VLTI Auxiliary telescopes: a full Object Oriented approach

Gianluca Chiozzi[*], Philippe Duhoux, Robert Karban

European Southern Observatory, Munich, Germany

ABSTRACT

The Very Large Telescope (VLT) Telescope Control Software (TCS) is a portable system. It is now in use or will be used in a whole family of ESO telescopes VLT Unit Telescopes, VLTI Auxiliary Telescopes, NTT, La Silla 3.6, VLT Survey Telescope and Astronomical Site Monitors in Paranal and La Silla). Although it has been developed making extensive usage of Object Oriented (OO) methodologies, the overall development process chosen at the beginning of the project used traditional methods. In order to warranty a longer lifetime to the system (improving documentation and maintainability) and to prepare for future projects, we have introduced a full OO process. We have taken as a basis the Unified Software Development Process with the Unified Modelling Language (UML) and we have adapted the process to our specific needs. This paper describes how the process has been applied to the VLTI Auxiliary Telescopes[1] Control Software (ATCS). The ATCS is based on the portable VLT TCS, but some subsystems are new or have specific characteristics. The complete process has been applied to the new subsystems, while reused code has been integrated in the UML models. We have used the ATCS on one side to tune the process and train the team members and on the other side to provide a UML and WWW based documentation for the portable VLT TCS.

Keywords: VLT, VLTI, Telescope Control Software, Object Oriented, UML.

1. INTRODUCTION

At the beginning of 1999, the first VLT Unit Telescope started its scientific activity after one year of commissioning. The Telescope Control Software developed for the VLT had already been used also on a number of other ESO Telescopes and some new ESO projects had decided to use the same core TCS. We can now say that the VLT TCS is a highly portable system, used in a whole family of ESO telescopes: VLT UTs, VLTI Auxiliary Telescopes, NTT, La Silla 3.6, VLT Survey Telescope and the Astronomical Site Monitors in Paranal and La Silla.

That was the right time to start evaluating the overall software development chosen at the beginning of the VLT project, about 8 years ago, in order to identify its weaknesses and its strong points.

The objective of this activity was to identify an improved development process to be applied to new projects and to retrofit whenever possible and convenient in the VLT Project with the new methodology. In order to warranty a longer lifetime to the system (improving documentation and maintainability), we decided to introduce a fully Object Oriented development process. We also decided to take as a basis the Unified Software Development Process[1] with the Unified Modelling Language (UML)[3] and to adapt the process to our specific needs.

2. THE SOFTWARE PROCESS FOR THE VLT CONTROL SOFTWARE

The Very Large Telescope (VLT) Control Software has been developed making extensive usage of Object Oriented design and development methodologies, but the overall software process chosen at the beginning of the VLT project [5] and based on IEEE 828 standards (see [5] for references) was not Object Oriented.

The success of the VLT Project proves that the choices made and the methodology used were good. In the 8 years we have been working on this project, the Software Process has shown many strong points [6].

Nevertheless a number of weak points have been identified:

- **Requirements capture/trace.** The process does not provide good support for discussing conveniently the requirements with the customers of the system. We need also a way of expressing requirements in a format

[*] Correspondence: Email: gchiozzi@eso.org; WWW: http://www.eso.org/~gchiozzi; Telephone: +49-89-32006 543

comprehensible and formal enough to be used for tracing them during the whole software process and to produce the final acceptance tests. It also does not properly support requirement changes during the evolution of the project.

- **Interface tracing**. One of the major problems we have had in the project is the control of the interfaces between the various components developed by different groups and external contractors and consortia. The ICD documents could not be directly traced to the implementation of interfacing products, causing problems in the integration phase.
- **Documentation consistency.** During the process, paper deliverables are produced in each phase (requirements, preliminary design, detailed design and so on). The single documents are very useful for the review of each phase, but the way they are structured and written makes it too resource demanding to keep all of them up to date when moving from one phase to the next. As a consequence, after a few iterations, requirement, design documents and implementation become inconsistent.

Some of these weaknesses become critical due to the specific parameters of 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.

Here are some important aspects to consider:

- A scientific experiment is always at the limits of knowledge and technology. As a consequence requirements are often unclear and change in 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 it comes to final system integration, 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.
- Interfaces with the hardware and the electronics have a very big influence on the system and we have to spend a big effort in making them clear and stable.

