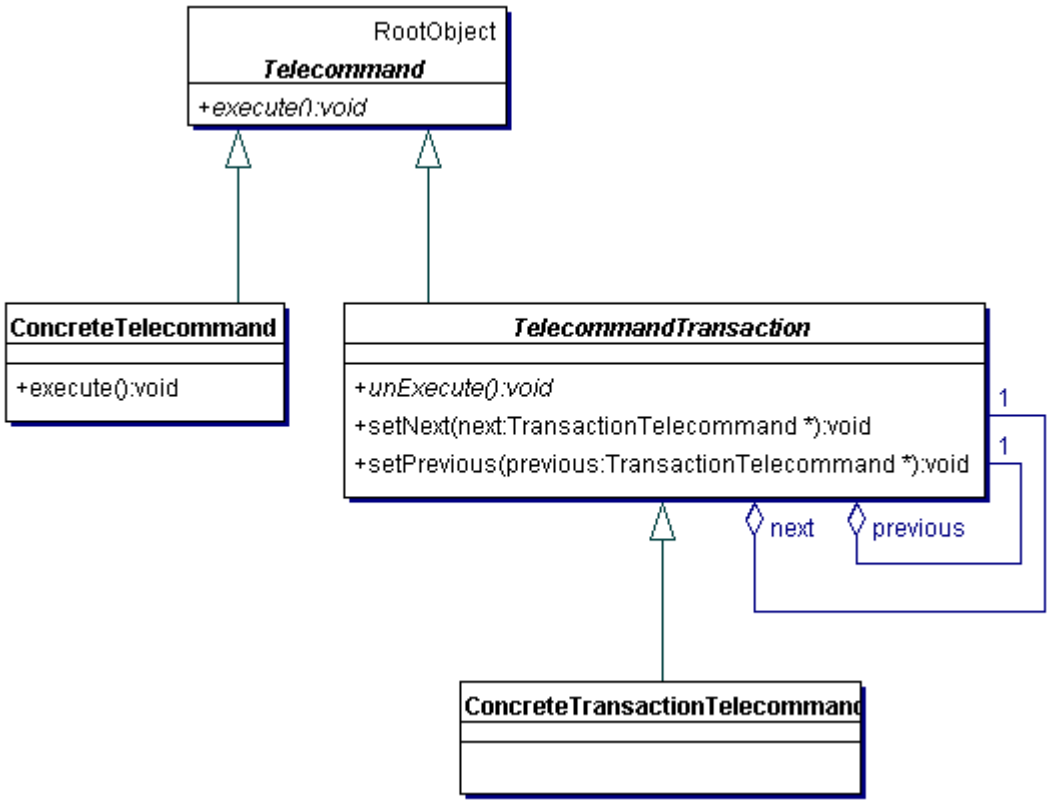
`ReconfigureTelecommand`	Telecommand to start a reconfiguration in a reconfiguration manager
`ManoeuvreTelecommand`	Telecommand to load a parameterless manoeuvre in the manoeuvre manager
`AttitudeSlewTelecommand`	Telecommand to configure and load an attitude slew manoeuvre in the manoeuvre manager

7 THE TELECOMMAND TRANSACTION DESIGN PATTERN

A *telecommand transaction* encapsulates a sequence of telecommands that can be treated as a transaction. This means that in case of failure of a telecommand in the sequence, telecommands that have already been executed can be unexecuted and the AOCS can be returned to the state in which it was at the time the transaction began.

Telecommand transactions are made up of a sequence of linked telecommands that are executed as a single entity. The individual telecommands in a telecommand transaction are called *transaction telecommands.*

