EUROPEAN SOUTHERN OBSERVATORY

Organisation Européenne pour des Recherches Astronomiques dans l'Hémisphère Austral

Europäische Organisation für astronomische Forschung in der südlichen Hemisphäre

ALMA PROJECT

ATACAMA LARGE MILLIMETER ARRAY

SE Practices
Software Development Process
Methodology and Tools

Doc. No.: ALMA-PRO-ESO-xxxxx-xxxx

Issue: 1.0

Date: 2000-07-28

Prepared: G.Chiozzi, R.Karban, P.Sivera

Name Date Signature

CHANGE RECORD

ISSUE	DATE	SECTION/PAGE AFFECTED	REASON/INITIATION DOCUMENTS/REMARKS
1.0	2000-07-29		First issue

TABLE OF CONTENT

1 Introduction

1.1 Purpose

This document intends:

- to provide a methodology, based on the Unified Software Development Process(UP)[3], for the analysis, design and implementation of a project;
- to provide guidelines in the use of the Unified Modelling Language[2] to specify, construct and document the artefacts of the project;
- to identify the set of tools to be used.

The process has been applied at ESO in a pilot project for the Auxiliary TelescopesThe ATCS Online Documentation[8] is available to verify the examples (most of them are drawn from the ATCS project) and to access glossary and help information.

The UP is intended to be modified to fit a project. The development process defined here addresses these modification and extensions with respect to the ALMA project. Project management needs to place milestones on the development process.

Finally, this is not a project plan. This is a process to design the software and is part of the project plan.

1.2 Scope

This document describes the Software Development Process of the ALMA project, to be applied to Phase 1. It is intended to be reviewed in some months, before deciding on its applicability to Phase 2.

In the current implementation, several activities have to be performed by hand or using tools that could be largely improved. We apologize for that in advance, but due to the limited available resources considering that it was more important to start to use the methodology than wait to have it perfect, we decided to make it public in spite of the known limitations.

Project leaders who are in charge of defining the analysis and design of a project and SW developers who have to take care, in a second stage, of the implementation of the project compose the intended audience.

Knowledge of the basic concepts of the Unified SW Development Process (UP)[3] as well as Unified Modelling Language (UML)[1][2] is required: we don’t intend to repeat what can be easily found in the books reported in the reference section.

After the list of reference documents, list of acronyms and a short glossary, (everything is in Chapter 1), we describe the basic concepts and theories of the process (Chapter 2). This chapter explains the interpretation of the Unified Software Development Process model, putting in evidence the corner stones of the process.

Chapter 3 treats in detail the Unified Process and Use Cases concepts. Readers who are familiar with the concepts could skip this chapter.

Chapter 4 is the reference part of the document, where the work process and the tools used are described.

1.3 Acknowledgements

This work is the result of many hours of discussions, trials and tests inside the group responsible for the ESO ATCS projects (G.Chiozzi, R.Karban, P.Duhoux) and include contribution from G.Filippi and B.Gilli.

Members of the ALMA collaboration have also provided significant input, in particular members of the NRAO software group (G.Harris, F.Stauffer, M.Brooks, J.Pisano and R.Heald)

1.4 Reference Documents

The following papers, books and publications contain additional information and are referenced in the text:

[1] **The Unified Modeling Language User Guide** - G.Booch, J.Rumbaugh, I.Jacobson - 1998, Addison Wesley Pub Co; ISBN: 0201571684

[2] **The Unified Modeling Language Reference Manual** - J.Rumbaugh, I.Jacobson, G.Booch - 1998, Addison Wesley Pub Co; ISBN: 020130998X

[3] **The Unified Software Development Process** - I.Jacobson, G.Booch, J.Rumbaugh - 1998, Addison Wesley Pub Co; ISBN: 0201571692

[4] **Writing Effective Use Cases and Introducing Collaboration Cases** - L.Mattingly H.Rao - JOOP, Oct '98

[5] **Structuring Use Cases with Goals** - A.Cockburn JOOP, Sep-Oct '97 and Nov-Dec '97

[6] **Applying Use Cases : A Practical Guide** - G. Schneider, J.P.Winters, I.Jacobson - 1998, Addison-Wesley Pub Co; ISBN: 0201309815

VLT Project Software documents are available from the ESO VLT Project Archive or online at the following URL:

http://www.eso.org/projects/vlt/sw-dev/ftp/index.html

[7] **VLT-TRE-15151-1917** - Technical Report on Analysis and Design with UML for the Auxiliary Telescope Control System

[8] ATCS Online Documentation
It is available at the following URL:
http://www.eso.org/~gchiozzi/ATS/atcsdoc
UID: atcsdoc PWD: ______________ (ask G.Chiozzi for the PWD and write it here)

Other readings on the subjects like the ATCS Technical Report[7] can be found here:

[9] Suggested readings from the ESO VLT Object Oriented Working Group:
http://www.eso.org/projects/vlt/sw-dev/oowg-forum/readings.html

[10] Making a List and Checking It Twice - J.D.McGregor - JOOP June 2000

1.5 Acronyms

All the acronyms used in the ATCS project are available in the abbreviations' section that is part of the online documentation [8].

Here we just provide some basic acronyms used in this document.

ALMA	Atacama Large Millimeter Array
AT	Auxiliary Telescope
ATCS	Auxiliary Telescope Control Software
ATS	Auxiliary Telescope System
ESO	European Southern Observatory
HTML	HyperText Markup Language
HW	Hardware
ICD	Interface Control Document
NRAO	National Radio Astronomy Observatory
PWD	PassWorD
SE	Software Engineering
SW	Software
UID	User IDentification
UML	Unified Modelling Language
UP	Unified (Software Development) Process
URL	Universal Resource Locator
VLT	Very Large Telescope
WWW	World Wide Web

1.6 Glossary

A complete glossary of all the terms used in the UML language and in the Unified Software development process is given in the two reference books [1][2] and [3].

All the definitions used in the ATCS project are available in the glossary that is part of the online documentation [8].

Here we just provide some basic definitions, extracted from these referenced documents.

Actor
An Actor is a role of an entity external to the system. Actors can be humans, machines, or devices.
"One physical object can play several roles and therefore be modeled by several Actors".
A primary Actor is one having a goal requiring the assistance of the system.
A secondary Actor is one from which the system needs assistance to satisfy its goal.

Phase
The span of time between two major milestones of a development process

Stakeholders
Stakeholders are the funding authorities. They typically include users, salespeople, project managers, line managers, production people, regulatory agencies, and so on.

Use Case
The specification of sequences of actions, including variants sequences and error sequences, that a system, subsystem or class performs that yields an observable result of value to a particular Actor.

Workflow

A realization of (a part of) a business Use Case. Can be described in terms of activity diagrams that include participating workers, the activities they perform, and the artifacts they produce.

2 Overview

2.1 Methodology corner stones - "One Document" objective

The Software Development Process is built on the following corner stones (most of them are taken up in detail in chapter 3):

- **The Unified SW Development Process[3].**
  The Unified SW Development Process is nowadays well established and is the result of the joint work of three among the most recognized experts in the field. It is the starting point for the described methodology, in the sense that it has been tested, tried out and finally adapted to the specific context of the project.

- **The Use Cases (see section 3.9).**
  Use Cases are a fundamental part of the process, since they allow collecting requirements in a clear and at the same time light way. The requirements are expressed at a sufficient level of formality that makes them traceable throughout the whole process up to the final acceptance testing. They build the common language of SW engineers and system customers.
  Since it is very important to have a common starting point and a common language, for communication between different groups or even different people in the same group, **a common Use Case template has been compiled**. The template is based on the excellent papers of L.Mattingly[4] and A.Cockburn[5] and on the book entirely dedicated to Use Cases of G.Schneider[6].
  The template is available in Appendix A, Chapter 5. It's recommended keeping the template at hand whenever you want to write a use case.
  The defined Use Case model allows two levels of access to the Use Cases introducing a ***design view*** identified by the ↳ symbol. This view contains details that are pertinent only to design and can be skipped to obtain the ***requirements view***, that contains information relevant at requirements specification level. The requirements view represents the contract with the customers of the system, whereas the design view adds - transparently for the customer -information only relevant for the architecture and design of the system.
  It is possible at any time to print all Use Cases containing only Requirements information, (an automatic script is provided to strip the design information from a file) marking what is changed with respect to the previous version of the document: this makes it easy to identify the changes in requirements that have to be discussed with the stakeholders.
  At the same time, team members editing Use Cases can easily see which modifications are just an elaboration and new details and which actually have an impact on requirements.
  Summarizing:
  - One single Use Case source, two views
  - Requirements information is the core
  - Design information is added and marked as such using the ↳ symbol[1].

- **The Unified Modeling Language (UML)[1][2].**
  The need for a uniform and consistent visual language to express the results of analysis, design and implementation is very strong. The UML is now a well-established standard, widely used and the

[1] Some html instructions have been defined to create the array symbol. These instructions are supplied with the Use Case Template (see Appendix A). Whenever you want to mark a passage in the text as "design view" you can cut&paste these instructions.

selected diagramming language for the process.

- **The WWW: online, hypertext documentation.**
  The WWW is the media that provides the features that allows to meet the "One Document" Objective, i.e. to arrive at the point where:
  - All the relevant documentation of the project is kept in one single document
  - No information has to be duplicated, but it can be simply cross-referenced wherever needed.
  - The latest version of the whole documentation is always available for immediate access
  - Changes can be easily done, but everything is, at the same time, kept under configuration control
  - Requirement and design documents and the actual implementation code are seamlessly integrated. It is possible to navigate from one to the other and back.

  However, online documentation alone is not enough.
  Printed documents are essential not only for reviewers, but also for members of the development team.
  As a general rule, everything has to be done in HTML format and the starting point is always the HTML documentation. Here, the information is kept as atomic as possible. The elementary information units can then be used as building blocks and inserted and reused in very different documents.
  Here are some guidelines:
  - Each Use Case is kept as a separate HTML file and contains hyperlinks to other Use Cases, help pages or other documents and files.
  - All HTML files are inserted in the printable documents, gluing them with the proper descriptive text and fitting them in the printable document architecture.
  - Whenever a Use Case is modified in the online documentation, we supply some automatic procedures that take care of updating the printable documents, often processing the file, for example to strip design information when it has to be used in requirements document (see chapter 4).

  In this way, a document aligned with the current development can be printed.
  The configuration control assures that changes to the whole documentation are kept under control and it is always possible to identify what has changed and, if necessary, to rollback.

- **Standard milestones and deliverables.**

  It is necessary to stick to standard project milestones that fit in current working environments and are widely accepted. The following table shows the milestones associated with the process phases:

Phase	Project Milestone	Unified Process Milestone
Inception (first)	Software Requirements Review	-----
Inception	Preliminary Design Review	Life Cycle Objectives
Elaboration	Final Design Review	Life Cycle Architecture
Construction	System Delivery	Initial Operation Capability
Transition	Final Acceptance Test	Product Release

The second column shows the milestones defined at ESO level for a project. These milestones are based on IEEE standards, adopted for the former VLT Software Life Cycle.
The third column contains the milestones defined in the Unified Software Development Process[3].
Comparing the scope and meaning of the milestones in the IEEE definition and in the Unified Process it can be concluded that they map very well.
Since the IEEE names are used by other technical groups (e.g. mechanics and electronics) within the organization and are widely used with external contractors these names are kept.
At each milestone, printed documents are released, reviewed and archived for reference.

- **Standard procedures.**

  The work is organized in a software module kept under configuration control to guarantee information consistency. A module template that provides the right directory structure and a certain number of files to be edited is available. Some procedures to help the users in maintaining the software (documents and Rose model) are supplied. In Chapter 4 you can find a detailed description of the work steps.

2.2 Software for science versus commercial application software

Since our environment deals with the development of control software for scientific experiments, this type of project is very different from other software projects and in particular from commercial application software.

Most of the books available on the market on software development focus on commercial applications, since they cover a larger fraction of the software development market. The concepts described in those books have to be accurately evaluated considering the specifics of our type of project. Few interesting books deal specifically with the development of Real Time Control Systems and are listed in [9].

Here are some important aspects to keep in mind:

- A scientific experiment is always at the edge of knowledge and technology. As a consequence, requirements are often unclear and change during the course of the project.
- The software to be developed is highly linked to the hardware and to the electronics. This has many implications. For example software deadlines actually depend and are affected by hardware delivery dates. When the hardware is ready, software must also be ready. Planning is always done a priori based on the hardware components.
- The system is unique. There will be just one or very few installations. We cannot count on hundreds or thousands of beta testers to debug the software.
- When the system is eventually integrated, the control software is used also to validate hardware and electronic performances. The software team responsible for the final integration becomes automatically also responsible for the verification of the whole system. Sometimes it is very difficult to see if a problem lies in the software or originates from the hardware or electronics.
- Psychologically, the stakeholders of the system are naturally driven to think that hardware is fixed, while software can be easily modified. For this reason, hardware requirements are typically analyzed in great details, while software requirements are left unclear or modified freely.
- Interfaces with the hardware and the electronics have a very big impact on the system and have a big effort has to be spent in making them clear and stable.

Nevertheless, this manual can be applied to other types of project (not necessarily a Control System project!) adapting it to the different requirements another type of project could have.

2.3 Based on the experience

A lot of time has been dedicated in tuning the process and adapting it to our specific environment. The results are very satisfactory and there has been very good feedback from people involved in other aspects of the project.

There has been also a very good feedback from the stakeholders for what concerns the usage of the Use Cases.

In particular:

- Use Cases are a very good way of capturing system requirements
- The process provides a good support for tracing requirements through all phases of the project
- With the support of good tools, it is very well suited for team work and collaborative development
- The use of the World Wide Web as a documentation repository is very effective, but work is required to provide also good printable documentation automatically extracted from the Web documentation.

Open discussion and co-operation among team members have proven to be essential to build and keep a "common vision". Tools for collaborative development help a lot in this respect. Tutoring is also very important to train new team members or to introduce the process in a new project.

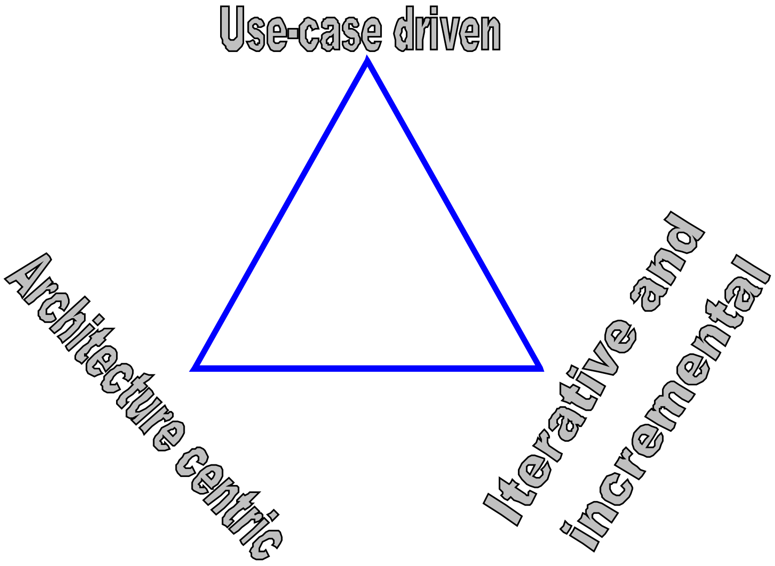
The next Chapter explains in greater depth the Unified Development Process, giving a detailed description of each project milestone.

3 Software Process Overview

A software development process describes all the activities that are necessary to transform user requirements into a final software system. In particular it is used to organize, track, document and test the project.

3.1 Process key aspects

The essential aspects of this process are captured, as the authors say, by the following three key-words:



3.1.1 Use Case driven process

Use Cases describe the external interactions and elaborate and identify the system functionality. The set of all Use Cases makes up the Use Case model.

The main purpose of the Use Case model is to give an answer to the following question:

What is the system supposed to do ***for each user****?*

With respect to the traditional functional specification, this question focuses on the value each Use Case has for a specific user. This is much more than just providing a list of functions that might be good to have.