3. THE PILOT PROJECT: VLTI AUXILIARY TELESCOPE CONTROL SOFTWARE

We have decided to select the ATCS as pilot project for the introduction of a new methodology, able to solve or mitigate the problems described in the previous section.

The adoption of a new design methodology requires:

- Tuning of the methodology itself, to adapt it to the needs of the specific domain in which it will be used.
- Training of the development team

In order to reach these two key objectives, the right project has to be selected: the ATCS project satisfies all the necessary requirements:

- **It is a complete system.** We can exercise the methodology on a real scale project and get a realistic evaluation of the results.
- **It is a complete project (analysis, design, development and deployment).** We can exercise the whole software life cycle, from the requirements phase to the delivery and maintenance of the final product.
- **It is an well-understood system.** It is nevertheless just another telescope where we will have to run a Telescope Control Software very similar to the one running on the VLT. We can then concentrate on the problems related to the methodology without being "distracted" by the problems due to the comprehension of the system to be designed.
- **There is limited pressure/risk.** Because of the good knowledge of the system to be developed and thanks to the availability of most of the software needed for the implementation, we have no high pressure on the project and we can spend the necessary time discussing issues related to methodology.

The ATCS is based on the portable VLT TCS, but some subsystems are new or have specific characteristics. The complete process has been applied to the new subsystems, while reused components have been integrated in the UML models and partially reverse-engineered.

As a by-product we have also obtained a more general UML and Web based documentation for the portable VLT TCS.

4. SOFTWARE PROCESS OVERVIEW

A software development process describes all the activities that are necessary to transform *user requirem*ents into a final software system. Adopting a structured process is essential to be able to keep the development under control, i.e. to be able to evaluate and measure at every step the amount and the quality of the work done and to assess the amount of remaining work.

The software process we have adopted is based on the Unified Software Development Process, which is described in details in the reference book [1]. The essential aspects of this process are captured, as the authors say, by the following three cornerstones:

- Use Case driven
- Architecture centric
- Iterative and incremental

4.1 Use Case driven process

Use Cases are a way to describe the interaction between the system and its users. The set of all Use Cases makes up the Use Case model, which describes the complete functionality of the system.

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.

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

4.2 Architecture Centric Process

The architecture of a software system identifies the most important static and dynamic aspects of the system itself and provides a common vision that all developers and customers must agree on or at least accept.

Through different views of the system being built, the architecture shows how the system will allow the realisation 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 a view of the whole design with the important aspects put in evidence by leaving aside the details.

The architecture is the starting point to identify the *shape* 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 specified in detail and mature.

4.3 Iterative and incremental process

The work necessary to build the final system is divided into smaller slices. The driving concept consists in the idea that each 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 realisation 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.

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

4.4 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***. Each *cycle* consist of four ***phases:***

- Inception
- Elaboration
- Construction
- Transition

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***. An *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
- Analysis
- Design
- Implementation
- Test

Use Cases play a key role in any of the four *phases*:

- In the *Inception Phase*, high level Use Cases are developed to identify what is in the scope of the project.
- In the *Elaboration Phase*, 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*.
- In the *Construction Phase*, 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*.
- In the *Transition Phase*, Use Cases are the core items of the Acceptance Test. They are also used to develop user guides and for training.

4.5 Project milestones

The official project milestones are extremely important, since they make available to managers and customers 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 we consider essential that milestones are not moved forward in time in case of delays in the project, but that rather the scope of the milestone is clearly and officially restricted. This approach helps in better identifying and manage project problems and allows to decide and timely undertake corrective actions. We have adopted this approach in the VLT project and we consider it a key factor for its success.

Since we consider the first phases of the project extremely important, we have introduced an “extra” official milestone at the end of the first iteration in the Inception phase. The following table shows the milestones associated with each phase:

Phase	Milestone
Inception (first)	Software Requirements Review
Inception	Preliminary Design Review
Elaboration	Final Design Review
Construction	System Delivery
Transition	Final Acceptance Test

In addition to these official milestones, that mark the end of each process phase, we consider equally important to have periodic milestones, that we call *releases*, at the end of each iteration with a typical period of six months for a project like the VLT control software. Also the release dates have to be fixed points, for the reasons already mentioned. At each release an updated version of all deliverables is internally distributed to all partners in the project. This is particularly relevant for big distributed projects, since it allows a synchronisation between all partners.

