The NEPTUNE consortium presents

NEPTUNE

Method,
Checking and
Document generation
for
UML applications

TABLE OF CONTENTS

PREFACE

REASONS FOR THIS BOOK

Our position of consultants has shown us that among the increasing number of people working with UML [ref19], [ref7], there is a strong demand of information regarding the way to implement this technology. UML is a modelling language, and as the norm does not define particular implementation guidelines, the field of investigation is still opened. Among the methods already published, we chose to develop and extend the Unified Process (UP), which is indeed the process suggested by the "three amigos". This is what we did and now present in this book. We also associate this instantiation of the UP with software tools aimed at improving the quality of the models, and consequently the final products. This package constitutes the global NEPTUNE offer.

The book has to be seen as a single piece of information, in which the reader can find some interesting stuff about academic domains like model checking, but also the most adapt way to support the NEPTUNE technology, whether it is methodological or software.

We also want to point out the benefits of a global use of the NEPTUNE technology, highlighting the bridges existing between the different facets of the NEPTUNE offer.

It is also important to mention that NEPTUNE, before being a product, was an European project part of the IST program. This book, in addition to being a very useful tool is also kind of a witness, as well as a synthesis of the work performed by the entire NEPTUNE consortium during two years.

The targeted reader of this book is the industrial public, working on UML, and wishing to find a full methodology based upon UML.

MAIN TOPICS

Three themes around UML modelling are widely explored in the following parts and chapters. Theses themes are:

- The methodology
- The model checking
- The generation of documentation

Each of these themes relies upon technologies, whose basics (and more) are given in the annexes at the end of the book. These technologies are

- UML, XMI, OCL
- XML, XSL

All along the book, the themes are developed with references to the aforementioned technologies, and presented for two different professional domains

- Business process
- Software engineering

THE XMI OPPORTUNITY

As the different case tools based upon UML are supposed to be compliant with the norm defined by the OMG, the people from this organisation thought about defining a generic backup format for the UML data. Such a format would for instance allow the exchange of UML data between two case tools developed by different editors.

The XML format being now widely used to exchange data, it is the one that has been chosen by the OMG to support the UML exchange format. A DTD and some guidelines have been defined, leading to the definition of the XMI format, embedding most of the UML concepts. Nowadays, most of the case tools, though having their own proprietary backup format do also provide some XMI features, often through an import/export function. The models can thus be loaded and saved in the XMI format.

For us, this fact is a great opportunity. Because the NEPTUNE offer is not a case tools but an additional tool based upon the UML norm, the

XMI leads our tool to be compatible with any of the existing and future case tools being themselves compliant with the norm.

HOW TO USE THIS BOOK

This book is divided into four separate parts. Each one can be read independently from the others.

The first part is devoted to modelling. Business process and software engineering are successively and independently presented.

Concerning the two central parts of the book, respectively dedicated to model checking and documentation generation, we can say that their organisation is quite similar. For each one, the content gets more and more NEPTUNE oriented as the number of chapter increases. In the first chapters, the reader will find some academic material, to slowly get into the NETUNE oriented material, such as the description of the tool and detailed explanations of how to use it. Consequently, if the reader needs some information about how to use the NEPTUNE software, then we recommend him to jump to the end of the parts. On the other hand, he will focus on the first chapters for anything concerning general information about either model checking or documentation generation.

The annexes constitute a huge source of information. They deal with the different technologies used in the context of NEPTUNE. They are an excellent means to develop further the associated knowledge. Even if it is recommended to read them, it is however not necessary to turn oneself into an expert for the understanding of the core of the book.

ACKNOWLEDGEMENTS

This book is an initiative of the NEPTUNE consortium, which deserves collectively all my thanks. Of course, a consortium is mainly a collection of people, and some of them had a real outstanding influence on the form and content of this book.

First my deepest thanks and appreciation go to L. Pomiès, the main co-ordinator of this book, for his invaluable involvement as author, integrator, and reviewer.

Of course, I express my sincere gratitude to the main authors: J. Ballesteros, P. Bazex, J.P. Bodeveix, J.C. Cruellas Ibarz, L. Feraud, C. Le Camus, J.M. Llovet Pérez, T. Millan and C.Percebois.

I also gratefully acknowledge the help of the other members of the consortium who were able to devote some of their time to reviewing early versions of the book: J. Boixadera Planas, Y. Cassaing, N. Pla Ferrer, A. Jammes, E. Roblet, E.J. Samper, R. Sobek, J.Y. Souillard, and G. Vallès Muñío.

Finally, I'd like to thank my son Jonathan for his creative participation in the graphical design of the cover pages.

This page is also a great opportunity to address a very special thanks to the people that have more or less actively participated to the two years and a half adventure of the NEPTUNE project.

To conclude, I'd like to thank V. Kopanas, our project officer, for his availability and support, and C. Gacek, J.M Morel, and V. Sylaidis, our reviewers, for their constructive work and remarks.