The whole development process follows a flow that origins from the Use Cases. At the same time, Use Cases mature and evolve during the life cycle of the project.

3.1.2 Architecture Centric Process

The architecture of a software system identifies the most important static and dynamic aspects of the system and provides a common vision that all developers and customers share.

The framework of the architecture is based on the understanding of the key functions that the system must be able to fulfil. These key functions are typically captured by 5% to 10% of all Use Cases. This estimates gives an idea of the amount of Use Cases that have to be analyzed to identify the base architecture of the system.

Through different views of the system being built, the architecture shows how the system will allow the realization of all specified Use Cases. During the project life cycle, the design will add to the architecture all details necessary for the actual implementation of the system.

The architecture will remain as the view of the high-level design stressing the important aspects while leaving aside the details.

The architecture is the starting point to identify the *layout* of the system and a strong initial effort is essential to get a final system that will work. It is also true, on the other hand, that the architecture will evolve during the project life cycle as the Use Cases are elaborated in detail and mature.

All object-oriented literature emphasizes strongly the parallel between software projects and the process of building houses. As in any comparison, there are some limitations to take into account, but in general it is very helpful in clarifying the concepts expressed by object-oriented analysis and design processes. All reference documents given in section 1.4 make extensive usage of this example.

3.1.3 Iterative and incremental process

The work necessary to build the final system is divided into smaller slices.

Finding the requirements, developing the use cases and selecting the most important use cases to be developed first starts the process. This is the first increment. One or more iterations occur for each phase and each phase uses the different design tasks.

The driving concept consists of the idea that every slice must be a mini-project in itself, which has well defined goals and which can be measured and controlled.

Each mini-project is an **iteration** in the development, since it performs a complete workflow.

The purpose of each iteration is the realization of a set of Use Cases. These Use Cases are first identified and specified. Then a design is created according to the chosen architecture for the whole system. The development team implements the design and verifies that the system satisfies the Use Cases.

If the iteration meets its goals, the system is released and the development proceeds with the next iteration.

The **iteration** is also an **increment** since it provides a new set of functionality in the system.

The Use Cases to be implemented at each iteration are selected according to two criteria:

- The selected Use Cases must all together extend the functionality of the system developed so far
- Each iteration must deal with the most important risk areas. Use Cases associated with higher risks have higher priority.

3.2 Project life cycle

The Unified Process repeats over a series of ***cycles*** making up the life of the project.

Each cycle concludes with a ***major product release*** (like Microsoft Word, as a well known example from the office world).

Each *cycle* consist of four ***phases***, where Use Case play a key role***:***

- ***Inception -*** This starts the definition task. Requirements are produced and analysis is started. High level Use Cases are developed to identify what is in the scope of the project.
- ***Elaboration*** - Requirements are refined, analysis completed, and the system designed. The system design is detailed enough to give the programmers the modules to design and code. More detailed Use Cases contribute to the baseline architecture and to the risk analysis. They are also used to define the planning for the *Construction Phase*.
- ***Construction*** - Programmers implement modules identified in the baseline design. Detailed design is performed, software is coded, and unit tests are designed and performed. Integration and system test plans should be created. Use Cases will be used as the starting point for detailed design and for developing test plans. Use Cases provide the core of the requirements that have to be satisfied at each *iteration*.
- ***Transition*** - System testing, acceptance testing, user documentation, software installation are performed. Use Cases are the core items of the Acceptance Test. They are also used to develop user guides and for training.

Each *phase* terminates in a ***milestone***, which is tagged by the official availability of a set of artifacts (documents, diagrams, source code, executable programs and so on).

Each *phase* is subdivided into ***iterations***. A *iteration* terminates in a ***build***, i.e. an internal release of artifacts that is used as a check point to verify that the objectives planned for the *iteration* have been met.

During each *phase/iteration*, five ***core workflows*** take place:

- ***Requirements capture***
- ***Analysis***
- ***Design***
- ***Implementation***
- ***Test***

Figure 1, extracted from [3], shows that the emphasis on each *core workflow* changes throughout the various *phases*. Note that all activities can be occurring during all iterations of the development. For example, prototyping of a high-risk area may take place during inception. This is not waterfall - it's merely that the emphasis varies between iterations.

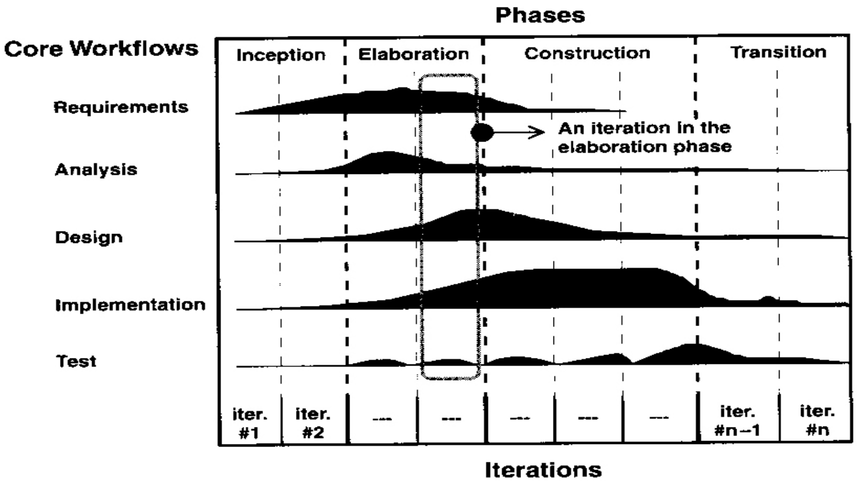


Figure 1 - Project phases and workflows

See [3] for more details on the project life cycle defined by the Unified Software Development Process.

3.3 Project milestones

The official project milestones are extremely important, since they make available to managers and stakeholders a set of artifacts to be reviewed.

They have in this way the possibility of making crucial decisions before work can proceed to the next phase and to monitor the progress of the work.

These milestones are fixed points in the planning and their existence helps the development team in focusing the work toward specific dates.

For this reason it is essential that milestones be not postponed in case of delays in the project, rather the scope of the milestone is clearly and officially restricted. This identifies and manages better the project problems and allows deciding and undertaking corrective actions before the project runs out of control.

Since the first phases of the project extremely important, an additional milestone has been inserted in the middle of the inception phase of the first project cycle. This defines the following milestones:

Deliverable milestones

1. Software requirements review (software requirements)
2. Preliminary Design Review (software requirements and base architecture)
3. Final Design Review (Design concept and architecture)
4. System delivery
5. Final acceptance test

These are deliverables. Internal and project management milestones need to be added.

They will be described in the following sections, in terms of the deliverables and of the workflow that drives to the milestone.

3.4 First Iteration of Inception: from Kickoff meeting to Software Requirements Review

Purpose:

- Produce software requirements and system behaviour using use cases, understand basic requirements
- Reach agreement on basic requirements between development team and stakeholders

Steps:

- Draft scope of the system
- Describe system context
- Identify the actors, functional and non-functional requirements, risks
- Formal Review

Artefacts:

- Glossary and overall system description
- System Context Diagram
- Actors
- Use Case Model (functional software requirements)
- General and non-functional requirements
- Risk assessment
- Assumptions

Documentation kit:

Online documentation

Software - Requirements Specification, issue 1.x

Draft project plan

Purpose

The purpose of the first Iteration in the Inception Phase is to understand the basic requirements of the system to be developed.

At the end of this first iteration the Software Requirements Review has the purpose to reach an initial agreement between the stakeholders and the development team on the requirements for the system and to demonstrate to the stakeholders that the development team has a good understanding of the agreed requirements.

This milestone is not part of the Unified Software Development Process, but it is essential for a good start of a project, since it provides the opportunity for an official clarification on the basic characteristics of the system using the formal language of the Use Cases.

Stakeholders have an early possibility to evaluate how the development team has interpreted the information provided (Statement of work, high level requirements, feature lists, etc.) and can provide valuable feedback.

In every project there are two iterations in the Inception Phase:

- A short iteration terminated with a Software Requirements Review
- A longer iteration terminated with the Preliminary Design Review

Artifacts

During the first iteration of the Inception Phase, the following items are developed and delivered for the Software Requirements Review:

- **Glossary and overall system description**
  A general description of the system to be developed and a glossary with the definition of the terms used are essential to create a common background between all stakeholders and team members and to avoid misunderstanding. If the domain of the project is not well known domain analysis should be done and a domain model produced.

- **System Context Diagram**
  The context diagram shows the system under design as a black box and all the Actors that interact with the system.

- **Actors**
  A description of all the entities interacting with the system. Primary Actors are the users of the system (not necessarily human, also other software systems). Secondary Actors are all the sub-systems that are part of the system and that have to be controlled by software or that are needed to fulfill the requirements (like a Time Reference System, that is necessary to satisfy time precision requirements).

- **Use Case Model**
  All the Use Cases are derived from the user requirements and from higher level documents. The development team is responsible for writing the Use Cases based on whatever information is initially provided by the stakeholders. Writing the Use Cases requires technical knowledge and a good insight on the software development process.
  It is not correct to ask the stakeholders to write them. When Use Cases have been inferred from the available initial documentation they are discussed with the stakeholders and accordingly modified. The language used to write the Use Cases is simple enough for the stakeholders to understand it, but the process to obtain them requires technical knowledge that only the development team can have.

- **General and non-functional requirements**
  Use Cases capture functional requirements, but a project is always bounded also by general and non-functional requirements. These requirements specify the adoption of specific standards, hardware architectures, software libraries or system performances, maintainability, extensibility and reliability. Some of these requirements fit in the bodies of the Use Cases, but for most of them specific document sections have to be written.
  Imposed by project standards and by contour constraints.

- **Risk assessment**
  A first basic analysis of the areas of major risks in the project.

3.5 Inception leads to Preliminary Design Review

Purpose:

- Define system scope
- Identify an architecture that can implement the requirements expressed for the system
- Identify and mitigate the risks critical to the successful implementation of the system

Steps:

- Elaborate, extend and complement Use Cases of the previous phase
- Detail critical Use Cases (create scenarios)
- Draw activity and conceptual diagrams for complicated Use Cases
- Identify sub-systems/packages
- Identify hardware interfaces
- Generate package diagram with interfaces in the packages
- Place high-level class diagrams in packages
- Step through use cases to verify package diagram satisfies use cases
- Elaborate packages
- Examine UML diagrams with checklists
- Formal Review

Artefacts:

- Update of all deliverables issued for the Software Requirements Review
- Packages, package diagram
- Interfaces with Actors
- Deployment Diagram and Process View
- Architecture diagram
- High-level class diagrams
- Performance Analysis
- Design of critical Use Cases, Use Case model
- Interface control documents
- Simple state diagrams for complex use cases
- Planning

Documentation kit:

Online documentation

Software - Requirements Specification, Issue 2.x

System Design Description, Issue 1.x

ICDs with electro-mechanical devices

Project plan

Purpose

In the Inception Phase, the development team has to:

- Define system scope, i.e. it has to identify what is inside and what is outside the system. The basic interfaces between the system and the Actors are sketched and they fit with what provided or foreseen for the Actors.
- Identify an architecture that can implement the requirements expressed for the system. The high level internal structure of the system is defined in a realistic way.
- Identify and mitigate the risks critical to the successful implementation of the system. Risks can come from many technical and non-technical areas.

At the end of the inception phase, the Preliminary Design Review has to demonstrate that all these objectives have been reached and that it is feasible for the team to proceed with the project, i.e. the team has the technical capabilities to implement the proposed architecture.

A successful Preliminary Design Review also demonstrates that the stakeholders agree on the requirements identified and on the objectives stated. They give their agreement to proceed with the next phase.

Artifacts

During the whole Inception Phase, the team works on the items already delivered for the Software Requirements Review (and in particular on the Use Case Model) and adds new details. It also works on a set of new items that analyze the system both as a black box and as an open box in order to identify the architectural baseline.

- **Packages**

  The first step to draw the architecture of the system is to subdivide it into smaller and more manageable units, called *packages* in the UML terminology[2].

  Packages should have[1]:

  - **A single functionality**
  - **Strong internal cohesion**
  - **Loose external coupling**
  - **Minimal communication to other packages**

  Packages can also be recursively nested inside other packages.

  The Packages Diagram shows the main interrelations between them. Figure 2 shows the complete ATCS Packages Diagram as an example.

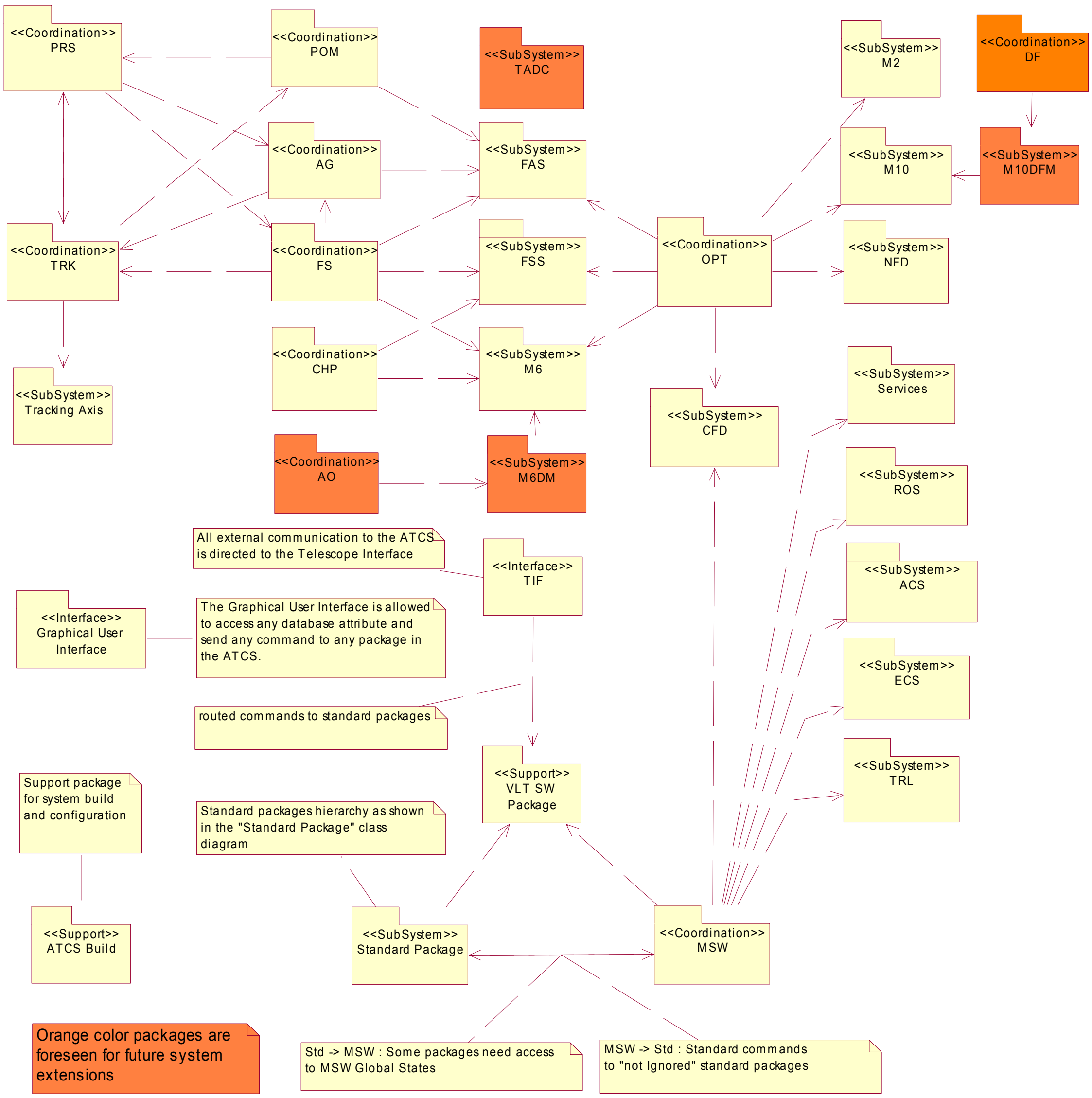

<<Coordination>> PRS
<<Coordination>> POM
<<SubSystem>> TADC
<<SubSystem>> M2
<<Coordination>> DF
<<Coordination>> AG
<<SubSystem>> FAS
<<SubSystem>> M10
<<SubSystem>> M10DFM
<<Coordination>> TRK
<<Coordination>> FS
<<SubSystem>> FSS
<<Coordination>> OPT
<<SubSystem>> NFD
<<Coordination>> CHP
<<SubSystem>> M6
<<SubSystem>> Tracking Axis
<<SubSystem>> CFD
<<SubSystem>> Services
<<Coordination>> AO
<<SubSystem>> M6DM
<<SubSystem>> ROS
All external communication to the ATCS is directed to the Telescope Interface
<<Interface>> TIF
<<SubSystem>> ACS
<<Interface>> Graphical User Interface
The Graphical User Interface is allowed to access any database attribute and send any command to any package in the ATCS.
<<SubSystem>> ECS
routed commands to standard packages
<<Support>> VLT SW Package
<<SubSystem>> TRL
Support package for system build and configuration
Standard packages hierarchy as shown in the "Standard Package" class diagram
<<Support>> ATCS Build
<<SubSystem>> Standard Package
<<Coordination>> MSW
Orange color packages are foreseen for future system extensions
Std -> MSW : Some packages need access to MSW Global States
MSW -> Std : Standard commands to "not Ignored" standard packages

Figure 2- ATCS Packages Diagram

First identify the packages and distribute them on the processing nodes, than go in design details package by package to identify the classes.