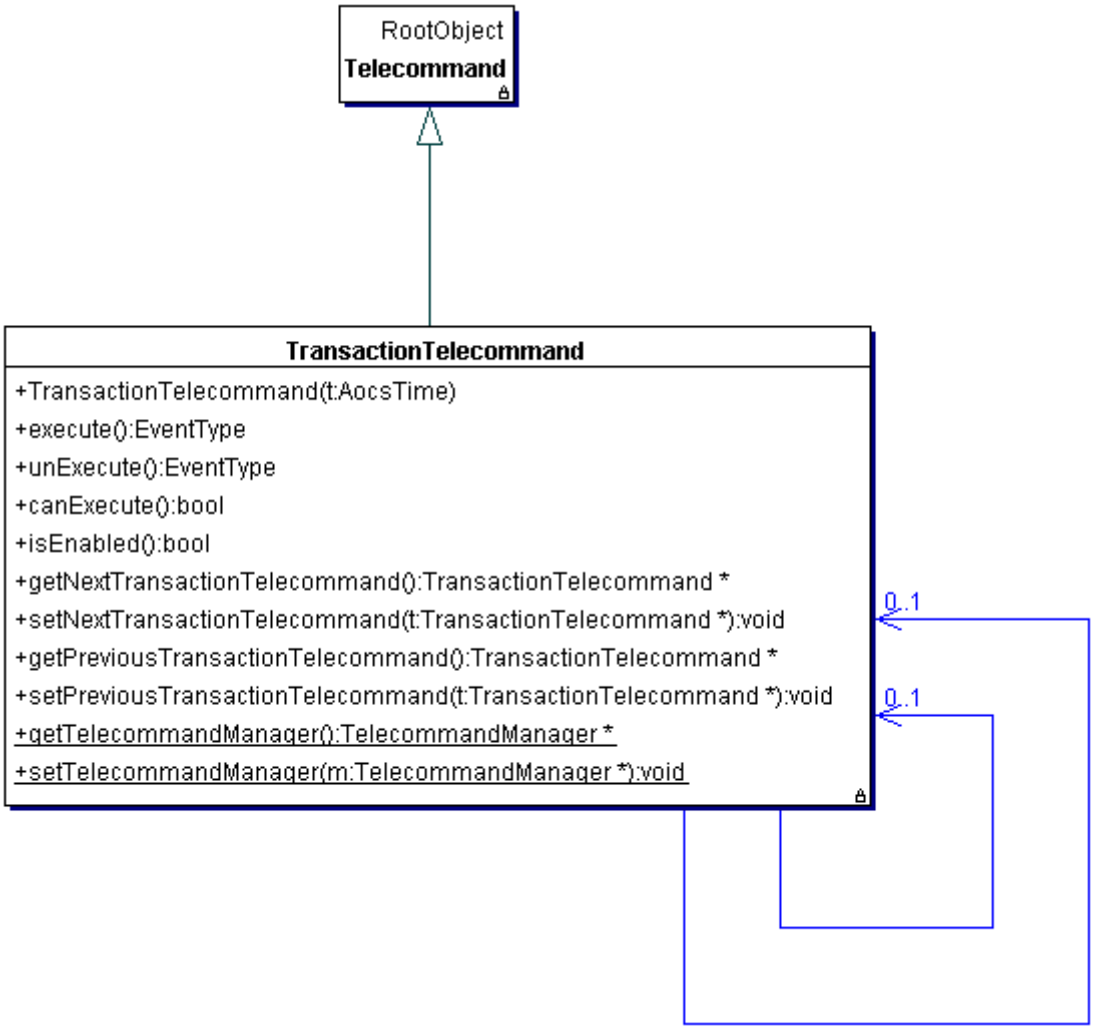
The AOCS framework introduces the *telecommand transaction design pattern* to address the problem of treating some telecommands or some sequences of telecommands as telecommand transactions. The pattern calls for the introduction of a class `TransactionTelecommand` that extends calls `Telecommand` with an `unExecute` method that "undoes" the actions of the telecommand:



Note that, as shown in the class diagram, telecommand transactions must be chained in a link to allow their manager to recursively undo all telecommands in the same transaction.

7.1 Telecommand Transaction Pattern Instantiation

The telecommand transaction pattern is instantiated by replacing the abstract interface for trasnsaction telecommands by a concrete class derived from class `Telecommand`. Class `TransactionTelecommand` is defined as follows:



The `TelecommandTransaction` class extends class `Telecommand` in the following ways:

- It adds attributes to link the transaction telecommands into a double-linked list.
- It adds an `unExecute` method to undo the action encapsulated by the telecommand.
- It provides an implementation of method execute to allow all the telecommands in the transaction to be executed as a single entity (see section 7.2).
- It adds a static reference to the telecommand manager.