A.CANALS
NEPTUNE Project Leader

PART I

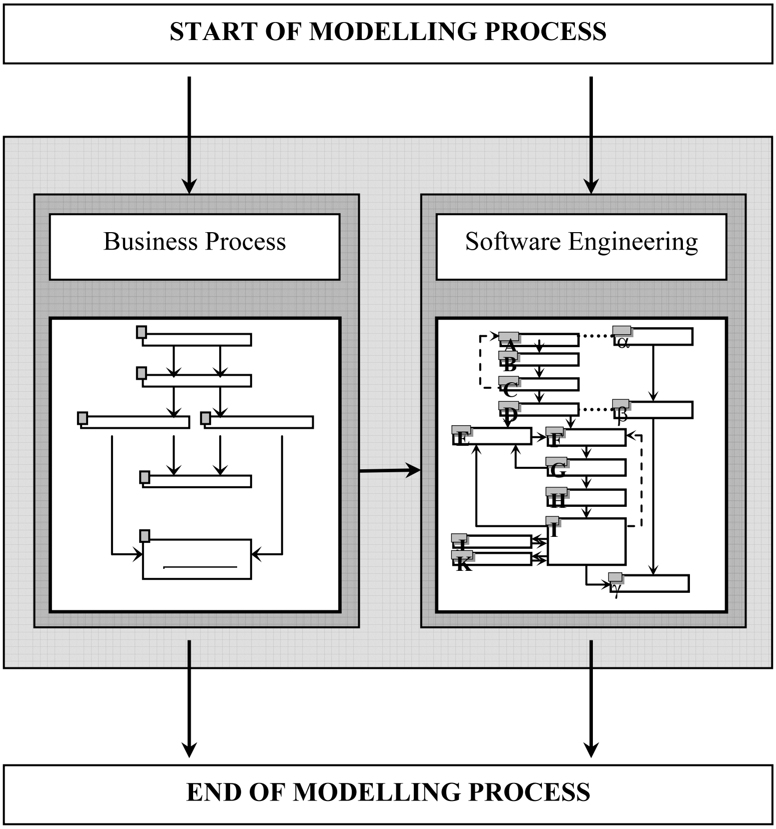
The NEPTUNE method

This first part of the book aims at presenting the NEPTUNE method. This method supplies guidelines and recommendations all along the UML modelling phases, leading to well-organised models.

As UML can either be used to model pieces of software or processes, it is important to mention that the support given by the method is not reduced to software engineering. Thus, the first part deals with the NEPTUNE method applied to particular processes, namely business processes. Then after is presented the method in the software engineering context.

CHAPTER 1: THE NEPTUNE METHOD OVERVIEW

As already mentioned the NEPTUNE method is a great help in modelling software engineering systems but also business processes. This chapter will show how these two approaches can be linked together or applied totally independently. We will see that the correlation level between the two sides of the NEPTUNE method strongly depends on which business field the NEPTUNE user works.



CHAPTER 2: MODELLING BUSINESS

A business model (BM) is a description (usually mixing both text and graphics) of the relevant aspects of the business (goals, opportunity, problems, structure, processes etc) and the manner in which they relate to each other. The level of detail differs according to the perspective of the person creating the model and/or its purpose. A business model serves as a description of how the business is performed and acts as the basis for evaluating and prioritising goals, and constitutes a basic tool for defining the business strategy.

Business modelling is not a new issue at all: the first techniques for modelling business processes date back to the 60's. But nowadays new technologies have increased the interest on this topic of both managers and software engineers. The growth of Internet has brought opportunities for new businesses and new ways of conducting the already existing ones. It has definitely increased the importance of the information systems, which have become a crucial element in the daily operation of thousands of private companies and public administrations. In the end it has acted as catalyst of change that they are facing every day. All these facts lead to a situation where lots of companies must be, from time to time, rethinking their business or the way they conduct them; public administrations are rethinking the public services offer and the way they are offered to the citizens. This raises the need of facing and implementing changes (which can be dramatically big and depth) as quickly as possible and efficiently.

In this context models play a continuously increasing important role in the daily development of today's business. They act as repositories where the business knowledge is concentrated. Depth and non ambiguous knowledge allows to assess the business as a whole, to rethink it and to correctly conduct the required changes. Knowledge is also crucial in the development of a kind of "language" for giving support to a sort of "company culture" shared by the staff. Business models also play a crucial role in the specification of the information system (IS) infrastructure that an organisation requires to achieve its goals in an efficient way, as they describe the different elements (business goals, business actors, resources, business processes) that constitute the daily life of the business and allow to extract their respective requirements for the supporting IS. Even when these IS already exist, business models are valuable as they make explicit among other things, the different business processes that are supposed to be supported by them, raising inadequacies or mismatches or simply adding detailed information that was not previously available. From a broader perspective, business models constitute valuable inputs for standardisation initiatives in the electronic transactions world (UN/CEFACT, ebXML, etc).

Due to the aforementioned reasons, it is not uncommon that modelling business becomes, for more and more specialists, a crucial discipline in software development process.

Different methodologies for building up models have been proposed, most of them referred to software modelling paradigms (e.g. object-oriented, data-focussed, etc). Examples of the methodologies currently proposed and used are: RUP (Rational Unified Process) and UMM (UN/CEFACT Unified Modelling Methodology [ref1]). The ebXML (electronic business XML) project has produced a profile of the UMM for modelling business dealing with exchange of electronic documents.

Methodologies are usually supported by a number of techniques. These in turn are supported by specific tools such as CASE tools, Workflow Management Systems etc. Some of the more well-known business modelling techniques are: Flowcharting, IDEF Techniques (IDEF0, IDEF3), Petri Nets, Simulation, Knowledge-based Techniques, Role Activity Diagramming, Entity-Relationship Diagramming, State-Transition Diagramming and Unified Modelling Language (UML). Nowadays the effort on specific techniques used in business modelling, has resulted in an increasingly broad range of tools made available for the specialists. A very relevant example are the Eriksson-Penker Business Extensions [ref2], that have taken advantage of the UML extension capabilities and built up a number of model patterns in the business modelling area.

Overview

The aim of this section is to give an overview of the NEPTUNE business modelling discipline. As it has been stated before, it is built on the UMM proposal with additions derived from the work of Eriksson and Penker to deal with uncovered aspects by UMM itself. In consequence it is worth to start by shortly describing what was initially UMM designed for, and which is its normal scope of application.

The main objective of UMM was "to model Business Processes and support the development of existing 'Next Generation' EDI for electronic business". As such, UMM was conceived to give support to the standardisation of electronic versions of common commercial documents in the light of latest technological advances: massive usage of Internet, object orientation paradigm in software engineering and widespread usage of XML syntax in the Web. UMM focuses mostly on those aspects related with commercial transactions. However, business also deal with a much more complex set of topics that any methodology looking for effectively modelling a business as a whole must take into account:

- Businesses are carried out by more or less complex organisations.
- Organisations conduct an extremely high diversity of processes.
- Entities that conduct business must accurately define their specific objectives, means to review them, and means to measure the degree of achievement.
- Entities conducting business must achieve their goals with limited resources. In addition, they have to clearly identify potential problems for these achievements and define solutions for these problems.
- Companies, organisations and public administrations usually relay on IS to give support to their daily activities. Adequacy of the IS to the daily activities and adaptability to changes is a critical issue.