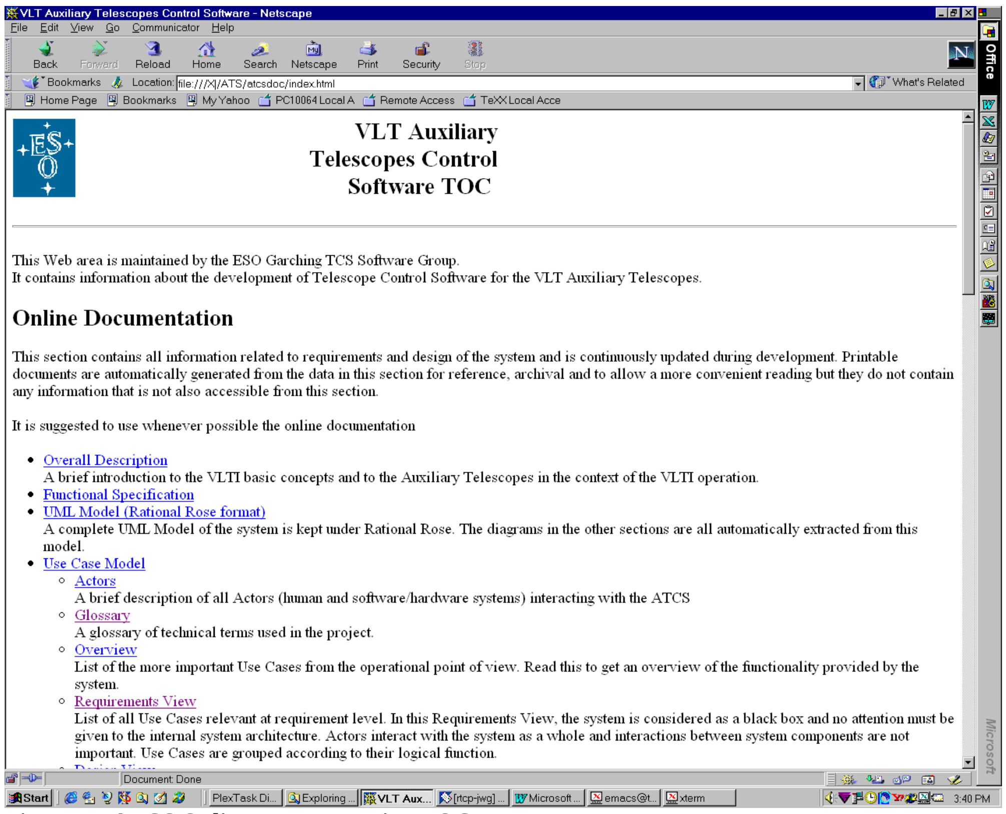
4.6 "One Document" approach

We have learned from the experience of our previous projects that keeping the documentation up to date and readily available to all the interested parties is a major problem.

We have then put as our objective that of having *one document*[4]. This means that:

- All the relevant documentation of the project is kept in one single, centralised, repository
- No information is 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.

The ATCS Online Documentation contains at any time all the documentation available for the ATCS project and all the documentation officially released at major milestones. It is kept under configuration control, so that it is possible also to analyse the evolution of the project with time. Automatic consistency checks warranty the coherence of the information provided.


VLT Auxiliary Telescopes Control Software - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: file:///X|/ATS/atcsdoc/index.html What's Related
Home Page Bookmarks My Yahoo PC10064 Local A Remote Access TeXX Local Acce
Office
Microsoft
Document: Done
Start PlexTask Di... Exploring ... VLT Aux... [rtcp-jwg] ... Microsoft ... emacs@t... xterm 3:40 PM

VLT Auxiliary
Telescopes Control
Software TOC

This Web area is maintained by the ESO Garching TCS Software Group.
It contains information about the development of Telescope Control Software for the VLT Auxiliary Telescopes.

Online Documentation

This section contains all information related to requirements and design of the system and is continuously updated during development. Printable documents are automatically generated from the data in this section for reference, archival and to allow a more convenient reading but they do not contain any information that is not also accessible from this section.

It is suggested to use whenever possible the online documentation

- Overall Description
  A brief introduction to the VLTI basic concepts and to the Auxiliary Telescopes in the context of the VLTI operation.
- Functional Specification
- UML Model (Rational Rose format)
  A complete UML Model of the system is kept under Rational Rose. The diagrams in the other sections are all automatically extracted from this model.
- Use Case Model
  - Actors
    A brief description of all Actors (human and software/hardware systems) interacting with the ATCS
  - Glossary
    A glossary of technical terms used in the project.
  - Overview
    List of the more important Use Cases from the operational point of view. Read this to get an overview of the functionality provided by the system.
  - Requirements View
    List of all Use Cases relevant at requirement level. In this Requirements View, the system is considered as a black box and no attention must be given to the internal system architecture. Actors interact with the system as a whole and interactions between system components are not important. Use Cases are grouped according to their logical function.