The semantics of the methods specific to class `TransactionTelecommand` are as follows:

`TransactionTelecommand(t)`
Constructor that sets the transaction telecommand time tag t.
`execute()`
Recursively execute all the telecommands in the telecommand transaction. The method returns a code indicating the completion status of the telecommand (see section 6.3).
`canExecute()`
Recursively calls `canExecute` on all telecommands in the transaction and returns `true` if all `canExecute` return `true`.
`getNextTransactionTelecommand, setNextTransactionTelecommand, getPreviousTransactionTelecommand, setPreviousTransactionTelecommand`
Methods to manage the linking of telecommands in a transaction.
`setTelecommandManager(), getTelecommandManager()`
Static getter and setter methods for the telecommand manager
`isEnabled()`
Recursively calls `isEnabled` on all telecommands in the transaction and returns `true` if all `isEnabled` return `true`.

7.2 Telecommand Transaction Execution

Class `TransactionTelecommand` provides default implementations of methods `execute` and `unExecute` that take care of checking the success or otherwise of a telecommand and, if necessary, undo its action by calling `unExecute`.

These implementations are such that the telecommand manager need not be aware of the distinction between telecommands and telecommand transactions. The logic for ensuring that telecommand transactions are handled as a single entity is coded in the implementations of methods `execute` and `unExecuted` provided by class `TransactionTelecommand`:

```
EventType TransactionTelecommand::execute()
{
```

```
     if (next!=NULL)
          return next->execute();
     else
          return TC_TRANS_SUCCESS;
}

EventType TransactionTelecommand::unExecute()
{
     if (previous!=NULL)
          return previous->unExecute();
     else
          return TC_TRANS_FAILURE;
}
```

Here `next` and `previous` are pointers to the next and the previous telecommands in a transaction.

Sample implementations of the `execute` and `unExecuted` methods in concrete transaction telecommands are as follows:

```
EventType ConcreteTransactionTelecommand::execute()
{
     . . . // Recover current state information and put on the stack

     . . . // Attempt to perform telecommand action

     if (telecommand action succeeded)
          return TransactionTelecommand::execute();
     else
     {    . . . // restore state
          return TransactionTelecommand::unExecute();
     }
}

EventType ConcreteTransactionTelecommand::unExecute()
{
     . . .  // Recover state information from the stack

     . . .  // Restore state

     if (state restore operation was successful)
          return TransactionTelecommand::unExecute();
     else
          return TC_TRANS_UNDO_FAILURE;
}
```

Since transaction telecommands must be capable of undoing their own actions, they need some mechanism to save the system state prior to their execution. Normally, this information would be saved as part of the telecommand's internal data. However, in this case, there is a requirement to keep the image of the telecommand objects as small as possible. Hence, the telecommand manager offers a stack where transaction telecommands can save state information (see section 9). Pop and push methods are provided for integers and real quantities together with stack reset methods.

7.3 Default Transaction Telecommands

The framework prototype provides three default transaction telecommands encapsulated by the classes listed in the table:

Transaction Telecommand Class	Telecommand Function
`ChangeModeTransactionTelecommand`	Transaction telecommand to change the operational mode of a mode manager component
`TelemetryFormatTransactionTelecommand`	Transaction telecommand to change the telemetry format of a telemeterable component
`ReconfigureTransactionTelecommand`	Transaction telecommand to start a reconfiguration in a reconfiguration manager