Note that alternative approaches are described in literature.

The above approach maps best our preconditions, since:

- Our system is distributed on an already defined HW architecture and set of processing nodes.
- We start from an existing and already known system, with HW sub-systems already clearly identified.

Identification of packages

Packages are identified starting from an analysis of the Use Cases and (in particular for Control Systems) of the defined HW architecture of the system.

The process consists of iterating the following four steps until the architecture is stable (see Figure 3):

1. **Identify candidate packages putting them in one of the following 4 categories:**
   - **Subsystem packages**
     There is a subsystem package per every physical device (or group of strictly related devices) to be controlled by our system. The package is fully responsible for the control of the device and implements all the interfaces that are necessary to interact with the electro-mechanical units that are part of the device itself.
   - **Coordination packages**
     Are hierarchically above subsystem packages. They are responsible for actions of a combined coordination nature, and they often use one or more subsystem to execute their actions (for example "telescope presetting" or "tracking")
   - **Interface packages**
     Implement the public interfaces that the system provides to primary Actors. These include graphical user interfaces, but also programmatic interfaces and any other type of command interface.
   - **Support packages.**
     Packages that cannot be well classified in any of the three previous categories are just simply defined as support packages.
2. **Assign Use Cases to packages**

   Take the Use Cases one by one and assign each of them to the package that seems to be the best candidate to handle the responsibility for that Use Case. Add new packages if none seems to be responsible for the Use Case.
3. **Take each Use Case and step through the courses**
   - Each Use Case must be entirely executed inside the package responsible for it
   - If a step (or group of steps) has to be performed by another package, it becomes a Use Case for that package
     The package becomes a secondary Actor in our Use Case.
     The steps are replaced with an interaction with that Actor, invoking the new Use Case just identified.

4. **Write Package Documentation consisting of:**
   - **Package Description**
     A textual description of the package, following a predefined template
   - **Package Class Diagram**
     A first Class Diagram where every package is just represented by a class. It allows representing the basic relations between the packages
   - **Package Use Case Diagram**
     It shows all Use Cases that are responsibility of the package and the relations with Actors.

See chapter 8 for a template package documentation. In chapter 9 you find an example package description extracted from the ATCS Online Documentation[8].

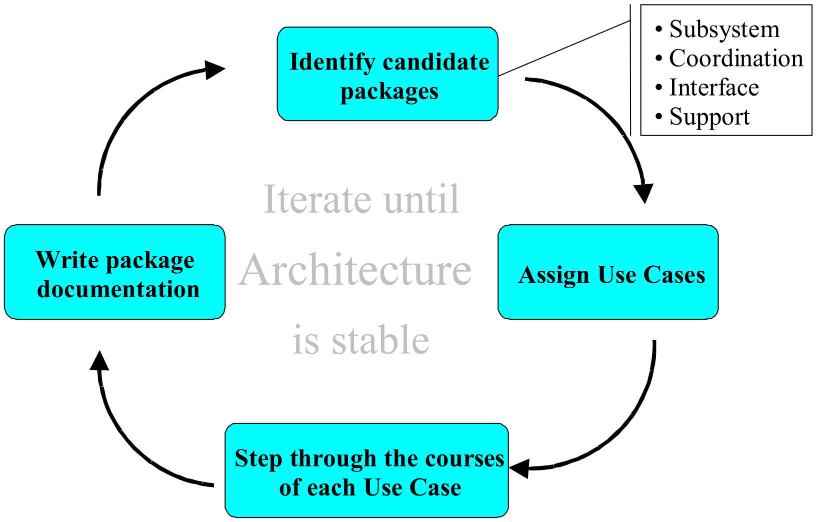

Identify candidate packages
• Subsystem
• Coordination
• Interface
• Support
Assign Use Cases
Step through the courses of each Use Case
Write package documentation
Iterate until Architecture is stable

Figure 3-Identify Packages

Interfaces

This is the most important item for the external view of the system.

- Create an ICD (Interface Control Document) section in the Online Documentation. There must be one ICD sub-section per Actor.
- Every ICD is subdivided in Interfaces, where every Interface describes a small number of Operations that are highly internally coherent and loosely coupled with other interfaces.
- Every time a step in a Use Case involves an interaction of the system with an Actor, an Hyperlink to the corresponding **Actor:Interface:Operation** is added.
- The ICD sections are used to extract the ICD documents for not already implemented/existing Actors and to check the already implemented interfaces with the existing Actors.

This process is essential for a correct cross-referencing and checking of interfaces. Clearly, the level of definition of the Operations depends on the stage of analysis, design and implementation.

It has to be ensured that all the needed Interfaces are specified. As soon as more detailed information are available the ICDs are updated.

An example of ICD is given in APPENDIX C in chapter 7.

- **Deployment Diagram and Process view**

  The Deployment Diagram is a Structural Diagram. It shows a set of nodes and their relationships. This diagram is used to show the static deployment view of the architecture, i.e. the allocation of processes, as identified in the packages, to the processing nodes in the physical design of the system. It is essential to be able to perform a performance analysis.

- **Performance analysis**

  It is essential for PDR to demonstrate that a reasonable estimation of the performances required is available and that they can be met with the available HW. This has been done through a table with processes/packages allocated to HW:

  - CPU power/budget
  - Estimated CPU consumption of SW, based on experience or previous projects
  - Percentage of coverage from existing(reusable) SW
  - Evaluation of the cost of implementation

- **Design of critical Use Cases:**

  For the most important and/or complex Use Cases it is necessary to provide more details, and in particular it is necessary to demonstrate how the proposed system design is going to allow the implementation of the Use Cases.

  For this purpose some behavioral diagrams can be used[1][3]:

  - **Activity diagrams**

    An activity diagram shows the flow from activity to activity within a system (and in particular for what concerns a specific Use Case). The diagram is especially important in modeling the function of a system and emphasizes the flow of control among sub-systems or, going to higher design details, objects.

  - **Interaction (Collaboration and Sequence) Diagrams**

    These diagrams are essential to model the dynamic aspects of a system. An interaction diagram shows an interaction, consisting of a set of objects (at our design level, sub-systems) and their relationships, including the messages that may be dispatched among them.

    The Sequence Diagram emphasizes the time ordering of messages.

    The Collaboration Diagram emphasizes the structural organization of the objects that send and receive messages.

  - **State Diagrams**

    State Diagrams show a state machine, consisting of states, transitions, events and activities. They are particularly important in modeling the behavior of an interface, class and collaboration. They emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

For a preliminary design it is not necessary to go in the detail of state machine internal to every single package. It can be very useful to provide a high level system state diagram and some state diagrams for the sub-systems involved in external interfaces, where they can help in understanding the dynamic behavior of important Use Cases.

- **Planning**

  For the PDR documentation kit a planning has to be provided, showing:

  1. Planned dates for Milestones. These include the major milestones for Final Design Review, System Delivery and Final Acceptance Test, possibly repeated for the number of incremental releases foreseen for the product.
  2. Description of the purpose of each milestone
  3. Deliverables to be releases with each milestone (documentation, tests, software)
  4. Required items (type, quantity and dates) necessary as external preconditions to meet the milestone.

  An essential part of the planning activity consists also in assigning a priority level to all Use Cases or eventually to Use Cases sub-flows, i.e. to scenarios. The content of a release and of iteration in the Construction Phase is actually determined by identifying which Use Cases (or scenarios) will have to be implemented, based on their priority.

  This will be formalized in the planning delivered for FDR, that has to contain a mapping of all Use Cases associated to any release, to allow formal tracing.

  During this phase is also necessary to develop a first version of the detailed planning, that is not necessarily part of the PDR documentation kit given for review to the stakeholder but that is anyway essential for the FDR.

The System Design Description document contains also ALL information that is part of the Requirements Specification. Readers interested only in requirements should get this last document. Readers interested also in more details and in the system architecture need to get only the System Design Description. The common information is in both cases imported from the ATCS Online Documentation[8] to warranty consistency.

3.6 Elaboration leads to Final Design Review

Purpose:

- Identify a robust and resilient architecture
- Identify and mitigate major risks
- Support with a proper project plan a realist estimate of schedule, cost and quality.
- Elaborate packages, Uses Cases, ICDs, class-diagrams, activity and interaction diagrams
- Complete as much as possible use cases, packages, interfaces and classes

Steps:

- Prototype high risk areas
- Identify common patterns and frameworks and detailed design for them
- Build more detailed class hierarchies for packages
- Create more detailed interaction diagrams for packages
- Create more detailed state diagrams for classes
- Detailed use cases
- Detailed interfaces
- Detailed package descriptions
- Examine UML diagrams with checklists
- Formal Review

Artefacts:

- Updated and detailed deliverables issued for the Software Requirements Review and System design description
- Class diagrams, interaction and activity diagrams for all packages
- Project plan
- Executable architecture
- Subsystem descriptions
- Package descriptions

Documentation kit:

Software - System Design Description 2.x

ICDs

Detailed design of complex packages

Detailed design of frameworks or patterns

Purpose

The purpose of the Elaboration Phase is:

- Identify a robust and resilient architecture baseline
- Identify and mitigate major risks
- Support with a proper project plan a realist estimate of schedule, cost and quality.

The Final Design Review must demonstrate that these objectives have been reached and the stakeholders officially accept the proposed architecture. On the other side, the organization responsible for the development finally commit itself in delivering the product with the agreed features and within the agreed budget and planning forecasts.

This milestone is the no-return point in the project and all major risks must have been investigated.

Artifacts

During the whole Elaboration Phase the team works on the items already delivered for the Preliminary Design Review, adding new details.

It is very important in this phase to develop prototypes to verify the proposed architecture and to analyze the major risks. The prototypes also help in estimating the time and resources necessary for the implementation, in particular when no historical data coming from previous projects are available.

Building prototypes as complete as possible in terms of control electronics and including a number of hardware components is considered very critical. These prototypes are used in the elaboration phase to assess the proposed architecture and verify that critical requirements can be satisfied.

During this phase, most of the time is used in the Analysis and Design work-flows[3] to build the architecture of the packages (see Figure 4, where there is an example drawn from the ATCS Project).

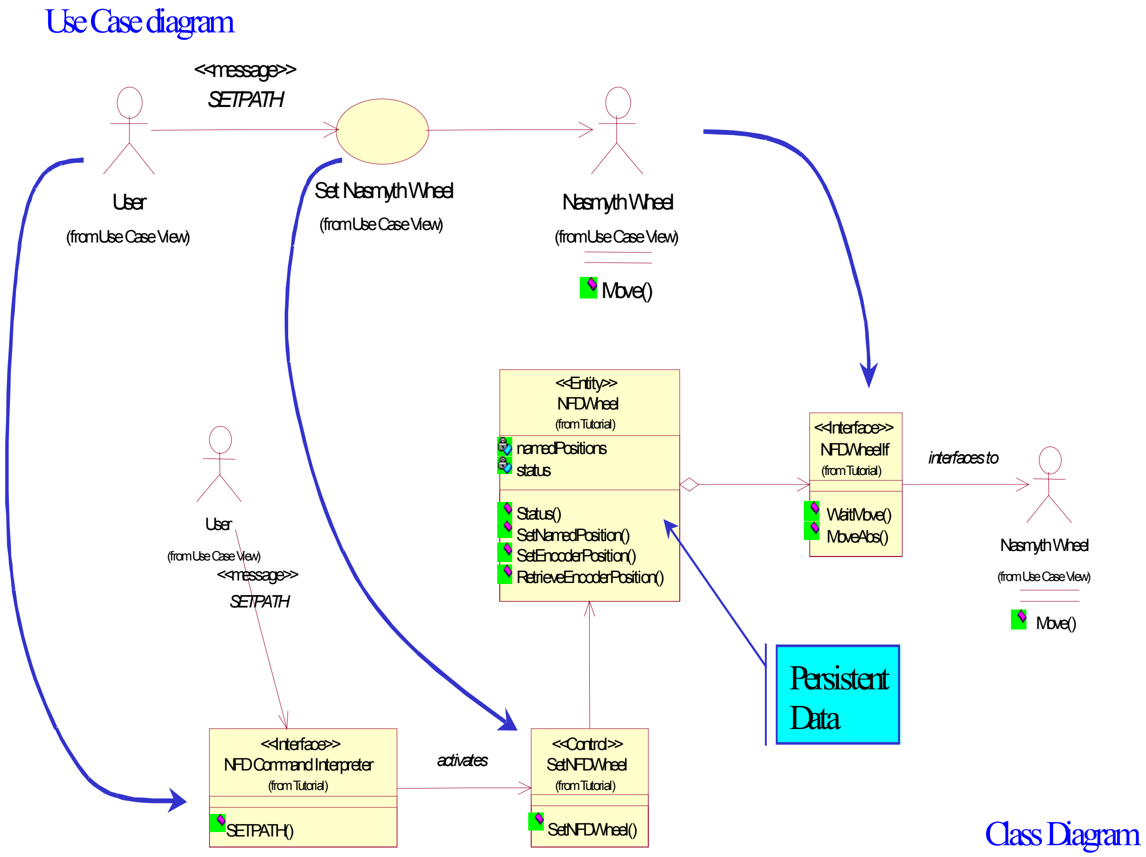


Figure 4

The basic architecture classes of a package are obtained from the step by step analysis of all the Use Cases under the responsibility of the package itself.

These classes always fit in one of the following basic three categories:

- **Boundary classes**
  A boundary class is used to model the interaction between the package and the Actors, i.e. an exchange of information or of action requests between the package and the Actors or other packages in the system.
  A change in an interface is usually isolated in one or more boundary classes.
  In every packages there is typically one boundary class per every Actor interacting with it. It implements all interactions identified in the Use Cases assigned to the package.

- **Entity classes**
  An entity class holds information that typically lasts beyond the life of a Use Case.
  Entity classes are identified by:
  - Finding from each Use Case description the information-bearing objects
  - Finding them from the problem domain
  - Finding them from the original requirement documents

Only classes needed in some Use Case must be introduced. One has to begin the search in the Use Cases and use the other sources to confirm the choice and structure of the classes identified.

- **Control classes**
  A control class represents coordination, sequencing, transactions and control of other objects. The dynamics of the system are modeled by control classes.
  There is typically a control class per each Use Case, although simple Use Cases may not need a control class.