Eriksson and Penker have identified a number of relevant views for a business that incorporate all these topics. In addition they have proposed means for modelling the most important aspects in each one using UML; they have also defined a number of modelling elements for relevant aspects in this field; finally, they have proposed a set of diagrams for modelling business from different views.

The business modelling discipline defined by NEPTUNE incorporates activities, steps and work products proposed by both UMM and Eriksson and Penker (who identified a number of business views that NEPTUNE incorporates).

NEPTUNE acknowledges the fact that businesses can be conducted in a number of relevant but different business areas and that each one requests an accurate modelling work. Because of that most of the activities proposed by NEPTUNE must be carried out for each area.

NEPTUNE business process discipline proposes the work definitions that are mentioned and shortly explained below:

- **Vision of the business**. It includes, among other issues, the justification of the business, a high level description of the objectives, the identification of the relevant areas, stakeholders, constraints, etc.

- **Business concepts identification**. This activity will be carried for each business area relevant to the organisation. Although it is mentioned in second, it will actually be completed iteratively and in parallel with the rest of the activities because these ones usually bring deeper knowledge of what it is being modelled and in consequence improve its conceptual perception.
- **Goals and problems identification**. This activity will be performed at two levels. The work at the higher one will usually be carried out by the promoters of the business likely with the help of external consultants. At the lower level, this activity must be performed for each relevant area for the business, and must include definitions of achievement assessment procedures.
- **Resources and organization structure analysis**. This activity deals with the production of a model of the actual organization that must conduct the daily activities in the business, its organizational units, the available resources and the managed information. If the model is accurate, it will be possible to assess crucial issues like the adequacy of the organization and resources to the goals in each business area, or the cost of reorganization of certain organizational units to support changes in the business, or even the adequacy of the IS supporting the business, etc.
- **Business Processes identification and description**. This activity must be performed for each business area. It will likely involve the identification of different categories of processes even within one single business area..
- **Business behaviour description**. This activity deals with the interaction between processes and resources.
- **Requirements Definition** (Preparation for supporting IS production). If the ultimate goal of the work is to produce an IS, a number of steps will have to be accomplished in order to take the results of the previous activities and prepare the rest of the work devoted to obtain such a system.

NEPTUNE business modelling discipline proposes to carry out a number of meeting-interviews in order to perform certain activities in the aforementioned work definitions. The first of the following sections details the established techniques for conducting these meetings and interviews.The next sections present the relevant details of each work definition. These include:

- Introduction to the activity itself.

- Table showing the actors who will carry out the activity, the inputs that they will use, the outcome of the activity, and the steps to perform the activity.
- Details on the different aspects summarised in the aforementioned table. This can include, wherever needed, additional details on the technique for carrying out meeting-interviews.
- Templates for those document that can result from the work done in the activity.

Established general technique for conducting meeting-interviews

Some activities in NEPTUNE business modelling discipline actually are meeting-interviews carried out to extract business knowledge and model it. This section shows the process that NEPTUNE has established for conducting them. This process is iterative. Usually, after each meeting, one or several documents must be produced. Achieving their final versions may take more than one meeting as their contents must be reviewed, modified and eventually agreed by all the implied actors. Usually one of the implied actors is a business analyst who is in charge of conducting the meeting-interviews in a proper manner so that the business knowledge is eventually extracted and accurately modelled.

Below follows a textual description of the steps foreseen by NEPTUNE for conducting the meeting-interviews:

1. First of all the business analyst must prepare scripts for the interviews. These scripts must detail the relevant aspects that the corresponding team will work on during the meeting (they usually will contain questions about these aspects addressed to the participants). Their specific contents strongly depend on the precise activity in the NEPTUNE discipline. Sections below include specific proposals for these scripts.
2. These scripts, along with the agenda for the meeting must be circulated among the participants days before the meeting. In this way, they can have enough time to prepare the meeting. In certain cases, e-mail exchanges before the meeting have to be maintained in order to clarify both, certain aspects of the script and/or the scope of the meeting.

3. One meeting-interview is carried out. This working session focuses on the aspects mentioned in the circulated script. As the session goes on, the different aspects mentioned in the script must be worked and questions answered. Meetings must not be too longs, as it is shown that after certain time period, it is not likely to get good additional results. At the end of the meeting a short review of the covered issues and those not treated must be done and agreed. It must also be agreed a calendar for the next steps. This includes: production of the documents formalising the results obtained, including short minutes of the meeting, period for comments on the documents, and date for the next meeting if needed.
4. The day fixed in the agreed calendar, the documents must be circulated among the participants.
5. During the established period, the participants read the documents and raise comments that eventually can introduce some changes, help the business analyst to propose an agenda for the next meeting-review, and even lead to modify the original meeting script.
6. If the comments reveal that certain points need additional clarifications, the next meeting-interview firstly deals with them before going on with the list of issues to deal with. Only when a complete agreement has been achieved in them, the team will proceed to discuss other items in the business analyst script.

Steps 3 to 6 are repeated until all the aspects of the business analyst script are covered in a proper manner. The final meeting must be devoted to formally give the approval to all the documents generated as a result of all the work.

Script for the meetings-interviews
As an example, below follows the script distributed to the attendees of the meetings-interviews for identifying and validating business rules.

Before the meeting-interview: A1.- Analysis of the documentation available for each process by all the participants. A2.- To send a copy of the clause “During the interview” of the present script.
During the meeting-interview: E1.- Present the objectives of the project to the process expert. E2.- Clarify concepts if the interviewers have doubts after studying the process.