7.4 Recursion

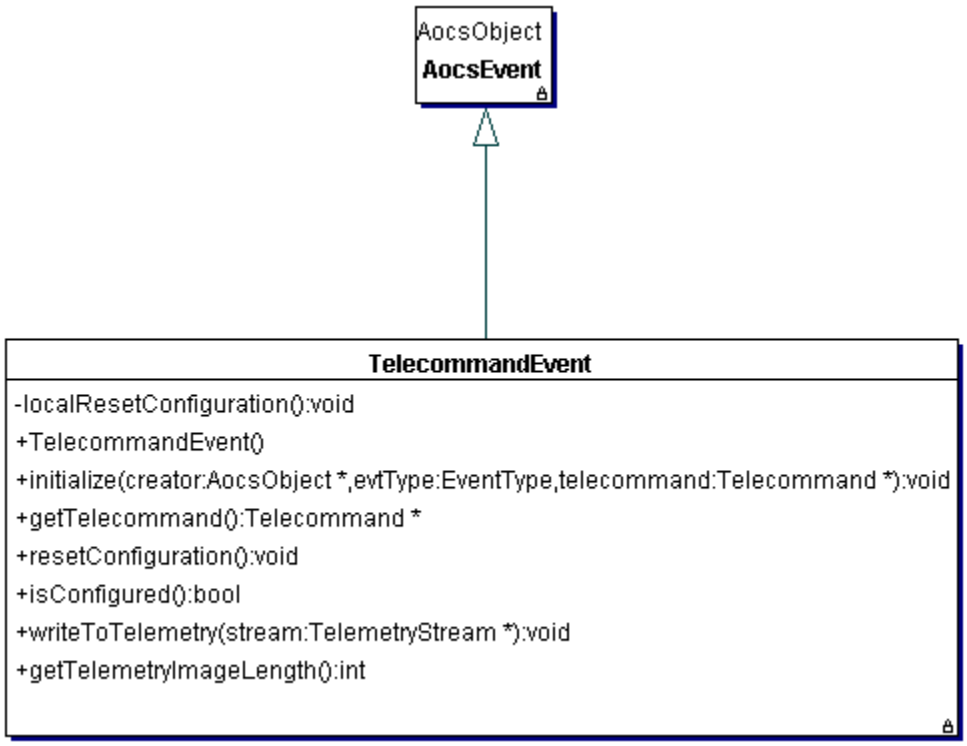
The telecommand transaction design pattern introduces the possibility of recursion since a call to method `execute` can now be recursive. The maximum depth of the recursion is given by the maximum number of telecommands that are chained into a single telecommand transaction.

7.5 Telecommand Sequences

Note that concrete transaction telecommands must provide implementations for both `execute` and `unExecute`. In some cases, unexecution is either not possible or not desired. In this case, the telecommand transaction becomes merely a *telecommand sequence*, namely a sequence of telecommands that are executed as a single telecommand but for which there is no guarantee of system integrity in case of partial or complete failure.

8 TELECOMMAND EVENTS

Telecommand-related events are recorded with the creation of events of type `TelecommandEvent`. The class diagram for this class is:



Telecommand events are generated in response to the handling of a telecommand. The `TelecommandEvent` subclass adds to its parent class one single attribute representing the reference to the telecommand.

Telecommand events are generated in response to a telecommand being loaded and in response to a telecommand being executed (see also section 5).

Note that telecommand failures are reported as telecommand events, not as failure events. This choice reflects the fact that telecommand events are generated by the telecommand manager that has no knowledge of the cause of the telecommand failure.

In some cases, the telecommand themselves may be able to create failure events. It is then up to them to record the exact cause of the failure (see also section 6.3).

For purposes of event reporting, telecommand transactions are treated as a single telecommand.

8.1 The Telemetry Interface

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

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

TM Format	TM Data
Short	instance identifier of telecommand
Normal	same as short TM
Long	same as normal TM
Debug	same as long TM