3.7 Construction leads to System Delivery

Purpose:

- Start building, using the previous work
- Attain initial operation capability of the product
- Developers implement the design
- Integrate software, generate system and acceptance test plan

Steps:

- Detailed module design
- Document module
- Code module
- Create and test module, provide code documentation
- Integrate into integration test
- Develop system test plan
- Develop acceptance test plan
- Examine UML diagrams with checklists
- Formal review of each detailed module design

Artefacts:

- Update and detail all deliverables issued for the Software Requirements Review
- Executable systems
- Draft user and support documentation
- Detailed unit design
- UML diagrams (state, sequence)
- UML component diagrams
- Code modules with documentation and automatic regression tests

Documentation kit:

Online documentation (elaborated UML model)

Coded, tested and archived modules

System test plan

Acceptance test plan

Purpose

The purpose of the Construction Phase is to actually build the system.

The System Delivery milestone marks the end of the Construction Phase and shall demonstrate that the system has reached a level of product capability suitable for initial operation in the final environment.

The system will still contain bugs and imperfections, but can be used. For commercial application software, this milestone corresponds to the release of the first beta version of the product.

Artifacts

In the Construction Phase, the emphasis shifts from the accumulation of the knowledge necessary to build the system to its actual construction.

- **Packages are assigned to software developers**, together with all the Use Cases that have been assigned to them in the previous phase and that have to be implemented for the current iteration.
- **Packages are implemented independently one each other.** It is essential that interfaces with Actors and between co-operating packages are defined in details. Every package must have its own package (modular) test.
- **Package implementation means implementation of all Use Cases (or part of Use Cases) assigned to it.** The whole process is Use Case driven.
- **Packages are tested independently (modular testing).** Black box testing is based on Use Cases. Scenarios are used to produce Test Cases. Stubs replace external packages. Open box test is based on Class Tests.
- **Each iteration is closed by system integration.** Frequent system integration allows identifying early interface problems. Test Cases for system integration are obtained from the Use Cases.

The prototype developed for the Elaboration phase becomes now a "Control Model" and used for modular testing of subsystem packages (that need to have access to specific electronic or hardware components) and for integration testing.

The detailed module design is exclusively done using tools supporting UML modeling. No new documents are produced. Detailed module designs are reviewed online.

Component diagrams are created, which derive from the package and class diagrams and map directly to implementation units.

If detailed design affects the system and high level design the corresponding documentation has to be updated.

The code documentation is integrated in the online documentation to have a complete reference available. In particular they are linked to the component diagrams.

Code is organized in modules where each contains a regression test that can be run in an autonomous and automatic way.

3.8 Transition leads to Final Acceptance Test

Purpose:

- Make the system ready for unrestricted release to the user community
- Perform system and acceptance tests
- Use Case are used to develop test procedures

Steps:

- Integration, testing and delivery activities

Artifacts:

- Update and detail all deliverables issued for the Software Requirements Review
- External release of final system
- Acceptance test procedure reports
- Final user and maintenance documentation

Documentation kit:

User and maintenance documentation

Test reports

Purpose

The purpose of the Transition Phase is to make the system ready for unrestricted release to the user community. In ESO terms, it corresponds to the end of the commissioning phase for the VLT.

Artifacts

A **Final Acceptance Test** is set at the end of the Transition Phase.

The only area that needs changes to be properly integrated in the Use Case driven process is the definition of the Test Cases. Test Cases for final acceptance are directly extracted from the Use Cases. This is needed in order to meet the objective of tracing requirements through the whole process down to final acceptance test by using Use Cases. All test procedures must be fully automatic or, when this is not possible, based on a detailed checklist. The tools developed for this purpose in the VLT project are very powerful.

Use Cases are valuable also for writing maintenance and user documentation.

The table below summarizes the phases and their respective artifacts:

SDD **Design Task** **Section ⇓**	**Typical Contents** **as Needed** ⇓	**Project Phases** **TIME ⇒**	**Inception**	**Elaboration**	**Construction**	**Transition**
Status **&** **History**	Life cycle status Revisions Signoffs	**Activities** ⇒	✓	✓	✓	✓
Definition: **Requirements** **&** **Analysis**	Problem description: • System description • Interfaces In • Hardware • Software • Requirements: • Functional • Non-functional • Actors & Agents • Risks • Assumptions		✓	✓		
Design	Baseline Architecture High Level Design Package Divisions • Interfaces Out • Hardware • Software UML diagrams: • Use Cases • Conceptual Model • Class • Activity • Collaboration • Sequence • Package • Deployment			✓		

Implementation	Detailed Design: Coding Spec Algorithms UML diagrams: • Activity • Collaboration • Sequence • State Machine • Class					✓	
Test							✓
Acceptance							✓
Delivery							✓
Glossary				✓	✓	✓	✓

3.9 Use Cases

Use Cases drive the whole software process and bind together all the phases from requirements capture to final delivery of the system and maintenance.

They are a very effective way of communicating with customers and among team members. Before every discussion we always provide the partners with a set of relevant Use Cases.

During meetings, they stimulate focused discussions and help identifying important details.

3.9.1 WHAT and HOW

It is important to keep in mind that Use Cases have to **describe WHAT** the system has to do in response to certain external stimuli and **NOT HOW** it will do it. The HOW is part of the architecture and of the design.

It if often not easy to keep these two aspects separated. In particular, during the project's phases, Use Cases evolve and many details are added. When Use Cases are assigned to packages (see chapter 2) and the black box representing the system is opened, all Use Cases are detailed, adding the communication between packages. This information is of no use for readers interested in the system only as a black box.

Two levels of access to the Use Cases are provided, introducing a ***design view***, identified by the ↳ symbol. This view contains details that are pertinent only to design and can be skipped to obtain the ***requirement view*** only.

The "**Notes:**" section in the Use Cases is used to describe non-functional requirements directly related to a specific use case.

3.9.2 Use Case Diagrams

The interrelations between Use Cases and Actors are expressed in graphical form by drawing Use Case Diagrams using the UML.

These diagrams are important to give an overview of the functionality of the system, but their understanding clearly requires a minimal knowledge of the UML syntax and this cannot be assumed for all stakeholders, which are often non-technical people.

For this reason there exists on one side the Use Case diagrams and on the other side the HTML tables of contents of the Use Cases, grouped according to different criteria. All the relations that appear in the Use Case Diagrams as hyperlinks are implemented in the online version of the Use Cases.

An example of Use Case diagram, representing the ATCS Use Cases related with telescope tracking, is given in Figure 5 below:

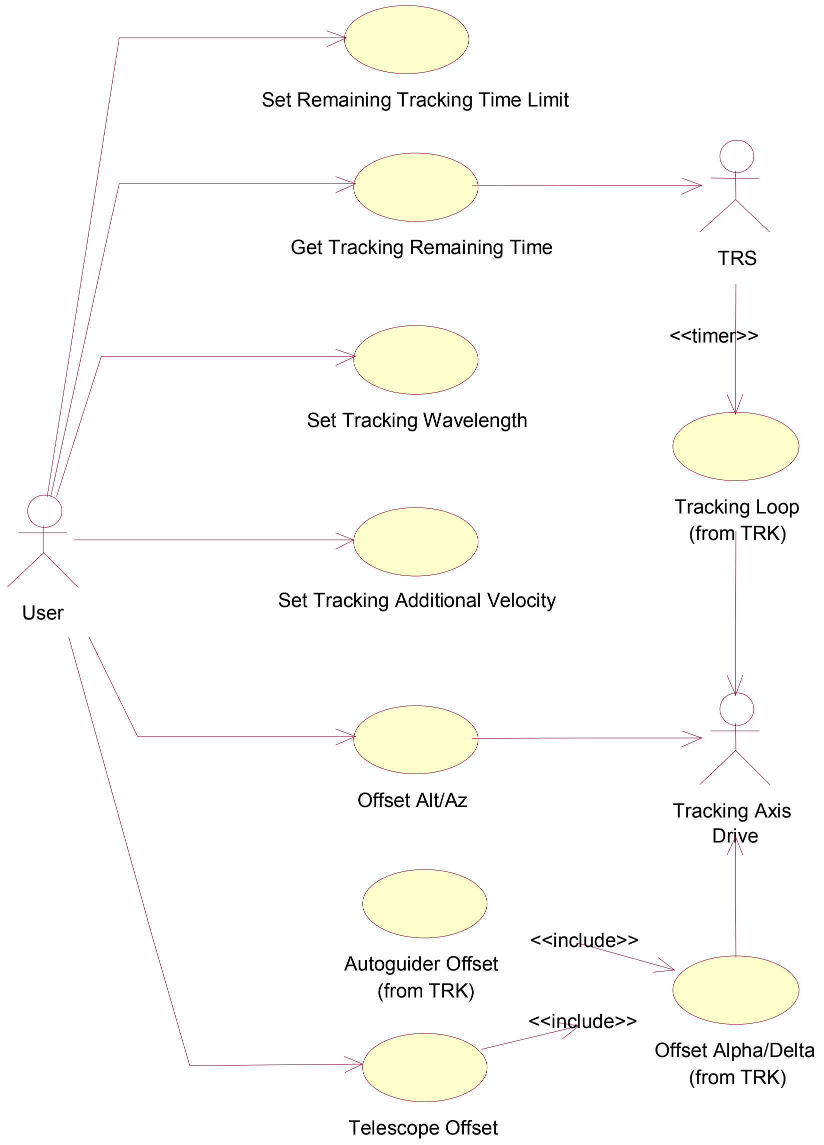


Figure 5- Tracking Use Case diagram

The syntax is rather intuitive:

- An ellipse represents a Use Case, i.e. it is linked with the textual description of the Use Case with that name.
- A stick man represents an Actor, no matter if human or not
- A line connecting two entities represents an association. The arrow represents the direction of the association.

For more details on the Use Case Diagram syntax, see [1].

The diagram represents a group of Use Cases and describes which Actors interacts with the system when a Use Case is executed and shows relations between the Use Cases. Selecting one of the Use Cases, the textual description of the Use Case itself is opened.

3.9.3 Use Cases: scope and target readers

It is very important to agree and define on the scope of the Use Cases and on the target readers.

The following constraints have to be taken into account:

- which type of software we are going to develop (for example: control systems, i.e. software that directly controls custom hardware and electronics);
- who are the main contact points and interlocutors during the project;
- who are the "users": in most cases, other software systems or human operators with a high degree of specialization and with a good knowledge of the architecture of the system;
- graphical user interfaces are not a primary focus of the development. User interfaces are considered as an independent layer on top of the application and use the same interfaces that are used to interact with other systems or between packages (see chapter 3.6). Engineering User Interfaces are built together with the software packages when needed, while final operator user interfaces are typically developed during the transition phase (see chapter 3.8) by the commissioning team, in tight co-operation with operators and final users of the system.

Taking these aspects into account, it can be decided which type of Use Cases we are going to write. At this point it is not possible to set up a unique rule, a Use Case can be small or large, more or less detailed.

Here follow some general rules:

- At the first phase of the project, the use cases should be not more of the 10% of the total; they should describe the system at “high level”;
- The detail level depends on the phases of the project (the more advanced is the project, the more detailed use cases can be written) and on the interlocutors or contact points for whom the Use Cases are written;
- When a choice is done, it is very important to keep it coherent throughout the whole process: all Use Cases must be "in the same style" and with the same type of contents. This can be achieved with a good communication and with a continuous exchange of information between the team members responsible for Use Case development and maintenance.

4 Reference

This chapter describes the practical work that has to be done during the different phases of a project. The tools to be used will be presented as well as some procedures that have the only aim of making work easier and faster. We report also the version of the tools with which we tested the procedures.

4.1 Setup of the working environment

The system needs a UNIX workstation as repository for the database (all the work that has to be done should be stored on it) and one or more PCs to run the Analysis and Design tools (Rational Rose).

The system has been tested in the following configuration:

UNIX (HP 10.20) with the following tools installed:

- **gnu make** 3.75
- **tcl** 8.2
- **scripts** contained under the directory src in the template module (see section 4.2).
- **a tool** to perform an automatic check on the web consistency. We use **MomSpider**, 1.00, that needs Perl
- **NFS exported directory** to the PCs (we use Samba 2.0.4b)

PC (WINDOWS NT 4.0 SP 4) with the following tools installed:

- **MS Word**, 97-SR 2, used for:
  - editing online pages
  - automatic generation of printable documents from online pages
- **Rational Rose Modeler Edition** (or **Professional C++ Edition)**, 2000e, used for the UML Model
- **DocEXPRESS/Reporter LITE**, 2.0, used to generate HTML files from Model files
- A **tool** to handle planning diagrams (for example MS project)

4.2 Module template

The work is accomplished using a module under configuration control. Everything has to be done within this module: the work with Rational Rose, any kind of diagram and any kind of documentation in HTML as well as MS Word format.

The module template is available on the web at the ESO ftp site. It can be retrieved by anonymous ftp from the directory:

/pub/vlt/vlt/pub/tmp

Retrieve the tar-gzipped file: almatpl.tar.gz and run:

```
gunzip almatpl.tar.gz
tar xf almatpl.tar
```

to get the template module directory

almatpl

Then, you can customize the module and change the name accordingly to the project you are working with.

The module has the following directory structure:

1. **index.htm**
2. **ChangeLog**
3. **HowTo.htm**
4. **Images**
    - arrow.gif
    - eso-logo.gif
    - line1.gif
    - redball.gif
5. **ArchiveDocuments**
    - ALMA-XXXX
    - README
6. **MilestoneReleases**
    - **FDR**
        - ALMA-XXXX
        - actionItems.txt
        - commentList.txt
        - commentsProcessed.txt
        - commentsReviewed.txt
        - newRequirements.txt
        - reviewCall.txt
    - **PDR**
    - **SRS**
7. **Model**
    - Help
    - ICD
    - Packages
    - Project
    - Rose
    - UseCases
8. **Plan**
9. **src**
10. **bin**
11. **object**

Here follows the detailed description of the directory structure of the template module. (The same description is supplied in a README file per each directory of the module.)

1. **index.htm**

   This file contains the index for the on-line documentation. (Beware that in the file, the model generated with Rational Rose as well as the html file generated with the Web Publisher are called rose.mdl and rose.htm. If you change the name, change accordingly the references to the name “rose” in the index.htm file).

   The Table Of Contents has three main sections.

   - **Online Documentation**

     This section is the main entry point to the web of hyper-linked documents; from this point every item can be reached.

     We have organized it to have a tree skeleton: directly reachable from the main page are overview documents, that link to more detailed pages.

     The purpose is to give to the occasional reader the possibility of getting easily a global picture of the system, while the experienced reader can in a few jumps go to the required level of detail.

     From low level information, transversal hyperlinks lead directly to detailed information in parallel branches of the documentation tree.

     To increase navigability we have made use of frames. Typically the left side is a table of contents, while the right side contains the selected item.

   - **Printable documents - Official releases**

     Documents are officially released at major milestones to keep track of the history of the project. This section points to the printable documents officially released and sent to the ESO archive, to allow download and printing (see directory MilestoneReleases in the module template).

     This section allows accessing the printable documents that have been officially reviewed by the stakeholders. It gives a picture of the evolution of the project through the various phases, marked by the major milestones.

     Reviewers are always encouraged to access the online documentation, but we have seen that they rather prefer to work on the paper documents. This is due probably to the fact that most of the people are used to work on paper documentation and reviews are typically done offline.

     We know that it is anyway not convenient to read on the computer screen a big amount of documentation.

   - **Printable documents - Current development**

     This section points to the current development of all printable documents, i.e. to the directories where the document's sources are kept and updated (see directory ArchiveDocuments in the module template).

2. **ChangeLog**

   This is a file where each user must document the modification yielded to the module, following the syntax:

   <modified_file> : one-sentence description of the modification

3. **HowTo.htm**

   This file contains the procedures that are described in the following sections.

4. **Images**

   In this directory some standard images used in the documentation are stored.

5. **ArchiveDocuments**

   In this directory we put all the documents under editing. For each document we prepare a directory, named accordingly to the ALMA standards. The README file gives some instruction about the standard documentation layout. At the time being, an ALMA document template is under preparation. We supply a first version of the WORD template, called: DocumentAlma.doc.