E3.- Review by all the participants of: ▪ The objectives of the process. ▪ The description of the process ▪ The process diagram, making special emphasis in the activities sequence, bifurcations, loops, etc. ▪ Analysis of the coherence of the process diagram and the textual description of the process. ▪ Actors of the process. ▪ Analysis of the activities in the primary path ("normal path") in the process. ▪ Analysis of the activities of the alternative paths in the diagram process. ▪ Analysis of the causes for state changes within the primary path. Identification of triggers. ▪ Analysis of exception causes that originate that the process follows alternative paths. Analysis of the triggers for the state changes in these paths. ▪ Context circumstances, cultural elements, and implied issues that can affect the process. ▪ Information volumes and work load, automated procedures and manual procedures. Tasks and circumstances that people involved in the process take into account when they have to make decisions. ▪ General comments on the process. E4.- Summary of the process and very first list of pendent issues that require ulterior work or/and documentation.
After the meeting-interview: D1.- Analysis of the coherence of the obtained results and the stated objectives and the description and initial documentation. D2.- Identification of pendent issues. D3.- Identification and formalisation of the business rules. D4.- Production of the minutes of the meeting-interview with the table of the business rules identified and additional comments when required. D5.- Delivery of a copy of these minutes to all the participants for comments. D6.- Reception of comments and modifications of the minutes. The comments can consist in the acceptation of the business rules in the table or suggestions for changes. This feedback can lead to the agreement of further discussions during the next meeting.

Vision of the business

This activity produces a number of outputs that help to setting up the strategy of the business. The following table summarises the most important details of the work definition, namely, actors, inputs, outputs and activities.

Actors:	▪ Business Expert (Managing Director or Board of Directors) ▪ Business Analyst
Inputs:	▪ None (unfortunately a common situation) or ▪ Business Plan (for new business) ▪ Strategic Plan (Forecast for a period of 3 / 5 years for an already existing business)
Outputs:	▪ Vision of the Business ▪ Business general goals. ▪ Currently used Business concepts in a particular company ▪ Current organisation of the company ▪ Main activities, products and services performed / supplied by the company ▪ TOWS matrix for the overall company

The vision of the business is a short document giving hints, among others [ref3], of the mission of the company, their global objectives, the strengths, its weaknesses, its opportunities and threats, the major strategies, etc.

It has to be mentioned that business goals can be defined at different levels within an organisation. The current activity concentrates on the global ones. The business' owners must define them, and once this has been done, it is unlikely to have substantial changes during long periods of time, unless a radical change occurs in the business. Usually, it makes no sense to model them.

Work techniques

The actors will conduct a number of meetings to produce the required material to be included in the output documents. For each interview the business analyst will produce a script that will guide its development.

Suggested base script

The figure below shows the suggested script that the business analyst could use as a base .

Before the meeting:
Same structure than previous meeting script

During the meeting:
Follow the following index of topics, being the meeting a mix of brainstorming and of high level analysis, an exercise to review the Business Plan or the Strategic Plan with the aim of fixing objectives, threads, strengths, weaknesses and opportunities, detailing products, main activities, services and organisation necessary to reach the desired goals.
The usual mechanism should be an iterative process (in several meetings) with the same index until a clear result shall be obtained. This result must be negotiated and accepted by all members involved in the future of the company (management and shareholders representation)

- Vision of the Business
- Business general goals.
- Currently used Business concepts
- Current organisation of the company
- Main activities, products and services performed / supplied by the company
- TOWS matrix for the overall company

After the meeting:
same structure than previous meeting script

Business concepts identification and definition

During this activity, a shared vocabulary for the business will be produced, modelled and agreed. There will also take place the identification of the high-level daily processes and/or activities (organised by business areas), as well as the relevant stakeholders and business rules for them. This is essential for getting a complete, accurate, understandable model; for avoiding misunderstandings; and for allowing a real sharing of the knowledge among the actual players in the business.

The following table summarises the most important details of the work definition, namely, actors, inputs, outputs and activities.

Actors:	▪ Business Expert (Director or Expert) ▪ Business Analyst (Internal or external Consultant acting also as Modeller) ▪ Occasionally any of the Managers or Directors of the company
Inputs:	▪ Vision of the Business ▪ Currently used Business concepts in a particular company ▪ Knowledge of the current organisation of the company ▪ Main activities, products and services performed / supplied by the company ▪ TOWS matrix for the overall company ▪ List of business goals (derived from the vision and strategy of the company)
Outputs:	▪ A standardised Business Concept's Dictionary (list of Business Concepts) ▪ Class diagrams for Conceptual Modelling ▪ Detailed list of relevant processes and/or activities performed by the company ▪ High-level list of stakeholders and resources (for each area or process) ▪ List of business rules (derived from the day to day activity)
Activities	▪ Identification, definition and modelling of the business concepts ▪ Identification of processes and/or activities performed by the company, including relevant stakeholders and resources and business rules.

Activity 1: identification, definition and modelling of the business concepts

The actors and the inputs for this activity are the ones mentioned in the table before. Below follow details on the outputs and the work techniques suggested for completing the activity

Work techniques

The actors will conduct a number of meetings to produce the required material to be included in the output documents. The technique for conducting the meetings will be the one explained in the corresponding section above, with the specific considerations that follow below:

- The script must focus on extracting as much relevant concepts to the business as possible.
- The business analyst must focus his effort in the identification of relationships among the concepts. He has also to work for achieving agreed and unambiguous definitions of the identified concepts.
- The objective of the first meetings will be to get the dictionary already mentioned. It will contain all the identified concepts, their corresponding definitions and their most relevant relationships with other concepts.
- This dictionary will be worked out by the business analyst that will also play the role of modeller, and he will produce the conceptual model in UML. One or more meetings will then be conducted in order to review and approve the model.
- From time to time, and as business modelling progresses, or even after the issuance of the business model itself, the results obtained from this activity may need to be reviewed (new concepts can arise, existing concepts might not be relevant any more, or change might occur in their definition, etc). The outputs of this activity have to be live documents, and the team may revisit and change them. While the business model is being produced, the team must establish these mechanisms, and repeat the meetings accordingly. Once the corresponding version of the model has been finished, the organisation must define a work-plan for reviewing and updating it.

Suggested base script

See meeting script suggested contents at the end of Activity 2: identification of processes, resources and stakeholders

Outputs

The outputs of this activity will be the Business Concept's Dictionary and the conceptual model expressed as a set of UML class diagrams.

Suggested base contents of the dictionary

The available dictionary could be organised in a tabular form, so that identification of concepts and their relationships would be easy.