Figure 1 - ATCS Online Documentation TOC

The most difficult task in designing the documentation architecture is to make the best usage of the Web technology to allow at the same time the readers:

- To get an overview of the project
- To navigate down to the deepest details, ideally down to implementation source code level

We are also well aware that online documentation alone is not enough. Printed documents are essential not only for reviewers, but also for members of the development team.

Web and printed paper have different characteristics and different strong and weak points that have to be exploited. The online documentation favours navigability, while printed documentation favours sequential reading. In order to exploit these characteristics, information must be kept as atomic as possible. The elementary information units can then be used as building blocks and inserted in very different documents.

In this area we have seen that the support of good tools and efficient procedures is extremely important. They shall intervene at various places to automate operations that would otherwise be too time consuming,, highly repetitive or error prone. Tools are also required to keep the update costs low, in particular with regard to the generation of the documentation. This aspect is critical in the life cycle of the project in order to overcome the risk of documentation inconsistency. At last, tools are necessary wherever the amount of information to process is large: check of model consistency, requirement traceability etc...

5. 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. For this reason we have invested a significant amount of time in understanding how they have to be written, in defining a template to be used and in analysing how they have to evolve and be used in the different phases of the process.

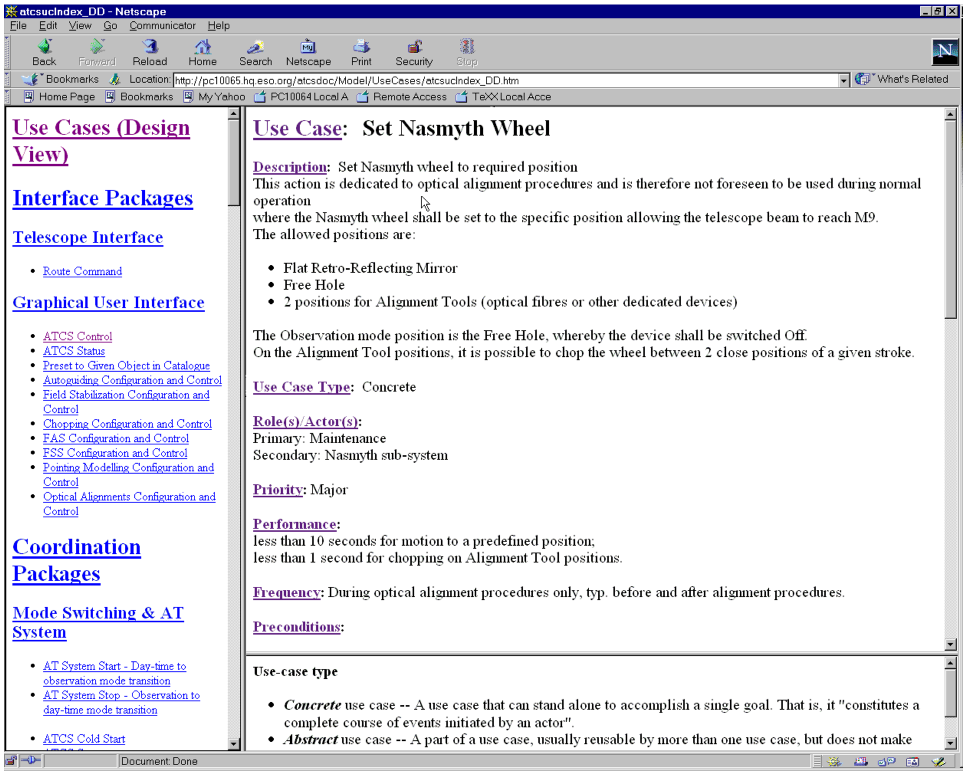
We have verified that they are a very effective way of communicating with customers and among team members. Before each discussion we always provide to the partners the set of relevant Use Cases. During meetings, they stimulate focused discussions and help identifying important details.

5.1 Use Cases: scope and target readers

In the tuning of the development process, we have spent a lot of time discussing on the scope of the Use Cases and on the target readers. We have concluded that the best choice between the various alternatives depends on the type of project and on the structure of the organisation that implements it.