6. **MilestoneReleases**

   This directory collects the documents have been released at each milestones (the official ones and the notes, comments, review calls and minutes have been produced). This documentation is not modified any more. Basically, when a document is consolidated, it is copied from the directory ArchiveDocuments to this one.

7. **Model**

   Under the directory **Help** a set of html files can be found. Two of them have to be edited and customized: Abbreviations.html and Glossary.html. The others are automatically generated and represent an on-line help.

   The directories: **ICD**, **Packages** and **Use Cases** have a common structure: four html files handle the initial page with a frame layout:

   the left-side frame (handled by the file IndexIp.htm) contains the list of ICD (Interface Control Document), Packages or Use Cases of the project; all these items have to be created as html files in the corresponding directory and listed in the IndexIp.htm file. In the UseCases directory the template for the use case can be found, (UseCaseTemplate.htm), as well as the UseCaseTemplateDoc.htm. This second file contains the explanation for the different sections of the Use Case Template; we supply also the Collaboration Case template (CcaseTemplate.htm): the only difference with the Use Case template is that in the Collaboration Case template there is not the Priority section (see section 4.4.1);

   the central frame (technically speaking this is the *target* frame, where the pages resulted from the "click" on a link in the left-side frame are open) is a general description of what is contained in the corresponding directory;

   the bottom frame contains some helpful definitions.

   The file Index.htm contains the necessary definitions for the frame layout of the page.

   Project contains some files for a general description of the project, like the Actors involved, the general requirements of the project, applicable documents, reference documents and so on. The names of the files are self-explanatory.

Rose contains the model created with Rose: rose.mdl and two directories: Html where all the files generated with the Rose Web Publisher are stored and doc where the files created with DocExpress are put (see the following chapter to better understand the meaning of these two directories).

8. **Plan**

   This is the directory where the documents concerning the planning of the project can be put.

9. **Src**

   Under this directory, you find the Makefile and a set of scripts that will be described in the following section. Beware that, if you rename the module template from almatpl to <your_name> you should change accordingly all the references to the almatpl name in the scripts and in the Makefile in the directory src. In the same way, we reference the Rose model as the file rose.mdl and the corresponding html file as rose.htm. Again, if you change the name, change accordingly the files under the src directory.

10. **Bin**

    In this directory some executable files (generated running *make* under src) are put.

11. **Object**

    In this directory some object files (generated running *make* under src) are put.

4.3 Organizing the project

The overall project has to be divided in sub-projects. For each sub-project a module is created that contains all relevant documentation. It means that you generate different modules with different names from the module template.

The interfaces between these modules are defined with ICDs.

A natural sub-division of the ALMA project could be to create a module for Correlator, Antenna, Data flow, Receiver etc.

4.4 Working within the module

Once the module has been created, a first customization work has to be done on the files supplied by the template. They are html files that can be edited with word. (Other tools are, at present, under examination, like Dreamweaver for the editing of html pages).

Then, the first use cases can be written, in html format as well as within the Rose model.

4.4.1 Working with Rational Rose

In this section, we intend to give some advice useful in organising the work with the Rational Rose tool. We recommend you to follow them to obtain a model with a standard structure and to make easier the communication with other groups.

✔TIP

> Starting with the Rose model, it's very useful to set the path to the module directory as the environment variable ALMATPL_ROOT, so that any other user who wants to work to the module from another directory can easily find all the references and links just setting accordingly this variable. From the menu Edit choose Path Map and set
>
> $ALMATPL_ROOT to the desired path.

- The first use cases you write belong to the requirement phase. It is suggested to create them under the Use Case View with a flat structure. In a later stage, these use cases can be organized in packages under the Use Case View and design information can be added. Each package will contain related use cases.

  At the same time, the need for writing new use cases will arise. All the use cases created after the requirement phase are called Collaboration Case and can be prepared using the template CcaseTemplate.htm supplied with the module in the UseCases directory.

  This second type of use cases should be kept separated from the previous one. It is recommended to organize the Collaboration Cases in packages under the Logical View.

- On the repository, under the directory Model/UseCases in the module, you should create a directory per each package; when a directory corresponds to a package containing Collaboration Cases, the directory name must be in the form:

  <name>-CC

  CC obviously means Collaboration Case and must be upper case.

- When you create diagrams, the name should be prefixed with the package name.

✔TIP

> If a diagram is created, don't let it empty, if not DocExpress will generate an output that doesn't meet the structure we need for parsing.

- The Actor package has to be created in the Use Case View.
- After creating the use cases, packages and actors in the Rose model, the following links have to be defined:
  - Each Use Case bubble has to be linked with the corresponding Use Case html file.
  - Each Package contained in the Logical View has to be linked with the corresponding package description.
  - Each Actor has to be linked with the corresponding ICD html file.

  To do this, from the Rose tool, select an item (use case, package or actor), right click on it and choose "Open Specification…". A window pops up. Select "Files". Right click on the window and choose "Insert File". Type the path to the htm file corresponding to your item. The link is done.

4.4.2 Creating printable documents: Web Publisher

When it is necessary to generate a printable document, perform the following steps to create the htm files from the Rose model, using the Web Publisher in the Rational Rose product:

- delete $ALMATPL_ROOT\Model\Rose\Html\*
- Open the model with Rose and load ALL sub-units
- Select Tools->Web Publisher
- Select as Level of Detail: Intermediate.
- Select as Root File Name: $ALMATPL_ROOT\Model\Rose\Html\rose.htm
- Press Publish and Wait...
- Open a Unix shell, cd to almatpl/src and execute:
- **make all**
- **make web** (NEVER run more than once after published by Rose)
  All loose ends in the model are linked to a file containing some information about the loose end. Since there might be several thousands of them, we remove them and replace them by one single file.
- Optional: note the drive where the Rose model is on your PC (just the lower-case letter) and run
  make DRIVE=<your_drive> replace
  The web publisher tool produces links that have the Windows path in it. We don't need them and we replace the absolute paths with the relative ones
- If you have a web server, transfer the created files to it.

The Rational Rose Web Publisher automatically generates all the Rose diagrams in one of these formats: jpeg, png (Portable Network Graphics), Windows Bitmaps. The default is png. You can change the default from the Rose Web Publisher, pressing the bottom “Diagrams…”. By the quality point of view, the best one is png, but it is not enough for a printable document.

4.4.3 Generate diagrams in EMF and GIF format from the Rose Model

To get good quality printable diagrams from Rose, use the product ATA DocExpress/Reporter LITE. With this product (and some final touches made by the script XP2IMG.tcl supplied in the src directory), two files per diagram are generated in the format gif (for the web) and emf (for the printable document). This is the procedure:

- Open the model ($ALMATPL_ROOT\Model\Rose\rose.mdl) in Rational Rose under Windows
- Select the menu: Report -> Document with DocExpress…
- In the welcome panel, click on "Continue…" and in the report panel, select:

    1.Report Format is HTML

    2.Report Type is Pre-Defined Reports

    3.Next

- In the pre-defined reports [HTML] panel, select:

1.Report Type Name is Entire Model

2.Set the output file name to $ALMATPL_ROOT\Model\Rose\doc\<modelName>.html[2]

3.Finish

Output

docExpress generates the following files:

<modelName>.html: Full html view.

<modelName>_FV.html: Full html view using frames for TOC, LOF and LOT.

<modelName>_TOC.html: Table of content

<modelName>_LOF.html: List of Figures

<modelName>_LOT.html: List of Tables

tmp_<mmddhhmnssn>.EMF: Diagram in EMF format and

tmp_<mmddhhmnssn>.gif: Diagram in GIF format

where <mmddhhmnssn> indicates the generation date (month, day, hour, minute, second) followed by a current index.

- Generate the final image files for the online and for the printable documentation: open a Unix shell, cd to almatpl/src, change in the Makefile the string rose.html with <modelName>.html[3], prepare the lookup tables in the XP2IMG.tcl script, under src (see next section for an explanation on how to use and maintain the XP2IMG script) and execute

  make all

  make diagrams

4.4.4 How to use and maintain the script XP2IMG.tcl

The Makefile calls the tool XP2IMG (located under bin).

It is an automatic procedure that parses the file Model/Rose/doc/<modelName>.htm and renames the image files tmp_*.EMF and tmp_*.gif to their assigned named into the Images directories of Model/UseCases and Model/Packages respectively. (The file extension of the .EMF files is set to .emf).

The lookup tables are maintained inside the script source and are the consistency links between the model and the documentation. You should edit the script accordingly to the diagrams you prepared in your model. In the script, in the MAIN section[4] you find the following tables:

[2] Be careful that, even if you put the extension 'htm', the DocExpress tool will create a file with 'html' extension.

[3] Note the 'html' extension: DocExpress generates files with the 4-letters extension.

[4] A new version of the script is in preparation, where the tables will be supplied as separated files.

- Deployment Diagrams
- Activity Diagrams
- Sequence Diagrams
- State Diagrams
- Component Diagrams
- Use Case Diagrams
- Class Diagrams

In each table, there is a list of the files that have to be processed by the script. The format of each record in the list is the following:

{new_name "title" where}

In braces, you should put:

- First field: your name for the diagram;

  warning: use always different names. If the same diagram is in both the Logical and UseCase View, it is recommended to call it <name>lv and <name>uc for the two views.
- Second field: string that correspond to the title for the diagram as it is in the almatpl/Model/Rose/doc/<modelName>.htm file;
- Third field: location for the diagram; you can put **lv** (Logical View) or **uc** (UseCases view). The final image will be put in the Package/Images directory or, respectively, in the UseCases/Images directory.

4.4.5 Generate filtered files for DD and SRS documentation

Printable documents can include only information relevant at requirements level or also all available information, added at design time or after design. With this final phase of the procedure, we generate both type of documents, running

make filter

from the src directory of the template module (it is assumed that you have previously run **make all**).

The Makefile does everything automatically, using the scripts: HTML2DD.tcl and HTML2SRS.tcl.

The result of this action is the creation of two sub-directories under almatpl/object:

SRS and DD.

In these sub-directories (SRS for “requirements” documents and DD for “design” documents) you will find all is necessary to build a word document: al the html files and all the images in gif as well as emf format. (We remind you that the emf format is the most suitable for a word document).

4.4.6 Working with word

Starting from the word document template DocumentAlma.doc contained under almatpl/ArchiveDocuments/ALMA-XXXX you can simply insert the different html files and images prepared under the SRS or DD directory.

Open the document under Microsoft Word, choose Insert -> File…. Check the box "Link to file" and select the desired file.

Warning: the inserted file has an absolute path hard-coded. You can easily check and eventually change this path following this procedure:

- select the inserted document;
- click with the right button of the mouse;
- choose "Toggle Field Codes".

✔TIP

When a document is released, it must be copied from the ArchiveDocuments directory to the MilestoneReleases directory. Before the copy is done, all the links should be removed from the document.

To remove the links, open the document with Microsoft Word and select the menu Edit -> Links…

A window called "Links" will appear. In this window, select all the Source files and press the button "break link". The source files containing pictures will remain in the list. To remove these links, check the box "Save picture in document" and press again the button "break link".

4.4.7 Checking the web consistency

The web consistency checking can be performed with the MomSpider tools. We are aware that a lot of other tools can be used. We give support for this one, but anyone is free to use the tool he prefers.

To use MomSpider, open a UNIX shell and run:

momspider

The results are put in the $ALMATPL_ROOT/MomSpider directory.

Look in particular for the "Broken links" section.

4.5 To be done

- The integration between the tools is not complete.
- A lot of work has to be done by hand or by writing support scripts and tools: the procedure has to be simplified or improved.
- New tools are under examination.

4.6 Troubleshooting: surviving MS-Word

Microsoft Word has many bugs and nuances, but with some tricks it is possible to survive (we are using MS-Word 97 SR2):

- To avoid crashes and corrupted documents when updating documents with many external links:
  - In the Tools->Option->Save panel, toggle OFF "Allow Fast Save" and "Allow Background Save" options
  - To update all links, always take the Edit->Links panel, select the links you want (also all) and update. DO NOT update many links directly from the editable area.
- It is better to open a word document launching Word and choosing the menu File->Open, instead of clicking on the document from the "Explore" tool. In this case, the current directory for Word is the default one, very likely different from the directory where you have all your linked documents. If you cannot find some imported files, close every document and reopen only the one you want from the File->Open menu.
- For the same reason, if document have links to HTML files, open and edit only one at the time. Word has just one search path for included files and gets confused if you have more documents open.
- Always update the TOC of a document just before printing. Go to the TOC and press F9. Otherwise it is very common to get "ERROR: Bookmark not found"

5 APPENDIX A: Use Case template

5.1 The Use Case template

Use Cases are written as structured text using a formal template. The usage of plain text facilitates the understanding of the Use Case by people with different background. A formal template forces a recognizable and common layout; relevant information is easier to find and communication is optimized.

What follows is the empty template:

1.1 Use Case: Use Case Template

[Provide a sentence or short paragraph that clearly defines the purpose or goal of the use case. A use case should have only ***one*** *goal. Using goals helps specify the scope of a use case.]*

Role(s)/Actor(s):
Primary:
Secondary:

Priority: ...

Performance: ...

Frequency: ...

Preconditions:

1. ...

Basic Course:

1. ...
   Alternate Course: ...
   Exception Course: ...
   Postcondition: ...

2. ...
   Alternate Course: ...
   Exception Course: ...
   Postcondition: ...

Subflow:

1. Do first step
   Alternate Course: Optional Alternate Course Title1.
   Exception Course: Optional Exception Course Title 1.
   Postcondition:

2. Do second step

Alternate Course:

1. ...
   Exception Course: ...
   Postcondition: ...

2. ...
   Exception Course: ...
   Postcondition: ...

Exception Course:

1. ...
   Postcondition: ...

2. ...

   Postcondition: ...

Postconditions:

1. ...

2. ...

Issues to be Determined or Resolved: ...

Notes:...

Last modified: Thu May 11 11:53:19 UTC 2000

The template contains many sections to be filled in, although it is not mandatory to fill in all of them.

Some examples are given in appendix A.

We do not go here into too many details, for which we have already given the proper reference documentation.

We just want to stress that an unstructured description of the course of a Use Case in not sufficient, because it leaves too much space to interpretation and does not force people to identify and discuss systematically important aspects of the system.

The purpose of the Use Case is to describe with as much details as possible the interactions with the system under development and the actions that it performs, seen as a black box, in response to the stimuli coming from the Actors. Details are added during the project phases as they are analysed and are discussed with the stakeholders.

In the following sub-sections we describe the most important fields in the Use Case template.

Role(s)/Actor(s):

[An actor is a role of an entity external to the system. Actors can be humans, machines, or devices.
"One physical object may play several roles and therefore be modeled by several actors".
A primary actor is one having a goal requiring the assistance of the system.
A secondary actor is one from which the system needs assistance to satisfy its goal.
One of the actors is designated as the system under discussion.]

Primary:
Secondary:

Priority: ...

[How critical is the Use Case to the system:

- *Critical: "I absolutely must have this function for any real system"*
- *Major: "I can live without this function for a short period"*
- *Desirable: "It is an important function, but I can survive without it for a while".]*

Performance: ...

[The amount of time the Use Case should take]

Frequency: ...

[How often the Use Case is expected to be performed]

Preconditions:

[Preconditions indicate circumstances that must be true prior to the invocation of the Use Case. If preconditions have an ordered sequence, display them in an ordered list. If this Use Case depends on a previous Use Case's successful execution, list the previously executed Use Case in this section. If a precondition is not satisfied, the final state of the Use Case is undefined.]

1. ...

Basic Course:

[Each Use Case has only one basic course. The basic (or main) course is the sequence of steps that the Role or Actor is likely to take in order to accomplish the goal of the Use Case. The first step describes the action that initiates the Use Case.

Each step can optionally have alternate course(s), exception course(s), and/or postconditions. The postconditions of the last step is not necessary because the postcondition of the Use Case (validates that the Use Case goal was met) should be valid.

Steps can optionally refer to other Use Cases for "use relationships" or Collaboration Cases for traceability. Alternate and exception courses should be defined within the same Use Case.