The details to be given for each concept could be, among others: concept name, concept definition, relationships with other concepts, multiplicity of these relationships, business area(s) where the concept applies, processes that use the concept

Being given the widespread of XML editors it is even possible that the initial output generated by the business analyst could be a XML document and that the NEPTUNE document generator (or any other tool incorporating XSLT capabilities) could be used to produce the final and available dictionary.

Activity 2: identification of processes, resources and stakeholders

The actors and the inputs for this activity are the ones mentioned in the table before. Below follow details on the outputs and the work techniques suggested for completing the activity

Work techniques

As before this activity will be performed by setting up several meeting-interviews. Below follow some aspects that the business analyst must take into account when planning and conducting them:

- First of all, the business analyst must focus on identifying business areas.
- For each of the business areas, the participants have to work in identifying the stakeholders, the relevant processes and the resources that they require.
- Special care has to be taken when dealing with those processes that can not be embedded within only one area, but pertain to several ones. They must be unambiguously identified.
- Concerning the stakeholders, a short enumeration of their corresponding interests and played roles must be produced.
- During this activity, the participants have to identify and formalise high level business rules (those that are related to general policies of the organization). It is essential to include in the script questions that lead the team in that direction.

Suggested base script

The figure below shows the suggested script that the business analyst could use as a base (suitably profiled). As Activity 1 and 2 are very close one to the other, meetings can be performed to reach conclusions for both. A remaining task for the consultant should be to identify and divide processes from business concepts. Therefore a common script should be:

Before the meeting: same structure than previous meeting script
During the meeting: Follow carefully the “list of main activities, products and services” and “knowledge of the current organisation of the company” which are inputs, in order the identify in detail processes and activities for each Organisational Unit, concepts used by OU, process or globally the whole company, stakeholders and resources for each process as well as the explicit or hidden business rules

Remarks: ▪ A good issue shall be obtained when the consultant working in the activity has enough experience either in business or in Public Administration or in both. ▪ Experienced managers are naturally an excellent starting point for these activities
After the meeting: same structure than previous meeting script

Outputs

Suggested template for the document

Outputs:
Enumeration and a short description of the different relevant business areas. The stakeholders and their expectations. An enumeration of the more important processes (including a short explanation) and the resources consumed For each process: ▪ List of resources required by the process ▪ List of stakeholders involved in the process, indicating which activities are to be performed by each one

Goals and problems definition

This work definition deals with the modelling of specific goals (for each business area) whose achievement is needed to accomplish the general ones present in the business vision. These specific goals must be documented and modelled (using goal/problem diagrams proposed by Eriksson and Penker) along with the expected problems and measures to counter them. There will be periodical reviews for both measuring the degree of accomplishment and redefining the goals themselves for the next period.

The table below shows the relevant actors, inputs, outputs and activities.

Actors:	▪ Business Expert ▪ Business Analyst

	▪ Occasionally any of the Managers or Directors of the company
Inputs:	▪ List of business goals (derived from the vision and strategy of the company) ▪ TOWS matrix for the overall company
Outputs:	**Goal** (for each goal): ▪ TOWS matrix for each goal (Global TOWS filtered in order to see only those topics related to the specific goal) ▪ Goal definition ▪ Processes involved in the goal or problem ▪ Resources required to reach the goal / solve the problem ▪ **Goal**/Problems **model** ▪ **Plan for** reviewing **the accomplishment of the goals.**
Activities	▪ Identification, definition and modelling of specific goals, their associated problems and envisaged solutions. ▪ Definition of mechanisms for measuring the accomplishment of the goals. ▪ Elaboration of a plan for reviewing the accomplishment of the goals.

The two first activities mentioned in the table, are strongly related and will be carried out in parallel. The third one will be a different activity.

Activity 1: identification, definition and modelling of specific goals and problems

The actors and the inputs for this activity are the ones mentioned in the table before. Below follow details on the outputs and the work techniques suggested for completing the activity.

Work techniques

The actors will conduct a number of meetings to produce the required material to be included in the output documents. The following considerations will apply for this activity:

- First of all, the business analyst must focus on obtaining a concise definition of high-level goals.
- After that, the team must focus on breaking it in a number of lower level sub-goals. This process should be continued to the extent where no further decomposition can be reached.
- For the ones that are quantitative, the participants have to agree in both, quantities and units.
- For those that are qualitative, the participants should try to identify indicators related to those goals.
- For each goal identified, the business analyst should conduct the meeting so that the participants might be able to accurately define a number of mechanisms to measure its degree of accomplishment. They should allow establishing levels of accomplishments.
- For each goal identified, the business analyst must focus on making the team aware of the importance of an accurate description of the most relevant problems that could prevent the fulfilment of the goal. As a second part of this step, they should also accurately describe possible solutions for each problem.

Suggested base script

The figure below shows the suggested script that the business analyst could use as a base (suitably profiled). This script is also valid for activities 2 and 3 that follow this first one.

Before the meeting: same structure than previous meeting script
During the meeting: Detailed review of Company goals, identifying which one interacts with particular OU processes and detailing subgoals involved on each process. During this phase must be done the identification of problems related to each goal and the definition of the way to measure any particular goal and the automatic mechanisms to implement them. As well as the goal definition it must be performed a review of the problems arisen along the meeting and relate them to the goals (reverse analysis) in order to verify the coherence and consistency of the results. A set of alerts and metrics to measure the accomplishment of the goals should be defined or proposed in the meeting too, as well as the frequency

they shall be applied and reviewed.
After the meeting: same structure than previous meeting script

Models

NEPTUNE proposes to produce a goal/problem model as defined by Eriksson and Penker. It includes a collection of object diagrams. Each one corresponds to one high level goal, and shows its sub-goals (being also possible to indicate whether their achievement would mean to achieve the goal itself) along with their corresponding problems and solutions. All of them are modelled using stereotyped UML modelling elements.

Eriksson and Penker propose to model goals as objects of classes that are stereotyped to <<**goal**>>. They define two different goal classes: **Quantitative Goal** and **Qualitative Goal**. This allows to the modellers to define class attributes for quantitative units, and class operations for measuring the degree of goal fulfilment.