8.2 The Reset and Configurable Interface

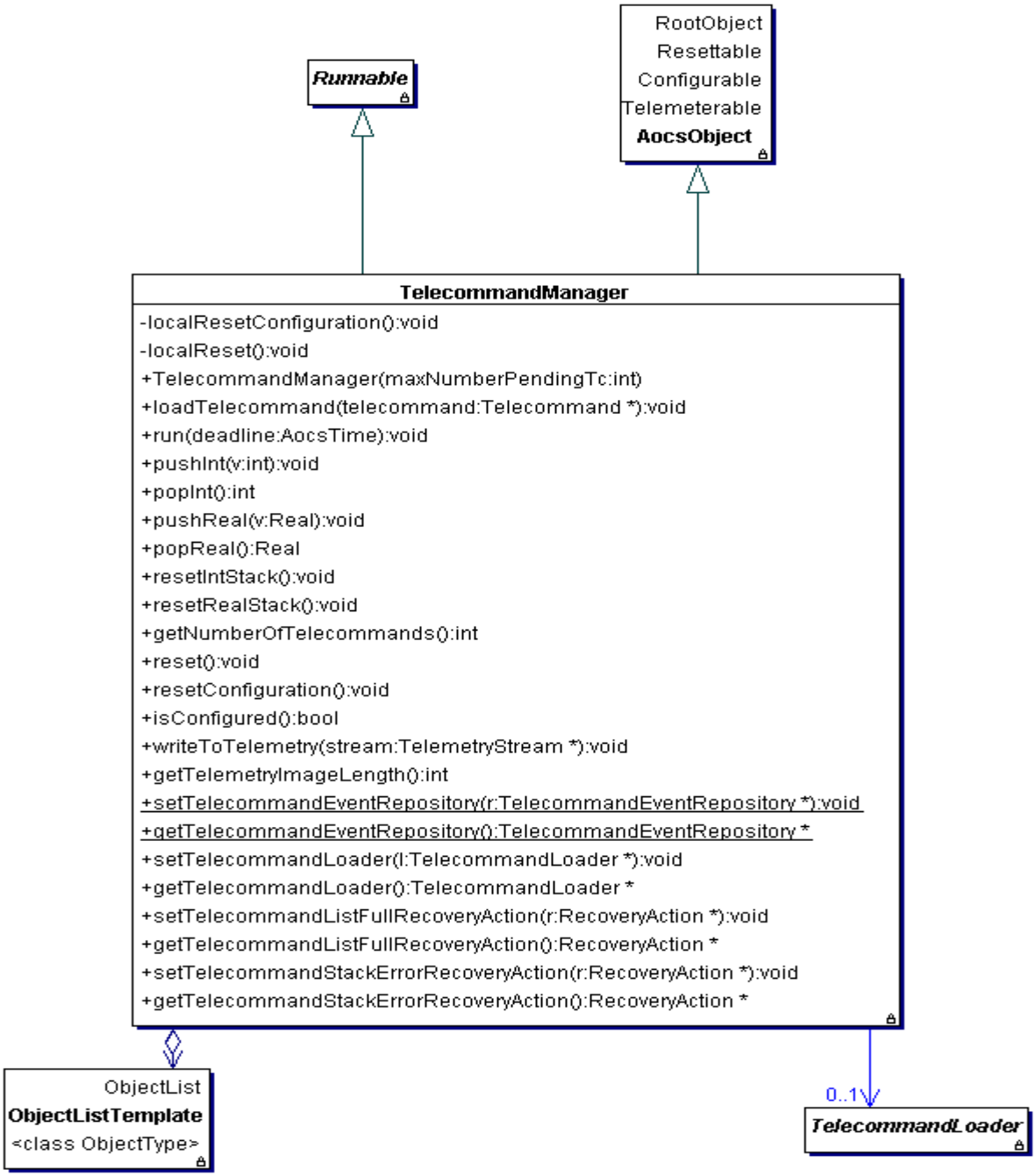
Telecommand events inherit from `AocsObject` the `Resettable` and `Configurable` interfaces and must therefore implement the corresponding method.

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

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

9 THE TELECOMMAND MANAGER

The telecommand manager is an active object that is responsible for the execution of the telecommands. Its class diagram is:



The telecommand manager is based on the command design pattern of RD1. However it can also be seen as an instance of the manager meta-pattern where telecommand execution is the pattern functionality, the telecommand manager is the functionality manager, and the `Telecommand` class decouples the functionality management from the functionality implementation.

The `TelecommandManager` class implements interface `Runnable` to signify that telecommand managers are active object.

The object list encapsulates the list of pending telecommands.

The semantics of the methods specific to this class (ie. not inherited from higher level classes) are summarized in the table:

`TelecommandManager(n)`
Constructor that sets the maximum number of telecommands that can be loaded into the telecommand manager.
`loadTelecommand()`
Method normally called by the telecommand loader to load a new telecommand into the telecommand manager. The new telecommand is inserted in the list of pending telecommand and remains there until it is executed. After execution, it is removed.
`pushInt(), popInt(),pushReal(), popReal(),resetStackInt(), resetStackReal()`
Methods to manage the internal stack used by transaction telecommands to save the system state. See section 7.2.
`getNumberOfTelecommands()`
Return the number of currently pending telecommands.
`setTelecommandEventRepository(),setTelecommandEventRepository()`
Static getter and setter methods for the telecommand event repository.
`setTelecommandLoader(),setTelecommandLoader()`
Getter and setter methods for the telecommand loader.
`setTelecommandListFullRecoveryAction(), setTelecommandListFullRecoveryAction()`

	Getter and setter methods for the recovery action associated to the failure arising when it is attempted to load a telecommand and the telecommand list is already full.
`setTelecommandStackErrorRecoveryAction(), setTelecommandStackErrorRecoveryAction()`	
	Getter and setter methods for the recovery action associated to the failure arising when a transaction telecommand performs an illegal operation on the internal stack.