***Use relationships** occur when you have a chunk of behavior that is similar across more than one use case and you don't want to keep copying the description of that behavior. Use it when you are repeating yourself in two or more separate uses cases and you want to avoid repetition.*

***Extend relationship** occur when you have one use case that is similar to another use case but does a bit more. Use extend when you are describing a variation on normal behavior.]*

1. ...
   Alternate Course: ...
   Exception Course: ...
   Postcondition: ...
2. ...
   Alternate Course: ...
   Exception Course: ...
   Postcondition: ...

Subflow:

[OPTIONAL - Each Use Case has zero to many subflows. A Subflow is a sequence of actions within a Basic, Alternate or Exception Course that is identified with a unique name. A Subflow acts as a building block of Courses. Courses can (if applicable) be build up with Subflows.

For example if we have a set of closely related requirements that form a single goal and help to clarify the requirement more as if we moved them to individual Uses Cases.

All the rules of Basic Courses apply also to Subflows (i.e. they can have Alternate flows, Exception flows etc.)]

1. Do first step
   Alternate Course: Optional Alternate Course Title1.
   Exception Course: Optional Exception Course Title 1.
   Postcondition: Optional step postcondition description
2. Do second step

Alternate Course:

[OPTIONAL - Each Use Case has zero to many alternate courses. The alternate course is a different sequence of steps that the Role or Actor can take that also accomplishes the goal of the Use Case. The alternate course extends the basic course with additional steps. The title of each

alternate course should appear exactly as written in the cross-referencing Basic Course step. Alternate courses may include exception courses and optional postconditions. NOTE: This is the same as the UML's "extend relationship" of Use Cases.]

1. ...
   Exception Course: ...
   Postcondition: ...
2. ...
   Exception Course: ...
   Postcondition: ...

Exception Course:

[OPTIONAL - Each Use Case has zero to many exception courses. The exception course is a sequence of steps that the Role or Actor takes when the task is interrupted or a single postcondition that validates the exception course was taken. The exception course indicates the cause of the interruption, then indicates how the Role or Actor recovers. Exception courses must provide at least a postcondition for the step that validates the exception course that was taken. The postcondition that validates the goal of the Use Case is not necessarily true after the exception course has been taken. The title of each exception course should appear exactly as it is in cross-referencing Basic Course step.]

1. ...
   Postcondition: ...
   [Optional step postcondition description]
2. ...

Postcondition: ...
[Mandatory postconditions that validate the exception course.]

Postconditions:

[The postconditions validate that the basic course and all alternate courses successfully achieved the stated goal of the Use Case. The postconditions are not necessarily true if an exception course was taken.]

1. ...
2. ...

Issues to be Determined or Resolved: ...

[OPTIONAL - List any issues that remain to be decided, or to be reviewed regarding this use case. These issues may include scope of the use case, task description, or user interface.]

Notes:...

[OPTIONAL - List any note that help in clarifying the Use Case, but that do not fit in any other standard Use Case field.]

Last modified: Fri May 12 11:34:48 UTC 2000

The Exception Course, Preconditions and Postconditions sections are very important, since they are traditionally not taken into account in other requirement capture methodologies. The fact of having to fill in these sections at each step in the Use Case helps in identifying the requirements related to the handling of unexpected and exceptional conditions (i.e. what the system must be able to do by itself, when and what manual interventions are necessary, what information must be preserved and so on). **In particular this forces a discussion with the stakeholders that in our experience otherwise will take place only at system delivery**.

5.2 Requirement and Design view: how to create them

To mark a text as Design view, you can tag it using a special comment instruction:

```
<!-- VLT-DD->
```

The scripts used to parse the files in the rose model and supplied with the module template are made in such a way that they can automatically recognize the text to be skipped via this particular tag. You can also choose weather to mark the text with an arrow (as you can see in the example in Appendix B) or not.

To enclose simple text, with no "arrow marker":

```
<!-- VLT-DD --><FONT FACE="Courier">
<B>.... your text.....</B>
</FONT><!-- /VLT-DD -->
```

To enclose text with the arrow on the left side you need a table. The arrow is a gif image arrow.gif supplied with the template module in the directory Images.

Depending on where you are in the file tree of the module you need to fix the path for the arrow image and for the link to the DD help information.

```
<!-- VLT-DD --><TABLE>  <TR VALIGN="top">
<TD WIDTH=12 BGCOLOR="#D0D0D0"><A HREF="../Help/DDEntry.htm"
TARGET="Help"><IMG SRC="../../../Images/arrow.gif" ALT="[Design
Description Entry]"></A>
<TD> ... this is the text to be marked as "design information" .....
</TR></TABLE><!-- /VLT-DD -->
```

6 APPENDIX B: Use Case examples

This section contains a few examples of Use Cases extracted from the ATCS Online Documentation [8]. They are put here just to give the feeling of how our Use Cases look like and to allow a first check of the concept described in the text. It is anyway suggested to look at the ATCS Online Documentation [8].

6.1 Use Case: ATCS Start

Description: Each required sub-system used during observation is brought from state STANDBY to state ONLINE, i.e. all sub-systems are completely activated and the telescope is under position control, ready to move at next operative command.
The brakes are disengaged, the control loops closed, A failure will leave the ATCS in state STANDBY substate ERROR.

Use Case Type: Concrete

Role(s)/Actor(s):
Primary: Operator, Maintenance
Secondary: Standard Packages

Priority: Major

Performance: A few minutes

Frequency: Whenever necessary.

Preconditions:

- ATCS is in state STANDBY

Basic Course: Start automatically

- Send command **`START`** to Start ATCS
  Exception Course: Command failed
- ATCS puts ONLINE all *required* and *not ignored* subsystems used in *observation* mode

↳ by using Standard Command

Control loops are enabled and brakes disengaged.

↳ (*Interface: IfAltDrive IfAzDrive*)

Exception Course: Some sub-systems failed to go ONLINE
Postcondition: all *required* and *not ignored* subsystems are ONLINE

- ATCS global system state update
  Postcondition: ATCS in state ONLINE, substate IDLE

- ATCS sends final OK reply

Alternate Course: Start manually

- Put ONLINE all *required* and *not ignored* sub-systems one by one.

↳ by using Standard Command from the Control GUI

Exception Course: Some sub-systems failed to go ONLINE
Postcondition: all *required* and *not ignored* subsystems are initialized and in ONLINE state

Exception Course: Command failed

- ATCS global system state update
  Postcondition: ATCS in state STANDBY, substate ERROR
- ATCS sends final ERROR reply

Exception Course: Some sub-systems failed to go ONLINE

- ATCS global system state update
  Postcondition: ATCS in state STANDBY, substate ERROR
- Ignore sub-systems that failed
  Alternate course: Retry to Start manually for failed sub-systems

Postconditions: These are postconditions that validate the goal of the use case.

- ATCS is in state ONLINE substate IDLE
- All *required* and *not ignored* ATS subsystems used in *observation* mode are initialized and ONLINE
  (control loops are enabled and brakes disengaged).

Issues to be Determined or Resolved:

Notes:

Last modified: Mon Jul 3 14:33:10 METDST 2000

6.2 Use Case: Telescope Offset

Description: Offset the telescope position of the given (alpha,delta) step.
The offset value is simply added to the reference position of the telescope. The telescope axes move to (alt,az) values corresponding to the new sky coordinates.
It is seen by the user as a change of the actual position with respect to the last preset (or rather the last position).
An offset step is always relative to the current position of the telescope (i.e . consecutive steps are accumulated). To issue a new offset it's not necessary to wait until a previous offset has completed. Consecutive offsets are therefore not lost if one does not exceed the defined performance limits.
This command CANNOT be used while guiding or doing field stabilization, since the guiding system is not aware of the offset and the behaviour of the system is unpredictable. Use Combined Offset instead.
This UC *uses* Offset Alpha/Delta

Use Case Type: Concrete

Role(s)/Actor(s):
Primary: Operator/Maintenance
Secondary: Alt/Az Axes

Priority: Critical

Performance: up to 1.5 deg/sec in Alt and 2 deg/sec

Frequency: Asynchronous up to 10 Hz

Preconditions:

- ATCS is ONLINE Tracking

Basic Course:

- Send command **OFFSAD** to offset the telescope of (alpha,delta) in arcsec
  Exception Course: Command failed
- ATCS sends the (alpha,delta) offset to the axes by Offset Alpha/Delta
  Exception Course: Command failed
- ATCS updates telescope position
  Exception Course: Command failed
- ATCS returns OK reply

Exception Course: Command failed

- ATCS returns ERROR reply
  Postcondition: ATCS state ONLINE TRACKING

Postconditions: These are postconditions that validate the goal of the use case.

- Alt/Az axis are offset of the (alpha,delta) amount
- Telescope position is updated

Issues to be Determined or Resolved:

Notes: See TCS for UTs

Last modified: Mon Jul 3 14:37:08 METDST 2000

6.3 Use Case: Set Nasmyth Wheel

Description: Set Nasmyth wheel to required position
This action is dedicated to optical alignment procedures and is therefore not foreseen to be used during normal operation
where the Nasmyth wheel shall be set to the specific position allowing the telescope beam to reach M9.
The allowed positions are:

- Flat Retro-Reflecting Mirror
- Free Hole
- 2 positions for Alignment Tools (optical fibres or other dedicated devices)

The Observation mode position is the Free Hole, whereby the device shall be switched Off.
On the Alignment Tool positions, it is possible to chop the wheel between 2 close positions of a given stroke.

Use Case Type: Concrete

Role(s)/Actor(s):
Primary: Maintenance
Secondary: Nasmyth sub-system

Priority: Major

Performance:
less than 10 seconds for motion to a predefined position;
less than 1 second for chopping on Alignment Tool positions.

Frequency: During optical alignment procedures only, typ. before and after alignment procedures.

Preconditions:

- Nasmyth Wheel is ONLINE

Basic Course: Motion to a predefined position

- Send command `SETPATH <FREE|RETRO|TOOL1|TOOL2>` to set the Nasmyth wheel to given predefined position
  Exception Course: Command failed

-

ATCS gets actual Nasmyth Wheel position
Exception Course: Command failed
Subflow: Nasmyth Wheel on position `TOOLn`

-

- ATCS sends command to Nasmyth Wheel device

Interface: *IfNasmythWheel.Motor*

Exception Course: Command failed

- ATCS returns OK reply

Subflow: Nasmyth Wheel on position `TOOLn`

- ATCS switch off Nasmyth Beacon `#n`
  Exception Course: Command failed
  Postcondition: Nasmyth Beacon `#n` switched off

Alternate Course: Chopping on Alignment Tool position

- Send command `SETPATH <TOOL1|TOOL2>,<stroke>` to chop the Nasmyth wheel of a given stroke off the specified Tool position
  Exception Course: Command failed

- ATCS sends command to Nasmyth sub-system

Interface: *IfNasmythWheel.Motor*

Exception Course: Chopping stroke >= 30mm
Exception Course: Command failed

- ATCS returns OK reply

Exception Course: Command failed

- ATCS returns ERROR reply
  Postcondition: Nasmyth wheel not moved

Exception Course: Chopping stroke = 30mm

- ATCS returns ERROR reply
  Postcondition: Nasmyth wheel not moved

Postconditions: These are postconditions that validate the goal of the use case.

- Nasmyth wheel at given position

Issues to be Determined or Resolved:

Notes:

Last modified: Mon Jul 3 14:33:28 METDST 2000

7 APPENDIX C: Interface Control Document example

7.1.1 Nasmyth wheel

- Control of the position of the wheel, including motor, tachometer, encoder and limit switches. This 4-element device consists of a Flat Retro-Reflecting mirror, a Free Hole and 2 positions for dedicated Alignment Tools (2 light beacons, half-masks).
  The motor and tacho are connected to the ESO standard VME4SA Servo Amplifier and the encoder to the MAC4 Motion Controller. The tacho generator is mounted on the motor. The velocity loop is closed in the amplifier. There are two switches: one of them is used as a reference switch.

ESO

- Control of the 2 Image Beacon Light Sources in one of the dedicated Alignment Tools (they are fed into a single optical fibre).

- The control is deployed on the Altitude LCU.
- The device is connected to the Terminal Block **X3**.

ESO Only

The motor control is a pure SW interface: the target position and velocity are passed to the Motion Controller via the VME bus. The motor control software is provided by ESO.
The functionalities to be provided are:

Set absolute position of Nasmyth wheel axis

- Set to Retro-Reflecting mirror
- Set to free hole
- Set 1st Alignment Tool position (Beacon)
- Set 2nd Alignment Tool position (Halfmasks)
- Chop Nasmyth wheel with a given stroke and frequency

Nasmyth wheel interfaces:

Item	From	To	Signal	Description	Pin
MOTOR: Minimotor DC Brush 3557-024CS 24VDC, 30W					
1	LCU	ATS	Motor power +	VME4SA Servo Amplifier Channel #1	X3_1
2	LCU	ATS	Motor power -	VME4SA Servo Amplifier Channel #1	X3_2
3			Shield (X3-1 - X3-2)		X3_3

TACHOGENERATOR: Minimotor 4.3 G60 4.3mV / rpm					
4	ATS	LCU	Tacho output +	VME4SA Servo Amplifier Channel #1	X3_4
5	ATS	LCU	Tacho output -	VME4SA Servo Amplifier Channel #1	X3_5
6			Shield (X3-4 - X3-5)		X3_6
ENCODER: Minimotor Optical HP HEDL 5540-500 5 VDC - 500 counts/turn - Resolution = 250 counts/degree = 14.4"/count					
7	LCU	ATS	Encoder power +	MAC4-INC Motion Controller Channel #1	X3_16
8	LCU	ATS	Encoder power -	MAC4-INC Motion Controller Channel #1	X3_17
9			Shield (X3-16 - X3-17)		X3_18
10	ATS	LCU	Encoder A+	Differential line driver A+ MAC4-INC Motion Controller Channel #1	X3_7
11	ATS	LCU	Encoder A-	Differential line driver A- MAC4-INC Motion Controller Channel #1	X3_8
12			Shield (X3-7 - X3-8)		X3_9
13	ATS	LCU	Encoder B+	Differential line driver B+ MAC4-INC Motion Controller Channel #1	X3_10
14	ATS	LCU	Encoder B-	Differential line driver B- MAC4-INC Motion Controller Channel #1	X3_11
15			Shield (X3-10 - X3-11)		X3_12
16	ATS	LCU	Encoder Z+	Differential line driver Z+ MAC4-INC Motion Controller Channel #1	X3_13
17	ATS	LCU	Encoder Z-	Differential line driver Z- MAC4-INC Motion Controller Channel #1	X3_14
18			Shield (X3-13 - X3-14)		X3_15
SWITCHES: Micromat KS35A11 24V, 4A max. - Repeatability=0.03 degrees					
19	ATS	LCU	Reference & Negative switch +	MAC4-INC Motion Controller Channel #1	X3_19
20	ATS	LCU	Reference & Negative switch Common	MAC4-INC Motion Controller Channel #1	X3_20
21			Shield (X3-19 - X3-20)		X3_21
22	ATS	LCU	Positive Limit Switch +	MAC4-INC Motion Controller Channel #1	X3_22
23	ATS	LCU	Positive Limit Switch Common	MAC4-INC Motion Controller Channel #1	X3_23
24			Shield (X3-22 - X3-23)		X3_24
25			Global shield (X3-1 - X3-24)		X3_25

Nasmyth Beacon light sources interfaces:

	Item	From	To	Signal	Description	Pin
	IMAGE BEACON LIGHT SOURCES:					
ESO Only	100	LCU	ATS	Switch Image beacon light source #1 on/off	contact closed to switch on image beacon light source #1 Digital Output	TBD
	101	ATS	LCU	Image beacon light source #1 status	contact closed means light source #1 switched on Digital Input	TBD
	102	LCU	ATS	Switch Image beacon light source #2 on/off	contact closed to switch on image beacon light source #2 Digital Output	TBD
	103	ATS	LCU	Image beacon light source #2 status	contact closed means light source #2 switched on Digital Input	TBD

Last modified: Wed Sep 8 17:12:10 METDST 1999

8 APPENDIX D: Package template

8.1 Package: XXX

Description: Description.