Dependency relationships are established between a goal and its corresponding sub-goals. A dashed line crossing all these relationships with a constraint indicates whether the sub-goals set is complete or incomplete.

Eriksson and Penker model problems as stereotyped notes to <<**problem**>>. The text of the note defines the problem. The causes of the problem are modelled as stereotyped notes to <<**cause**>>, linked to the problem itself. The model for the problem is completed by linking to the cause a note stereotyped to <<**action**>>, where the tentative solution to the problem is sketched, and a final stereotyped note to <<**prerequisite**>>, linked to the previous one, that contains the prerequisite for the action.

Activity 2: Definition of mechanisms for measuring the accomplishment of the goals.

The team will produce a set of alerts and metrics to measure the accomplishment of the goals: what, when, who, how (mechanisms and tools, range of values, etc.) will the goals be reviewed.

Activity 3: Elaboration of a plan for reviewing the accomplishment of the goals

The team will produce a plan for periodic goal.

Resources and organisation structure analysis

This work definition focuses on the modelling of the organisation conducting the business, including the available resources. Organisation models are crucial for achieving a suitable deployment of organisational and human resources in the view of the business goal. They also become an ideal way of knowledge sharing among all the players in the organisation itself.

The table below shows the relevant actors, inputs, outputs and activities.

Actors:	▪ Business Expert ▪ Business Analyst ▪ Occasionally any of the Managers or Directors of the company
Inputs:	▪ Current organisation of the company ▪ Main activities, products and services performed / supplied by the company ▪ TOWS matrix for the overall company ▪ List of Company goals (derived from the vision and strategy of the company) ▪ TOWS matrix for each goal ▪ Resources required to reach the goal / solve the problem
Outputs:	**Organisation level:** ▪ Departments and Areas in the organisation ▪ Organisation model **Department level:** ▪ List of processes that are completely or partially performed by each Area (if partially, list of activities performed) ▪ List of Resources used by the Area (and processes to which they are completely or partially assigned)
Activities	▪ Building the organisation model of the company ▪ Identifying the processes owned by the Areas.

Activity 1: building the organisation model of the company

The outcome of this activity will be a set of class and object diagrams. In the first ones the basic structure and the rules that govern it will appear. In the second ones, the actual organisation will be modelled.

Activity 2: Identifying the processes owned by Areas

The actors will identify the processes present in each area, as well as the resources allocated to them. Special care will be made on those processes that are owned by different areas or departments: dependencies of several organisational units add complexity to the overall management.

Suggested base script

The figure below shows the suggested script that the business analyst could use as a base (suitably profiled).

Before the meeting: same structure than previous meeting script
During the meeting: Detailed review of Company resources, Company Organisation, and allocation of the resources to processes and department, in order to analyse whether this allocation is balanced to the needs and response time required. To do this task properly the suggested way should be: • Analysis of each process performed by any department and measure the time needed to perform each task and the time for waiting situations • Define a pattern for each task including resource allocation • Review the process and the resources obtained comparing this result with the existing or desired available resources
After the meeting: same structure than previous meeting script

Business processes description

This work definition deals with the production of models for the processes that the actors within the organisation will put in place to achieve the business goals, solving the identified problems. The required resources to successfully accomplish them are also modelled. As the organisation models, process models are an excellent way for creating shared knowledge of the organisation and its day by day work.

The table below shows the relevant actors, inputs, outputs and activities.

For each Organisational Unit and process	
Actors:	▪ Business Expert ▪ Business Analyst ▪ Occasionally any of the Managers or Directors of the company
Inputs:	**Department level:** ▪ List of processes that are completely or partially performed in each Area (if partially, list of activities performed) **Goals:** ▪ For a Business Area list of goals in which this Area is involved
Outputs:	**Process Description:** ▪ Summary of the part of the Process performed by a particular Organisational Unit
	Scope: ▪ Entities that interact in this business area or process
	Inputs: ▪ List of physical inputs (or providers) ▪ List of informational inputs (internal or external)
	Outputs: ▪ List of physical outputs ▪ List of informational outputs
	Resources: ▪ List of machines, people and information (manuals, process instructions, customer order and specifications)
	Stakeholders: ▪ Internal or external users involved either in the process or in its trigger or in its delivery or acceptance

Work Techniques

As usual, the team will conduct a number of meetings according to a script that will be shown in the sub clause below.

This work definition will have two main results:

– A number of process diagrams as the ones suggested by Eriksson and Penker.
– Complementary documentation dealing with all the relevant issues mentioned in the table above. Tools like NEPTUNE, which can link UML models and XML files with complementary information, and generate documents, can play an important role.

Models

Eriksson and Penker propose to use a stereotype (<<**process**>>) of an UML activity element for modelling a process. The relationship of one process with its environment is done through special UML activity diagrams, called process diagrams. These diagrams incorporate stereotyped objects, the model, the goal(s) that the process is designed to achieve, and resources. Resources are further classified according to:

□ The type of resource. Resources modelled can be people (<<**people**>>), physical objects (<<**physical**>>) or information (<<**information**>>).

□ How they are present in the process: as input, output, supplying (stereotype <<**supply**>>) or controlling objects (<<**control**>>).

Suggested base script:

Before the meeting: same structure than previous meeting script
During the meeting: For each process ▪ Identification of the activities that belong to the process ▪ Inputs and outputs for each activity ▪ Activity sequencing, iteration or selection ▪ Conditions for sequence, iteration and selection. Business rules ▪ Triggers, timers and other mechanisms ▪ Users and stakeholders ▪ Volumes and frequence ▪ Identification of other processes that interact with the present

process and information exchanged
After the meeting: same structure than previous meeting script

Business behaviour description

This work definition will focus on the behaviour of the business studying, modelling and documenting:

- How the different processes interact among them.
- How critical resources in the process evolve.

The table below shows the relevant actors, inputs and outputs.