In our case we have to take into account the following constraints:

- We develop control systems, i.e. software that directly controls custom hardware and electronics
- Our main contact points during the project phases are the engineers responsible for the project and the engineers of the other development teams (hardware, electronics, external contractors).
- Our "users" are in most cases other software systems or human operators with a high degree of specialisation and with a good knowledge of the architecture of the system.


atcsucIndex_DD - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: http://pc10065.hq.eso.org/atcsdoc/Model/UseCases/atcsucIndex_DD.htm What's Related
Home Page Bookmarks My Yahoo PC10064 Local A Remote Access TeXX Local Acce
Document: Done

Use Cases (Design View)

Interface Packages

Telescope Interface

- Route Command

Graphical User Interface

- ATCS Control
- ATCS Status
- Preset to Given Object in Catalogue
- Autoguiding Configuration and Control
- Field Stabilization Configuration and Control
- Chopping Configuration and Control
- FAS Configuration and Control
- FSS Configuration and Control
- Pointing Modelling Configuration and Control
- Optical Alignments Configuration and Control

Coordination Packages

Mode Switching & AT System

- AT System Start - Day-time to observation mode transition
- AT System Stop - Observation to day-time mode transition

- ATCS Cold Start

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:

Use-case type

- ***Concrete*** use case -- A use case that can stand alone to accomplish a single goal. That is, it "constitutes a complete course of events initiated by an actor".
- ***Abstract*** use case -- A part of a use case, usually reusable by more than one use case, but does not make

Figure 2 - Use Case frame with table of contents, Use Case text and help

- Graphical user interfaces are not a primary focus of our development. For our systems, user interfaces are an independent layer on top of the application.

Taking these aspects into account,

- We have NOT put in our Use Case model very high level Use Cases like "Perform an observation". They are responsibility of the Data Flow development groups, while we have just to provide the building blocks that they are going to use.
- We have developed "command based" Use Cases, i.e. our Use Cases describe atomic functions that can be assimilated to commands sent by Actors to our system.
- We have specified with great detail the interfaces with the external software, electronic and hardware subsystems. These interfaces must be agreed with other engineering groups and are essential for the successful integration of the whole system.

These decisions make our Use Cases rather "technical" and they may be difficult to read and understand for non-technical people, like for example astronomers. We have explicitly made this choice, but in other contexts and for other projects a different choice (for example more similar to the description of graphical user interface interactions) may be better.

5.2 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 recognisable and common layout; relevant information is easier to find and communication is optimised.

The template we have defined is based on the very good papers of L.Mattingly[7] and A.Cockburn[8] and on the book entirely dedicated to Use Cases of G.Schneider[9].

We do not go here into details on the Use Case template, 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, since 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 customers.

5.3 Use Case diagrams

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

These diagrams are important to give and 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 customers, that are often non technical people.

For this reason we have on one side created the Use Case Diagrams and on the other side created HTML tables of contents of the Use Cases, grouped according to different criteria. We have also implemented all the relations that appear in the Use Case Diagrams as hyperlinks in the online version of the Use Cases.

6. INCEPTION

6.1 First Iteration in Inception: from Kick-Off meeting to Software Requirements Review

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 customers and the development team on the requirements for the system and to demonstrate to the customers that the development team has a good understanding of the agreed requirements.

This milestone is not part of the Unified Software Development Process, but we consider it very important 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.

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 customers and team members and to avoid misunderstanding.
- **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 glossary with 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, for example, of the Auxiliary Telescope and that have to be controlled by our Control.
- **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 customers.** Writing the Use Cases requires technical knowledge and a good insight on the software development process. It is not correct to ask the customers to write them. When Use Cases have been inferred from the available initial documentation they are discussed with the customers and accordingly modified. The language used to write the Use Cases is simple enough for the customers 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.
- **Risk assessment.** A first basic analysis of the areas of major risk in the project.

6.2 Inception leads to Preliminary Design Review

In the rest of 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.

During the whole Inception Phase, the team works on the artifacts already described, adding new details, and on new items. Among these, packages and interfaces are the most relevant.