Package name: xxx

Inheritance: **NONE.** This is NOT a standard package

Deployment: Control Workstation|LCU: `xxxControl` process.

Use Case diagram:

Insert here the Use Case Diagram

Class diagram:

Insert here the Class Diagram

Architecture:

Reuse from VLT TCS: 90%

Issues to be Determined or Resolved:

Last modified: Tue Apr 18 09:40:31 METDST 2000

Package:

A general purpose mechanism for organizing elements into groups. Packages may be nested within other packages. Both model elements and diagrams may appear in a package.

At implementation level, packages are mapped to VLT software modules, kept under configuration control in the VLT software archive.

A package can be:

- subsystem software
- coordination software
- interface software

Description:

Provide a sentence or short paragraph that clearly defines the purpose or goal of the Package.

Package name:

The package is uniquely identified by a name. The name of the package is used as the prefix of the correspoding VLT software module.

Inheritance:

Inheritance is the mechanism by which more-specific elemtes incorporate the structure and behaviour of more-general elements.

For the current package it shows the list of packages where their structure and behaviour is inherited.

Deployment:

Shows the nodes of the system's hardware topology on which the package is executed. It adresses the static view of the system.

For each package the concerned LCUs and server processes are listed.

Use Case diagram:

Shows the Use Cases that the package is responsible for, and that are completely executed within the package.

Class diagram:

The class diagram shows a set of classes, interfaces, and collaborations and their relationships. It illustrates the static design view of the system.

Each Package consists of a set of classes that are shown in the diagram.

Architecture:

The Package Architecture describes the basic design concepts of the package.

Reuse from VLT TCS:

Gives an estimation of the re-usability of existing VLT TCS code, in the view of needed ATCS code.

Issues to be Determined or Resolved:

List any issues that remain to be decided, or to be reviewed regarding this Package. These issues may include scope of the Package, task description, or user interface.

9 APPENDIX E: Package description

9.1 Package: Nasmyth Focus Device

Description: Control of the Nasmyth Focus related devices, like Nasmyth Wheel and the Nasmyth Beacon.

Package name: nfd

Inheritance: LCU Server Framework <<SubSystem>>`lsfSERVER`

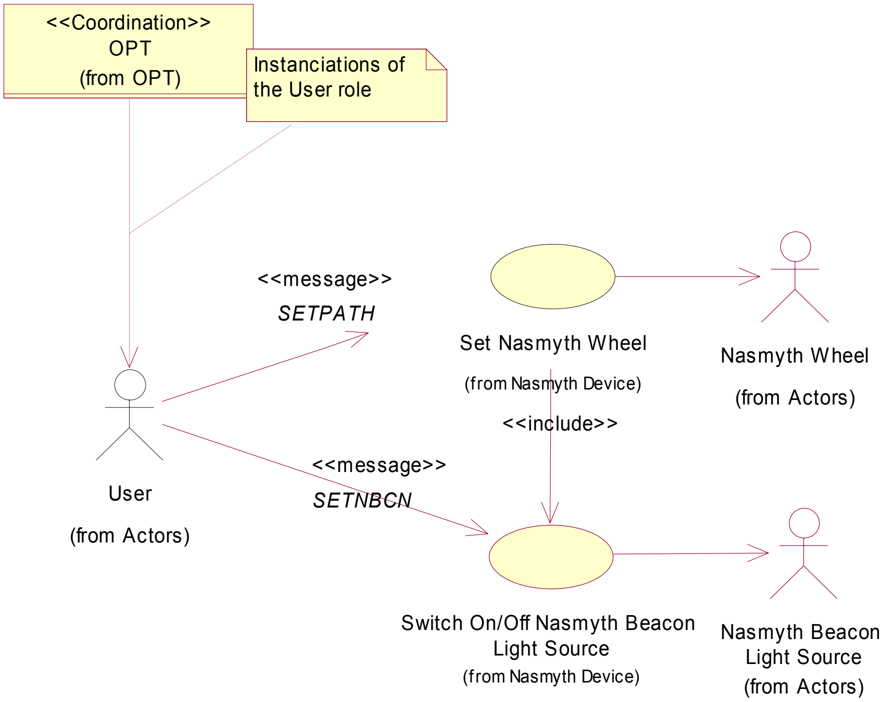
Controlled HW Devices:

- <<Device>>`nfdWHEEL:lsfMOTOR`
- <<Device>>`nfdBEACON:lsfDIGITAL`

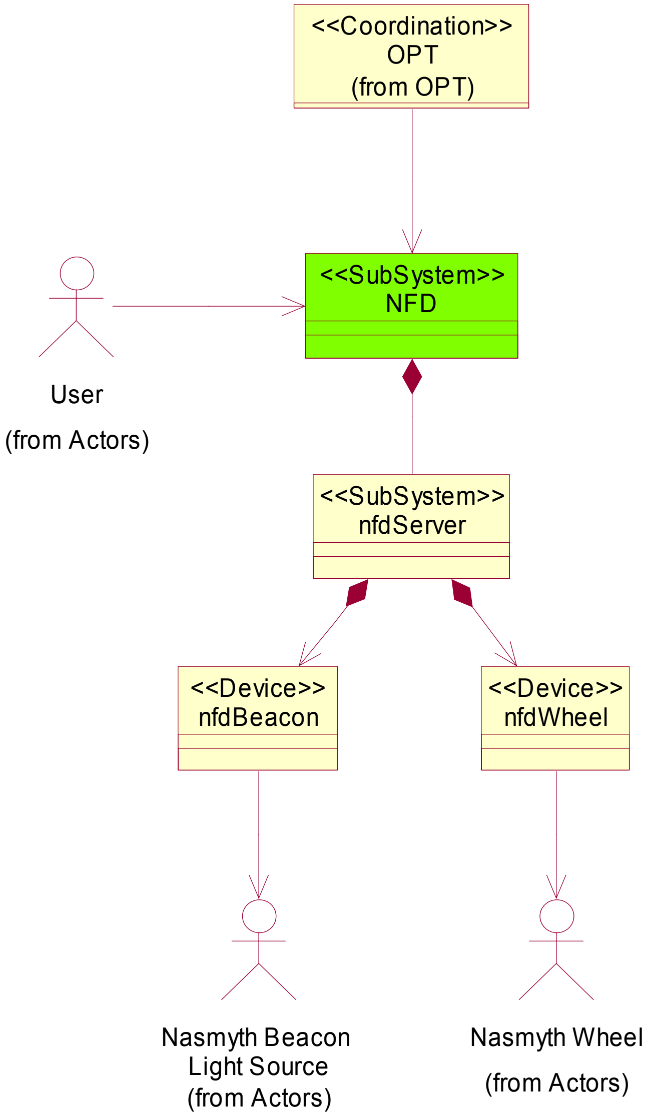
Controlled SW Devices: none

Deployment: Altitude LCU: **`nfdServer`**

Use Case diagram:

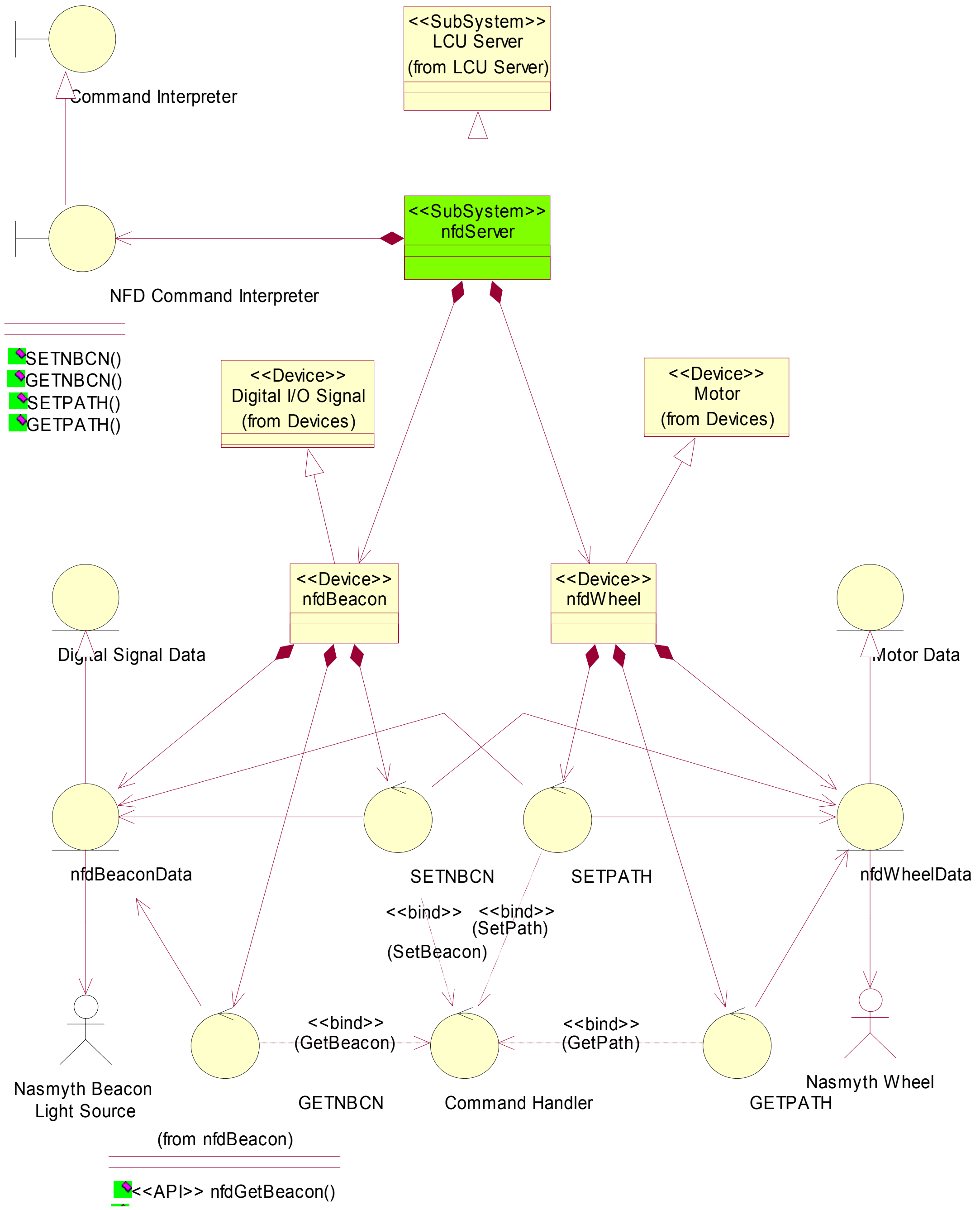


Class diagram:



Architecture:

This Software Device is instantiated from the LCU Server Framework **lsf**.
The sub-class `<<SW Device>>nfdSERVER:lsfSERVER` is composed of an instance of the motor class `<<Device>>nfdWHEEL:lsfMOTOR` and of one instance of the digital signal class `<<Device>>nfdBEACON:lsfDIGITAL`.



The Software Device **nfdServer** provides the following Specific Commands for the control of the Nasmyth wheel:

- **SETPATH** and **GETPATH**: to set and get the wheel position.

and one Specific Command for the control of the 2 beacons:

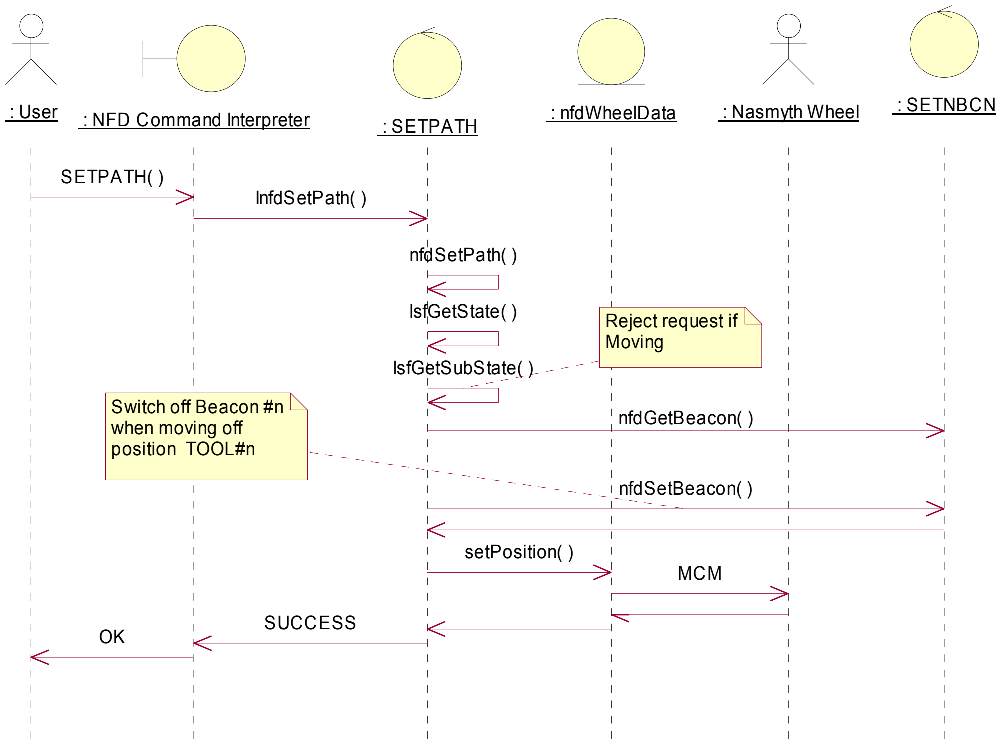
- **SETNBCN**: switch On and Off the beacons.

Command SETPATH / GETPATH

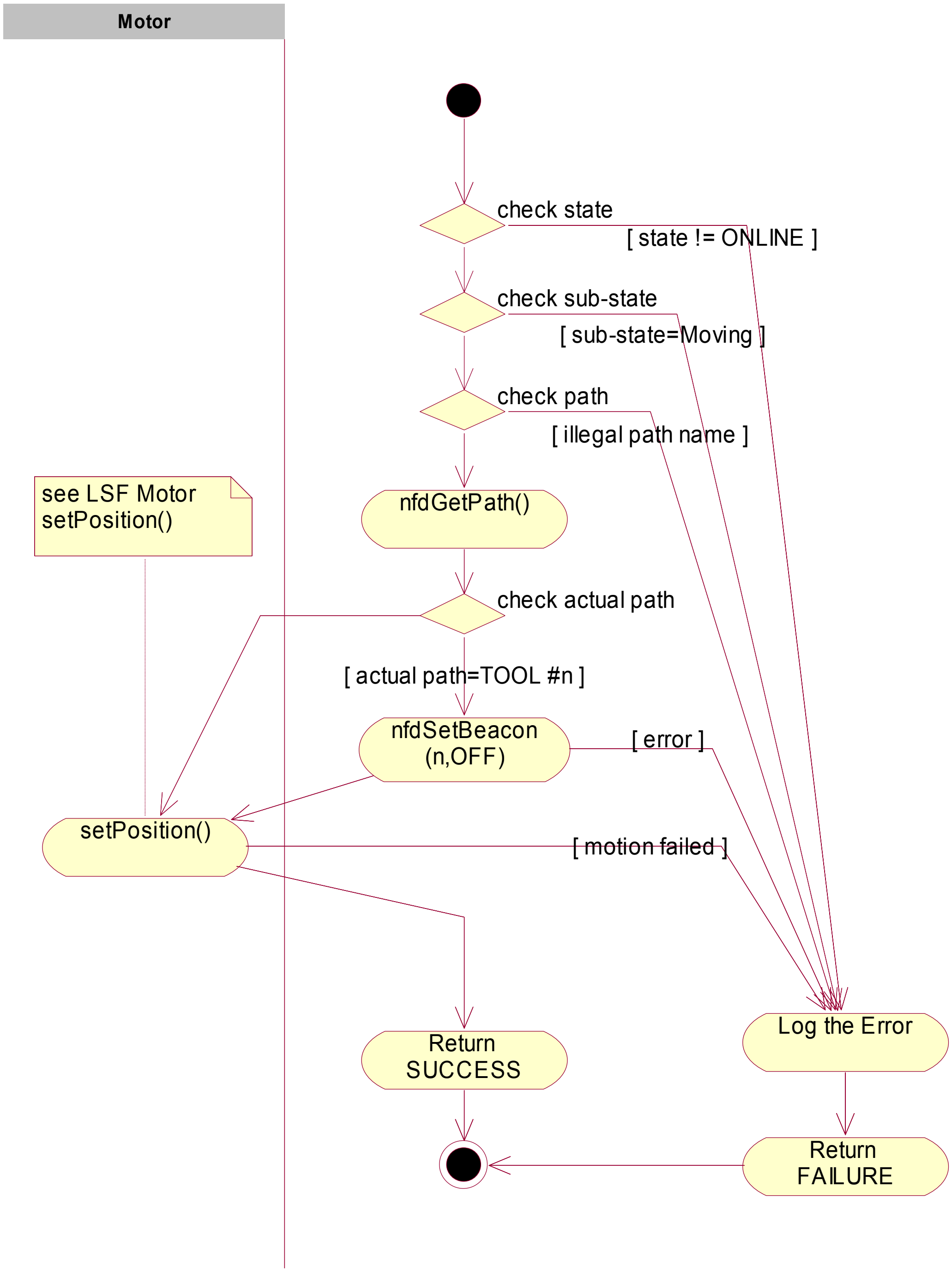
These commands are transient instances resp. of the classes: `<<Control>>nfdSETPATH:lsfCOMMAND_HANDLER` and `<<Control>>nfdGETPATH:lsfCOMMAND_HANDLER`.