Actors:	▪ Business Expert ▪ Business Analyst ▪ Occasionally any of the Managers or Directors of the company
Inputs:	**Department level:** ▪ List of processes that are partially performed by each Area ▪ List of activities performed by each Area in the partial process ▪ Resources used in the processes
Outputs:	**Process:** ▪ Process Name **ResourceName** ▪ Behaviour of individual resources in a single process: states, events and actions ▪ Interaction diagrams ▪ Statechart diagram ▪ Sequence and collaboration diagrams ▪ interaction diagrams showing interaction between processes) ▪ Activity diagrams for each process. **Constraints:** ▪ Process Business Rules

Work techniques

The team will work for identifying the business rules that govern the different processes. These rules will be identified during the meetings as suggested in the script shown at the beginning of the chapter. Tools like NEPTUNE, able to incorporate them and conduct checks of the produced models against them will play a relevant role in validating these models.

The team will also build process diagrams showing how distinct processes interact and are run by different organisational units (using swimlanes). For each process, they will produce a specific activity diagram with the details. Checks of these diagrams against the aforementioned rules will help in validating their correctness.

The team will also devote its effort to build up statechart diagrams for those resources identified as critical.. It is common that processes run by different OUs show more difficulties, and as a rule of thumb, it is worth to consider them first.

Proposed base script

For the identification of the business rules, the base script proposed at the beginning of the chapter.

For the production of the models, the following one:

Before the meeting: same structure than previous meeting script
During the meeting: For each process ▪ States (final situation after an event), events (external inputs) and actions (response to the events) ▪ Identification of the activities performed by an OU within a particular process when partially performed ▪ Detailed identification of activities within processes interacting with the actual process and information exchanged
After the meeting: same structure than previous meeting script

Requirements definition

Finally, this work definition focuses on defining the requirements imposed by the business to an information system that would support it. It also deals with the initial steps of the requirements analysis that will lead to the definition of the use cases.

Actors:	▪ Business Expert ▪ Business Analyst ▪ Computer Analysts
Activities	▪ Requirements definition ▪ Requirements Analysis ▪ Information Systems Analysis
Inputs:	▪ Business Process description ▪ Business Process behaviour
Outputs:	**Requirements definition** ▪ list of functional requirements ▪ list of not functional requirements **Requirements analysis** ▪ Assembly line diagrams ▪ Budget for building or purchasing the Information System

One of the outcomes is a set of Eriksson and Penker specific diagrams called assembly-line diagrams, which they propose to use as a way to derive the Use Cases for the systems.

An assembly-line diagram is a process diagram that has in its upper part a number of processes and in its lower part a number of horizontal packages (stereotyped to **<<assembly line>>**). They represent whatever group of objects the modeller decides. References coming from/to processes and assembly lines represent operations of the processes (reading/writing) on the objects. In these diagrams, the assembly lines may model objects of the information system to be developed, so that the references model information flow between the processes and this system, which actually is what Use Cases show. By selecting groups of these information flows, the team can identify the relevant set of Use Cases for feeding the software development process.

Proposed base script

Before the meeting:
same structure than previous meeting script

During the meeting:
For the whole Information System:

- Identify processes that are to be totally or partially included
- Identify information system process limits, ranges and interfaces with other IS
- Identify WHICH / WHAT information is going to be captured and produced by the IS, but not HOW the produced information is going to be calculated or obtained (i.e. list of functional requirements)
- Propose a prototype for the user interfaces of the Information System
- Identify stakeholders that shall use the IS and the hardware required
- Identify topics like response time, ergonomics, error handling, security and all non functional requirements

For each process within the Information System to be developed

- Review of documents generated during "Business Process Description" and "Business Behaviour description" activities
- Identify documents and data to be stored on electronic format
- Identify volumes of information and number of process instances.

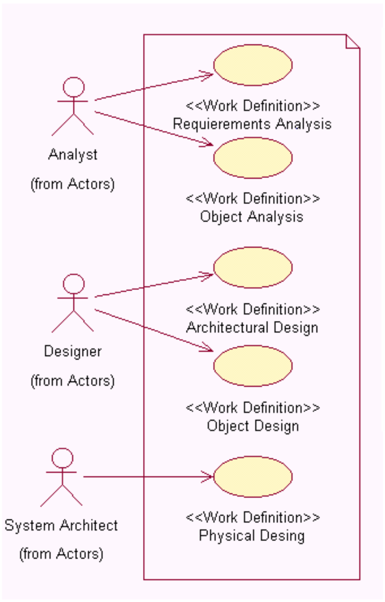
After the meeting:
same structure than previous meeting script

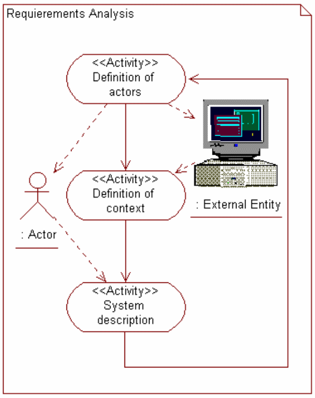
CHAPTER 3: MODELLING SOFTWARE

This chapter aims at presenting the NEPTUNE "Discipline", called **UML-NEPTUNE**, applied to software engineering. We can note two important things: first, the UML-NEPTUNE is a specialised instance of the analysis and design part of the unified process [ref18], and is also compliant with the MULLER approach, [ref16]. Second the UML-NEPTUNE is instrumented by the use of the NEPTUNE software.