6.2.1 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[3]. Packages should have[9]:

- A single functionality
- Strong internal cohesion
- Loose external coupling

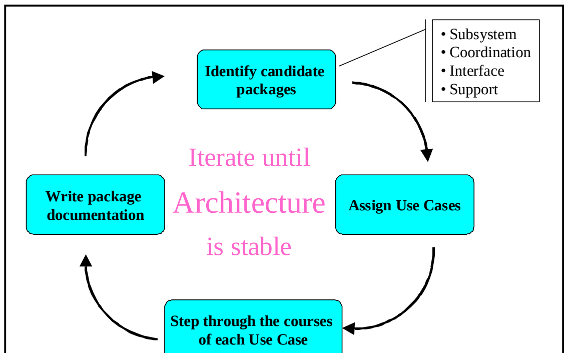

Identify candidate packages
• Subsystem
• Coordination
• Interface
• Support
Write package documentation
Iterate until Architecture is stable
Assign Use Cases
Step through the courses of each Use Case

Figure 3 - Identify packages

- Minimal communication to other packages

We have discussed between two alternative approaches for designing the internal architecture:

1. Start identifying the classes in the system and later on group them in packages and distribute them on the processing nodes.
2. First identify the packages and distribute them on the processing nodes, then go in design details package by package to identify the classes.

We have decided to go for the second one, since we think it maps better with our preconditions:

- 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.

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 on the following four steps until the architecture is stable:

1. Identify candidate packages putting them in one of the following 4 categories:
   - **Subsystem packages.** There is a subsystem package per each 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.
   - **Co-ordination packages.** Are hierarchically above subsystem packages. They are responsible for actions of a combined co-ordination nature, and they often use one or more subsystem to execute their actions.
   - **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 each package is just represented by a class. It allows to represent 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.

6.2.2 Interfaces

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

- Create an ICD (Interface Control Document) section in the Online Documentation. There must be one ICD sub-section per each Actor.
- Each ICD is subdivided in Interfaces, where each 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 the interfaces. Clearly, the level of definition of the Operations depends on the stage of analysis, design and implementation.

6.2.3 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 behavioural diagrams can be used[1][3]:

- Activity diagrams
- Interaction (Collaboration and Sequence) Diagrams
- State Diagrams

7. ELABORATION

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 customers officially accept the proposed architecture. On the other side, the organisation 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.

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 analyse 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.

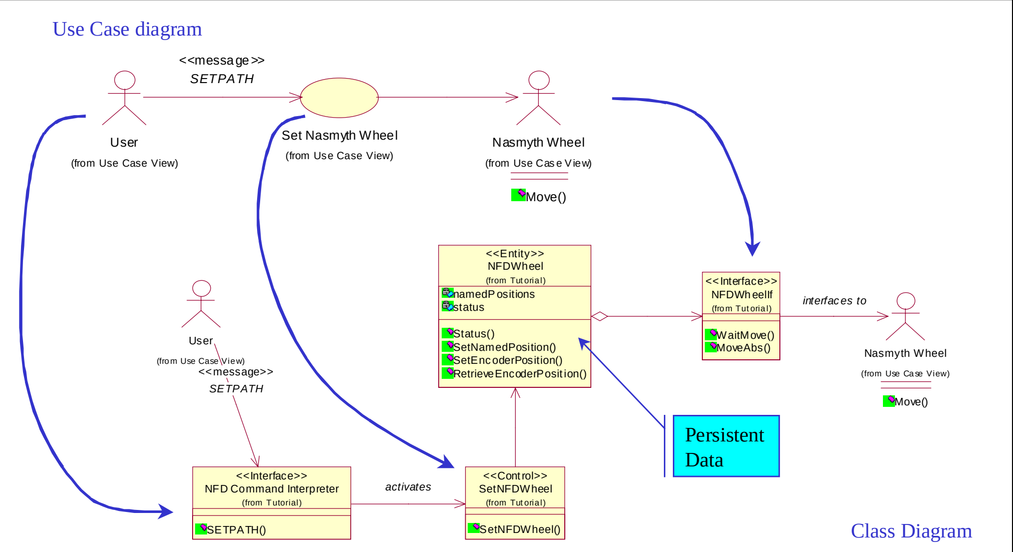


Figure 4 - Identify package architecture

For the ATCS Project, as before for the VLT TCS, we have built a prototype as complete as possible in terms of control electronics and including a number of hardware components considered very critical. This prototype is 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[1] to build the architecture of the packages. 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 each package there is typically one boundary class per each 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 Other Entity Classes can be inferred from the problem domain or from the original requirement documents. In any case, 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 co-ordination, sequencing, transactions and control of other objects. The dynamics of the system are modelled by control classes. There is typically a control class per each Use Case, although simple Use Cases may not need a control class.