They invoke the inherited method `lsfMOTOR::setPosition(char *name[], vltDOUBLE offset[])` and `lsfMOTOR::getPosition()`:

Sequence diagram:



Activity diagram:



The Nasmyth wheel is intended to be used mainly for optical alignments, thus only the position mode is relevant here. The positions are accessible via their names (see MCM Named Positions) to which an offset may be added as required (only for both alignment tools):

- `FREE`: Free hole.
- `RETRO`: Flat retro-reflecting mirror.
- `TOOL1`: Alignment tool #1.

- **TOOL2**: Alignment tool #2.

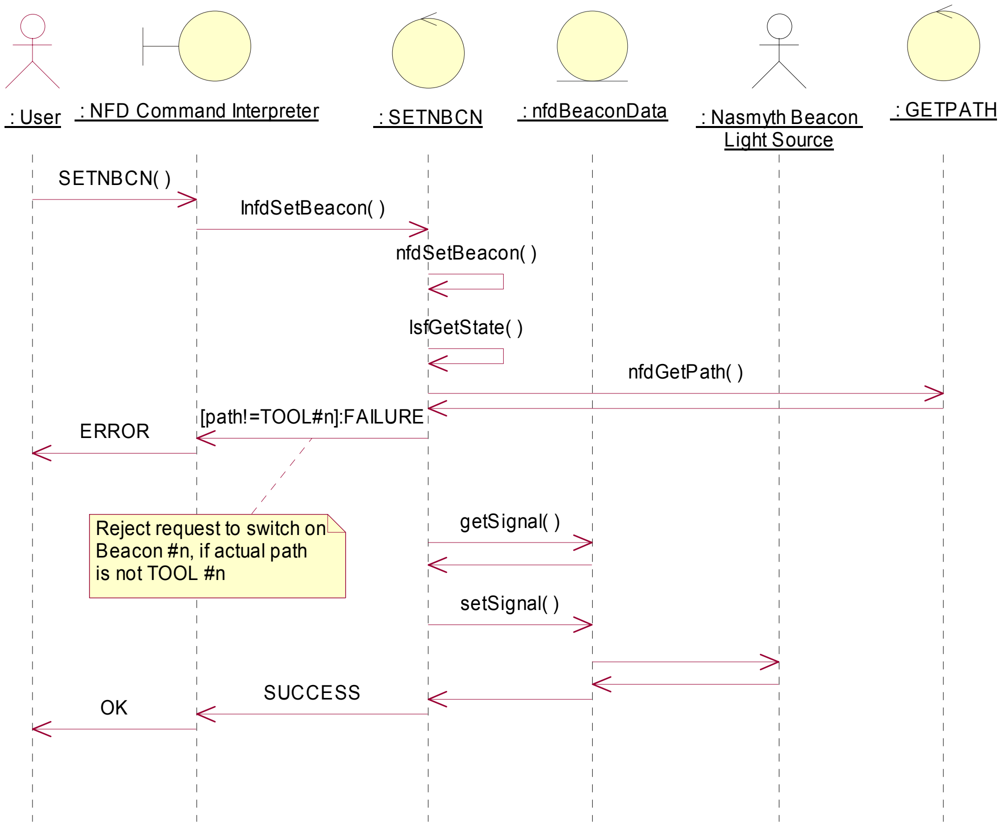
The default (observation) position is **FREE**.

Command **SETNBCN**

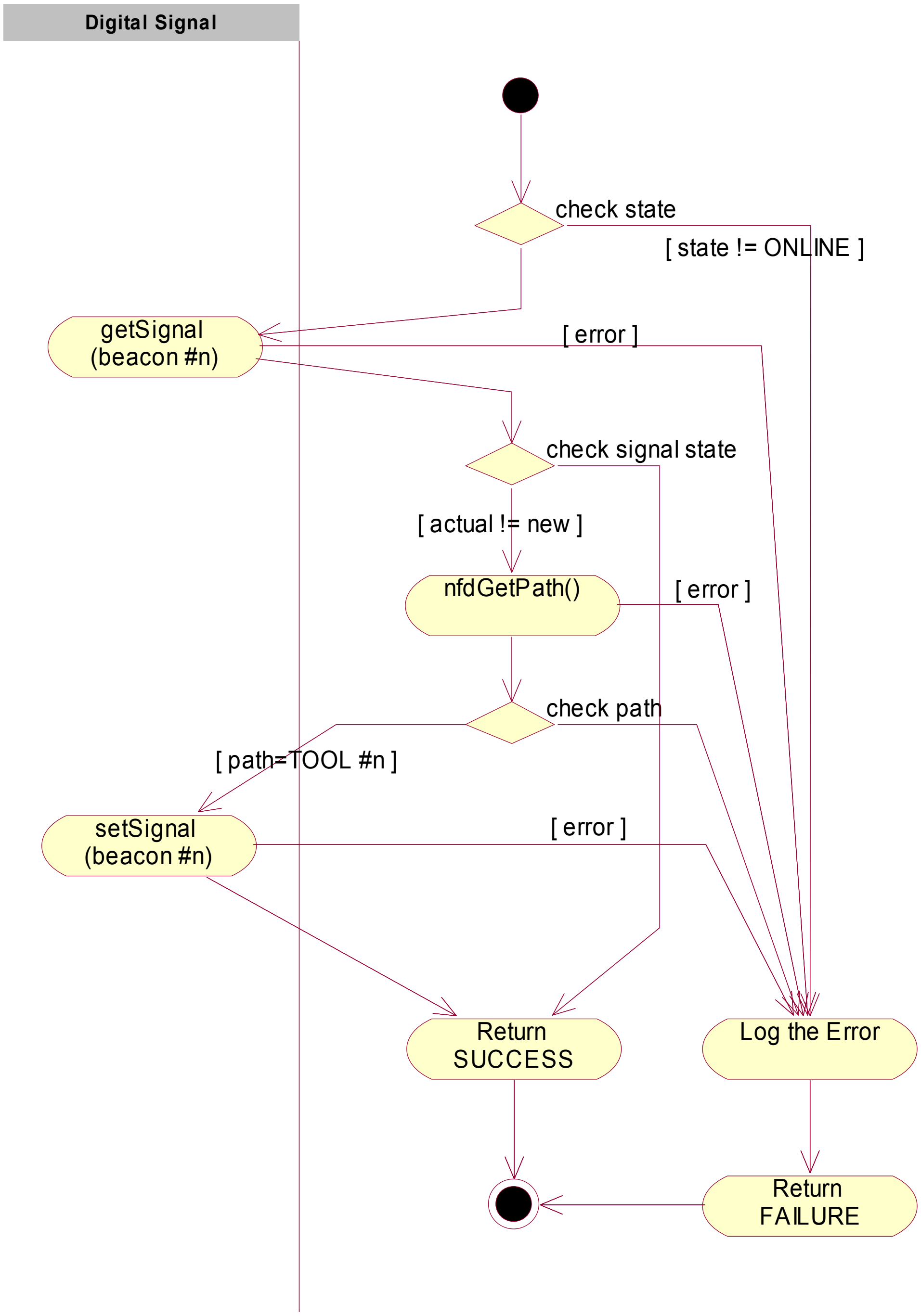
This command is a transient instance of the class `<<Control>>nfdSETNBCN:lsfCOMMAND_HANDLER`.

This command invokes the inherited methods `lsfDIGITAL::setSignal()` and `lsfDIGITAL::getSignal()`:

Sequence diagram:



Activity diagram:



Two beacons are available that can be controlled individually giving their number associated to the actual Nasmyth Wheel position (`TOOL1` or `TOOL2`). None of the beacons shall be switched on when the wheel is not in one of these 2 positions. The beacons are switched off automatically whenever the wheel is moved to the positions **FREE** or **RETRO**. The default (observation) signal states are **OFF**.

Command Definition Table: nfdServer.cdt

Command Interpreter Table: nfdServer.cit

Reuse from VLT TCS: New package

Issues to be Determined or Resolved: None

Last modified: Fri Mar 31 13:18:46 METDST 2000

10 APPENDIX F - Checklists

Checklists[10] are used to guide the examination of the diagrams for syntax problems.
Hereafter are checklists defined that contain items specific to s specific development phase.

UML Detailed Design Checklist		
Question	Yes	No
Analysis-to-design model issues		
Are all classes in the analysis Model that are not in the design model outside the scope of the application?		
Are all states in the analysis model statecharts also states in the statechart diagrams in the design model?		
Are the sequences of messages in all design-level sequence diagrams the same, even though additional messages may have been between the analysis-level messages?		
Internal design model issues		
Are all associations shown with no navigation information truly bi-directional?		
Are all composition relationships shown as unidirectional?		
Is every sequence diagram a subset of some activity diagram?		
Does every message sent in an interaction diagram appear as a method in the public interface class of the receiving object?		
Are the transitions out of a state in the state diagram mutually exclusive?		
Do all state machines, except for perpetual objects, contain initial and final states?		
Are all public modifier methods represented as transitions on each state even if they only result in a self-back loop?		
Is there a sequence diagram for each postcondition clause of each method that corresponds to use cases that meet the frequency/criticality threshold?		
Are all messages shown correctly as synchronous or asynchronous?		
Do they number of forks and joins balance in every activity diagram?		

Testability Checklist Items		
Question	Yes	No
Are all use cases written in sufficiently specific language to support the writing of test cases?		
Do all of the methods in the domain model have complete signatures?		

11 APPENDIX G - Frequently asked Questions

This section answers frequently asked questions related to the use of UML, ROSE or RUP that are not covered by literature.

1. **How should I use the component view to map the logical view**?
   A component does realise a class, but the meaning of realise is context sensitive. Also a task is a component that is an independent thread of control. A source code file is a good realisation of a class but a task (thread of control) is not a good realisation of a class. (Rose does allow this mapping).
   What you should model is:
   Class X is realised by component X.h and X.cpp. Component (task) XYZ.ocx is dependent on X.cpp Y.cpp and Z.cpp. Component ABC.exe is dependent on XYZ.ocx..
   Now component ABC may have many independent tasks (threads of control, FORKs) some of which could be instances of XYZ. Modeling of this should be done in deployment diagrams (not supported by rose).

2. **How to represent iterations (loops) in sequence diagrams**
   An asterisk (*) before the operation signature represents an unspecified number of iterations. An asterisk followed by a condition such as *[i:=1..n], represents a variable and specified number of iterations.
   If you have several operations at the same level of nesting, some of which are part of a loop, this loop can be represented by a constraint, as shown in Figure 6.

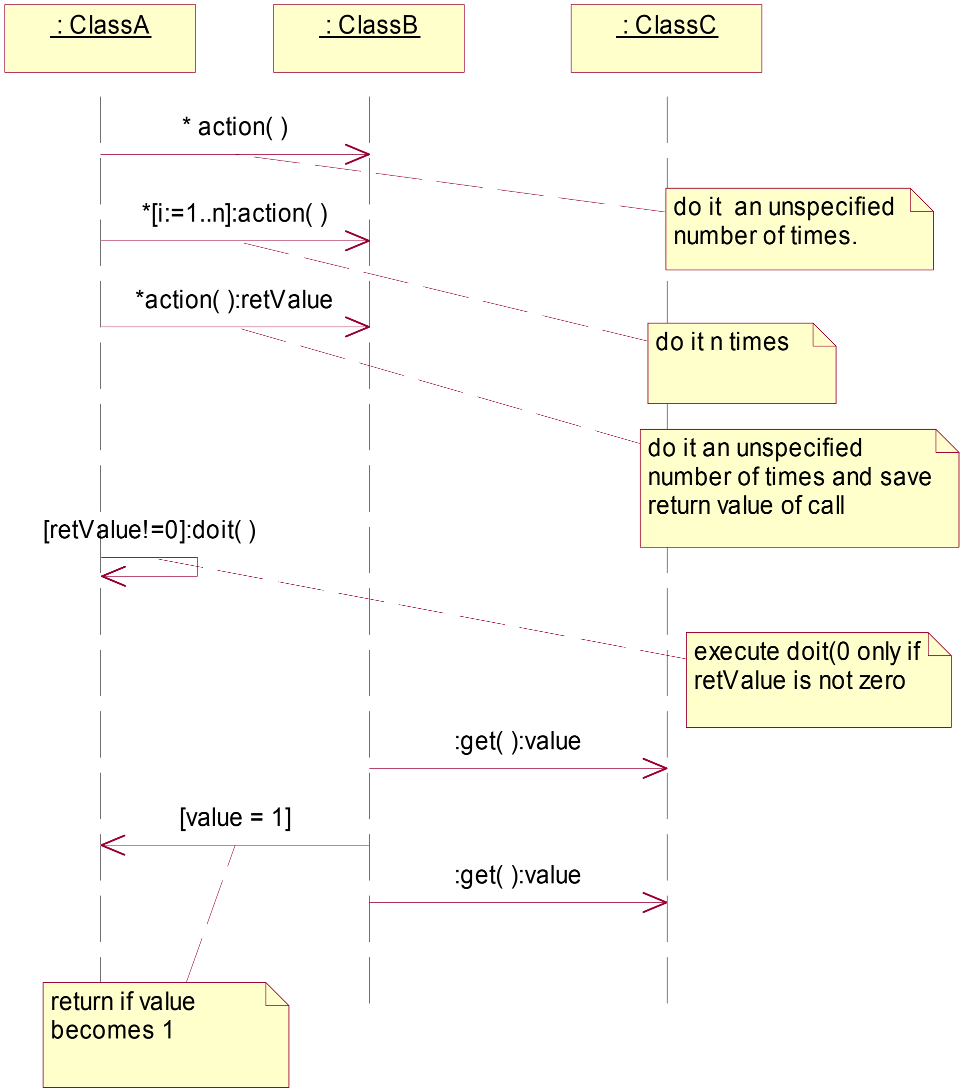

: ClassA
: ClassB
: ClassC
* action( )
*[i:=1..n]:action( )
*action( ):retValue
do it an unspecified number of times.
do it n times
do it an unspecified number of times and save return value of call
[retValue!=0]:doit( )
execute doit(0 only if retValue is not zero
:get( ):value
[value = 1]
:get( ):value
return if value becomes 1

Figure 6 - constraints in sequence diagrams

3. **How can I represent Packages in class and sequence diagrams?**
   Since Rose does not support associating classes and packages, create in each Package a class with the name of the package. Use this class as a representative of the package in other diagrams.

___oOo___