9.1 Telecommand Manager Activation

The telecommand manager is activated periodically by the scheduler. When it is activated, it performs the following actions:

- It checks if there are any pending telecommands waiting to be executed
- For all pending telecommands, it checks their time tag
- For all pending telecommands that are due for execution according to their time tag, it checks whether they can execute by calling their `canExecute` method
- For all pending telecommands that are due for execution according to their time tag, it checks whether they are enabled by calling their `isEnabled` method
- It executes all telecommands that are due for execution
- It removes from the pending telecommand list the telecommands whose time tag has been passed (regardless of whether they were executed or not)
- It releases the memory allocated to the telecommand by calling the release method on the telecommand loader
- It generates telecommand events in response to telecommands being loaded or unloaded and in response to telecommands being executed (using the return value of method `execute`)

9.2 The Telemetry Interface

The telecommand manager is a telemetry objects because it (indirectly, through `AocsEvent`) inherit the `telemeterable` interface.

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

TM Format	TM Data
Short	number of pending telecommands
Normal	same as short TM
Long	normal TM + instance ID of telecommand loader
Debug	same as long TM

9.3 The Reset and Configurable Interface

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

Resetting the telecommand manager causes all pending telecommands to be removed.

Resetting the telecommand manager configuration causes the telecommand loader and the failure recovery actions to be unloaded.

Method `isConfigured` returns true if the telecommand loader and the recovery actions have been loaded and if the list of pending telecommands is configured.

10 THE TELECOMMAND LOADER

The telecommand loader is responsible for loading telecommands from the telecommand interface in the AOCS computer into the telecommand manager. The loading mechanism cannot be specified by the AOCS framework as it is application dependent. The main options are described in section 5.3.

A typical telecommand loader performs the following actions in a telecommand loading cycle:

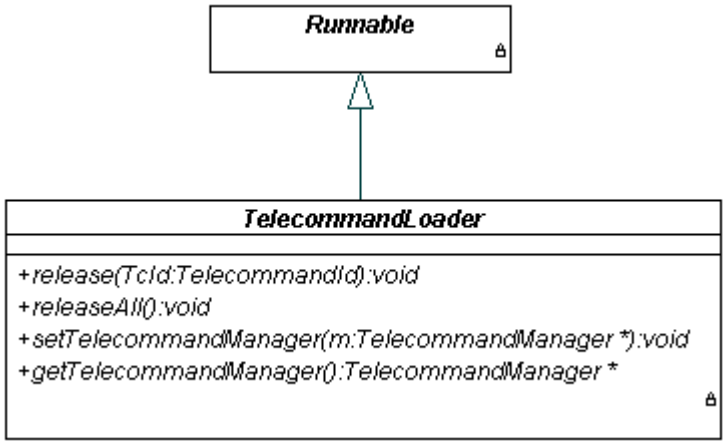
- the telecommand (its data and\or its code) is retrieved from the telecommand interface, packaged as an object of type Telecommand
- the telecommand is loaded into the telecommand manager through a call to the latter´s `loadTelecommand` method.

The telecommand loader might be triggered by an interrupt (indicating the arrival of a new telecommand) or it might be activated periodically by a scheduler. In the latter case, the telecommand loader processes all the telecommands received by the AOCS computer since the previous activation of the telecommand loader.

10.1 The `TelecommandLoader` Interface

The telecommand loader must be capable of dynamically constructing objects of type `Telecommand` to encapsulate the telecommands that are sent to the AOCS computer. Dynamic construction of objects implies management of a memory allocation process: memory must be dynamically assigned to the newly constructed telecommands. After they have been executed or after their time tag has come due, telecommands are discarded. The memory they occupy must be released.