8. CONSTRUCTION

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. In ESO terms, this corresponds to the "European Readiness Review", i.e. the system before installation and beginning of in-situ test and integration.

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.** It is essential that interfaces with Actors and between co-operating packages be defined in details. Every package must have its own package test (modular test, in VLT terminology).
- **Package implementation means implementation of the Use Cases (or scenarios) 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. External packages are replaced by stubs. 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 development process

9. TRANSITION

The purpose of the *Transition Phase* is to make the system ready for unrestricted release to the user community and a **Final Acceptance Test** is set at the end of the *Transition Phase*. In ESO terms, it corresponds to the end of the commissioning phase for the VLT.

We are very confident in the methodology and tools used for the VLT Commissioning [10], and we intend therefore to adopt them also for the new process.

The only area that needs changes to be properly integrated in the Use Case driven process is the definition of the Test Cases. We need now to extract Test Cases for final acceptance directly 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 detailed a check list. The tools developed for this purpose in the VLT project are very powerful.

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

10. CONCLUSIONS

After one year of work, we have completed Elaboration and we are in the Construction phase for our pilot project. In this time we have accumulated a big experience on the development process and on UML and we can draw first conclusions.

We have dedicated a lot of time in tuning the process and adapting it to our specific environment, but we are very satisfied of the results and we have collected very good feedback from people involved in other aspects of the project.

For the analysis and the design of the system, we can definitely say that it is a big improvement over the previous process and that it mitigates or overcome the weaknesses identified in the process used for the development of the VLT Telescope Control Software (like requirement tracing, documentation, maintenance and support for object orientation). 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.

Next to come is a deeper understanding of the Construction Phase, with the focus on getting consistency and traceability of requirements and design in the actual code. On one hand, we doubt that we can make this tracing process automatic and transparent. On the other hand, the process used for the VLT has proved itself very strong in the development, testing and deployment activities and we want to keep most of that positive experience.

Before the process can be applied on a large scale within ESO, it needs to be formalised in a way that new groups can adopt it. We are now working to document in detail how the Unified Development Process has been tailored to our specific application domain. We need also to enhance the set of dedicated support tools. The acceptance of a new development process is highly correlated with the quality of the framework to support it. It must be robust and attractive; in addition, little room shall be left in order to reinforce its internal coherency.

We have anyway registered a great interest inside our organisation for this work and other projects have already started using our methodology and capturing their requirements by means of Use Cases.

We want also to reaffirm that open discussion and co-operation among team members have proven 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.

11. ACKNOWLEDGEMENTS

This work is the result of many hours of discussions, trials and test inside our Software Group. We want to thank here all our colleagues for the important contribution to the definition of the SW Development Process for our projects.

12. REFERENCES

1. B. Koehler, C. Flebus, "VLTI Auxiliary Telescopes", Interferometry in Optical Astronomy, SPIE Vol. 4006, paper 03, Garching 2000.
2. I.Jacobson, G.Booch, J.Rumbaugh, *The Unified Software Development Process*, Addison Wesley, Reading MA,1998
3. G.Booch, J.Rumbaugh, I.Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, Reading MA,1998
4. G.Chiozzi. J.M.Filgueira, "Real-Time Control Systems: a "One Document" Object Oriented Development Process, *Proc. ICALEPCS '99*, Trieste, Italy, Oct. 1999
5. VLT-PLA-ESO-00000-0006, "VLT Programme, Software Management Plan", Issue 2.0, ESO, Garching, Germany, 1992
6. F. Carbognani, G. Filippi, "Software practices used in the ESO Very Large Telescope Control Software", *Proc. ICALEPCS '99*, Trieste, Italy, Oct. 1999
7. L.Mattingly, H.Rao, "Writing Effective Use Cases and Introducing Collaboration Cases", JOOP, Oct 1998
8. A.Cockburn, "Structuring Use Cases with Goals", JOOP, Sep-Oct 1997 and Nov-Dec 1997
9. G. Schneider, J.P.Winters, I.Jacobson, *Applying Use Cases : A Practical Guide,* Addison Wesley, Reading MA, 1998
10. A.Wallander J.Spyromilio K.Wirenstrand, "Commissioning VLT unit telescopes: methods and results", *This volume*
11. M.Kiekebush, J.Pavlich, "Automatic report generator", *This volume*