Telecommands are discarded by the telecommand manager and it is therefore up to the telecommand manager to trigger the memory release process. For this purpose, telecommand loaders are made to implement the following interface:



The interface is derived from interface `Runnable` because telecommand loaders must be active objects. The semantics of its other methods is:

`release(TcId)`
A telecommand loader manages memory and other resources that it allocates to the telecommands that it loads. This method will cause the resources allocated to the telecommand whose identifier is `TcId` to be released. The method is normally called by the telecommand manager after the telecommand has been executed.
`releaseAll()`
This method will cause all memory allocated to telecommands to be released.
`setTelecommandManager(), getTelecommandManager`
A telecommand loader needs a reference to the telecommand manager in order to be able to add the newly-loaded telecommands to its list of pending telecommands. These are the getter and setter methods for the telecommand manager.

The telecommand manager sees the telecommand loader exclusively through the `TelecommandLoader` interface.

10.2 The DMA Telecommand Loader

In a typical telecommand loading mechanism, telecommands are placed in the AOCS memory by a dedicated DMA controller. The AOCS framework offers two default

components implementing two different DMA-based telecommand loader. `VsDmaTelecommandLoader` is designed to operate in the VisualStudio environment (where the AOCS framework was initially tested) and `Erc32DmaTelecommandLoader` is designed for the ERC32 environment with the GNU compiler (where the AOCS prototype was tested). These components were developed to allow testing of the AOCS prototype and can be used as blueprint for application-specific telecommand loaders.

The default telecommand loaders assume that incoming telecommands are available as a *telecommand buffer* consisting of an array of bytes with the format described in the header file `DmaTelecommandLoader.h`. The telecommand data are placed in the telecommand buffer by an interrupt routine triggered by the DMA device when reception of a telecommand has been completed.

A telecommand buffer contains one or more *telecommand packets*. A telecommand packet contains either a simple telecommand or a telecommand transaction.

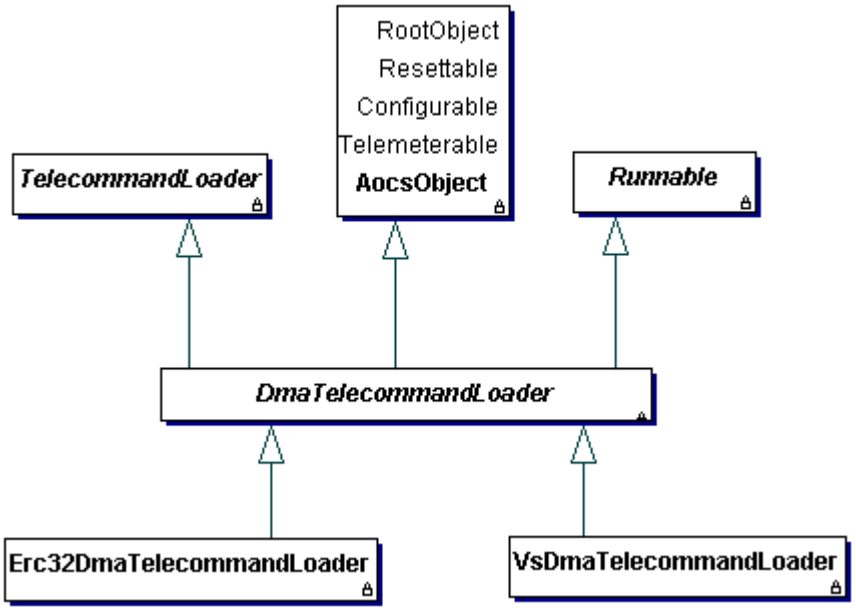
A telecommand packet is divided into one or more *telecommand blocks*. A telecommand block contains an individual telecommand, either a simple telecommand (if the packet contains a simple telecommand) or a transaction telecommand (if the packet contains a telecommand transaction).

A telecommand block contains a *data section* and may contain a *VTable section* and one or more *code sections*.

A data section contains the data part of a telecommand. The VTable section contains a virtual function table and each code section contain the code for one method of the newly loaded telecommand.

When the newly loaded telecommand is an instance of a class already present on-board, then only its data section is uplinked. When instead the telecommand represents an instance of a new class, it must carry a VTable and one or more code sections.

The class diagram for the default telecommand loaders is shown below. For detailed information on the two loaders, readers should refer to the commented source code.



10.3 Telecommand Layout in the Uplink Channel

In principle, the layout of a telecommand as it is uplinked by the ground station can be the same as the layout of the telecommand object that is being uplinked. This is simple but can be wasteful. Consider, for instance, the `enabled` field that all telecommand inherit from the base class `Telecommand`. Depending on the implementation, it can take up to four bytes. This field however carries no information since it is always initialized to true when the telecommand is first loaded onto the telecommand manager. Similarly, the instance identifier of the telecommand object could be created on board by the telecommand manager thus saving another 2 or more bytes. A third saving can be achieved with the time tag field. This field is redundant in most transaction telecommands since telecommands in a transaction all carry the same time tag.

Saving in the number of bytes to be uplinked can therefore be realized by having a telecommand loader that supplies some of the data required to construct the telecommand object. This approach was followed for both default telecommand loaders provided with the AOCS framework (see previous section).

Short telecommands can also be ensured by sensibly defining the classes of which telecommand objects may be instances. Consider for instance the telecommand to change the operational mode of a mode manager. Its base class is:

```
class ModeChangeTelecommand : public Telecommand {

     ModeManager* modeManager;
     unsigned char newMode;

public :

     ModeChangeTelecommand(AocsTime timeTag, ModeManager* m, int n);

     virtual EventType execute();
};
```

The declaration of `newMode` as `unsigned char` helps keep the telecommand image small.

11 FRAMELET HOT-SPOTS

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

11.1 Telecommand Hot-Spot

Name: Telecommand Hot-Spot
Visibility Level: framework –level
Adaptation Time: compile-time
Adaptation Method: implementation of subclasses of `Telecommand`
Pre-defined Options: default telecommands listed in section 6.4
Related Hot-Spots: none
Description Concrete telecommands are realized as instance of subclasses of class `Telecommand`. This is the hot-spot where application-dependent telecommands are built.

11.2 Transaction Telecommand Hot-Spot

Name: Transaction Telecommand Hot-Spot
Visibility Level: framework –level
Adaptation Time: compile-time
Adaptation Method: implementation of subclasses of `TransactionTelecommand`
Pre-defined Options: default transaction telecommands listed in section 7.3
Related Hot-Spots: none
Description Concrete transaction telecommands are realized as instance of subclasses of class `TransactionTelecommand`. This is the hot-spot where application-dependent transaction telecommands are built.

11.3 Recovery Action Plug-In for Illegal Stack Operations

Name: Recovery Action Plug-In for Illegal Stack Operations
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in the telemetry manager (method `setTelecommandStackErrorRecoveryAction`)
Pre-defined Options: no recovery action is defined by default
Related Hot-Spots: none
Description When a transaction telecommand attempts to perform an illegal operation on the internal stack maintained by the telecommand manager, a failure is declared. A recovery action should be associated to this failure. This hot-spot allows this recovery action to be loaded.

11.4 Recovery Action Plug-In for Too Many Telecommands

Name: Recovery Action Plug-In for Too Many Telecommands
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in the telemetry manager (method `setTelecommandListFullRecoveryAction`)
Pre-defined Options: no recovery action is defined by default
Related Hot-Spots: none
Description When the telecommand loader attempts to load a telecommand in the telecommand manager and its list of pending telecommands is already full, a failure is declared. A recovery action should be associated to this failure. This hot-spot allows this recovery action to be loaded.

11.5 Telecommand Event Repository Plug-In

Name: Telecommand Event Repository Plug-In
Visibility Level: framelet-level
Adaptation Time: run-time
Adaptation Method: plug-in component in the telecommand manager class (method `setTelecommandEventRepository`)
Pre-defined Options: `TelecommandEventRepository` component exported by inter-component communication framelet.
Related Hot-Spots: none
Description Telecommand managers log telecommand events in the telecommand event repository. This hot-spot allows this event repository component to be loaded. Note that this component is loaded as a `static` reference.

11.6 Telecommand Loader Hot-Spot

Name: Telecommand Loader Hot-Spot
Visibility Level: framework-level
Adaptation Time: compile-time
Adaptation Method: implementation of interface `TelecommandLoader`
Pre-defined Options: default telecommand loaders of section 10.2.
Related Hot-Spots: none
Description Telecommand loaders are application-specific. This hot-spot allows their definition.