Overview

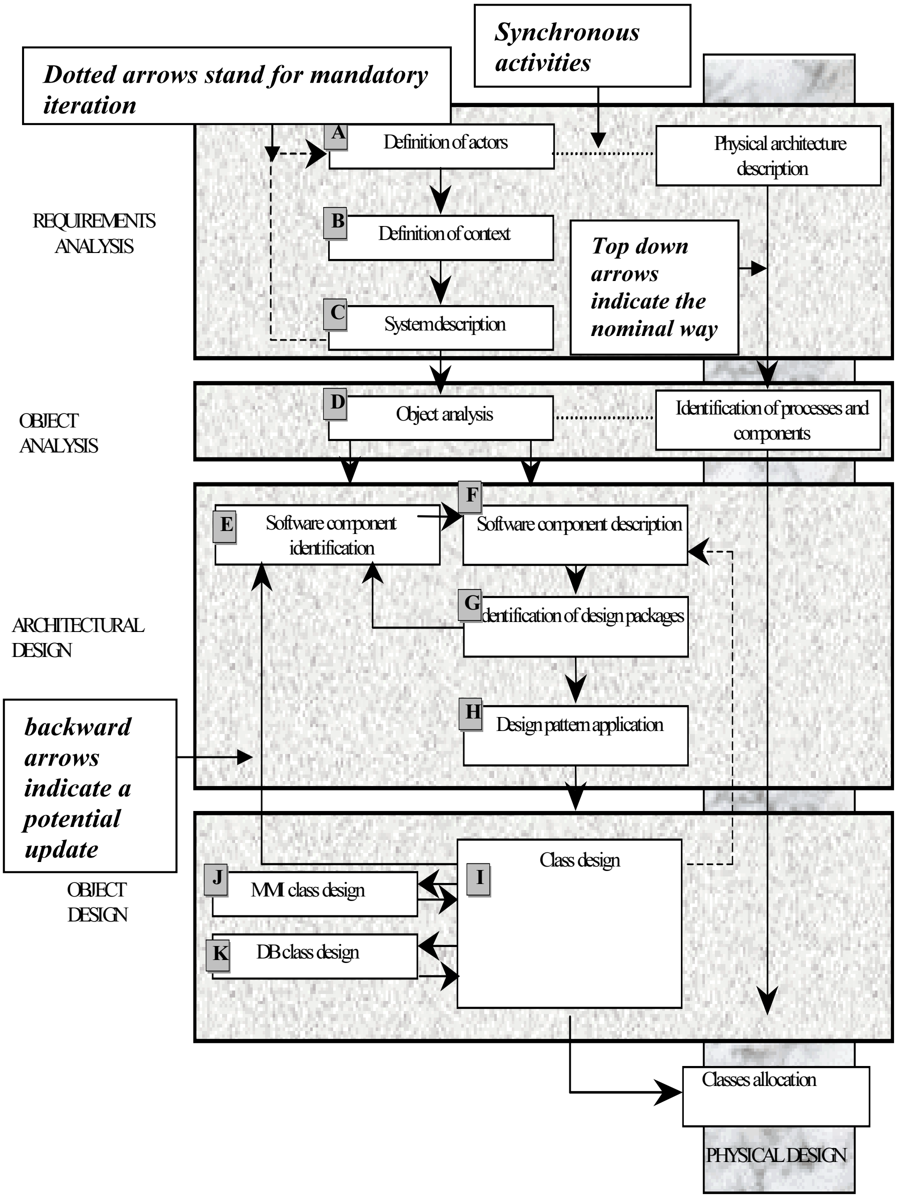
Within this section is presented an overview of the UML-NEPTUNE that is based on the UML/CS approach, [ref10]. It is shown, that UML-NEPTUNE makes use of "Work Definitions", and that each "Work Definition" is composed of "Activities". A first approach of the various "Work Definitions" and "Activities" is presented in the diagrams below.


Analyst
(from Actors)
Designer
(from Actors)
System Architect
(from Actors)
<<Work Definition>> Requierements Analysis
<<Work Definition>> Object Analysis
<<Work Definition>> Architectural Design
<<Work Definition>> Object Design
<<Work Definition>> Physical Desing


Requierements Analysis
<<Activity>> Definition of actors
<<Activity>> Definition of context
<<Activity>> System description
: Actor
: External Entity

The two first diagrams use the formal UML notation while the third one is a complete view of the process in an artistic and informal “modelling style” !

IMPORTANT: *The presentation of UML-NEPTUNE made in this book is sequential for easy understanding, although it is naturally iterative and incremental.*


Synchronous activities
Dotted arrows stand for mandatory iteration
A Definition of actors
Physical architecture description
REQUIREMENTS ANALYSIS
B Definition of context
Top down arrows indicate the nominal way
C System description
OBJECT ANALYSIS
D Object analysis
Identification of processes and components
E Software component identification
F Software component description
G Identification of design packages
ARCHITECTURAL DESIGN
H Design pattern application
backward arrows indicate a potential update
I Class design
J MMI class design
K DB class design
OBJECT DESIGN
Classes allocation
PHYSICAL DESIGN

Notes:

1. *The activities presented in our discipline are not systematically implemented in the process. We can say that the execution sequence is depending on the content of the system modelled. For instance, under specific conditions (described in the next paragraphs) the activity B can be bypassed. The analyst will then move directly from activity A to activity C.*

2. *The discipline does not focus on the activity diagrams. However, replacing a sequence diagram with an activity diagram can be valuable. The readers interested in this UML diagram can read the following article that details five different contexts of use [ref12].*

Requirements Analysis

The aim of this "Work Definition" is to get a complete description of our system environment, focusing on the user needs. The objective is thus to identify the external actors, the data flows between the actors and the system and finally the list of Use Cases.

Recommendation: *It is strongly recommended not to go on to the next "Work Definition" before getting the user approval (validation) of the results of this first modelling stage.*

Actors:	• The **analyst team**, specialised in collecting and formalising the user requirements. The key question here is: "**What** is the system?"
Inputs:	▪ User Requirements (needs)
Outputs:	▪ UML Model ▪ MMI prototype (if needed) ▪ Requirements Analysis Document ▪ Validation Plan Document
Activities:	▪ Activity A: Definition of Actors ▪ Activity B : Definition of Context ▪ Activity C : System Description

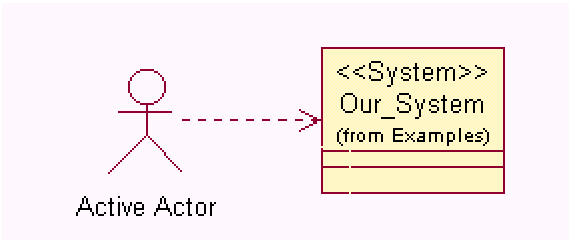
At the end of this "work definition" it is recommended to use:

- the NEPTUNE checker, to check the requirements analysis rules,
- the NEPTUNE document generator to generate the "Test Cases Forms" from the use cases.

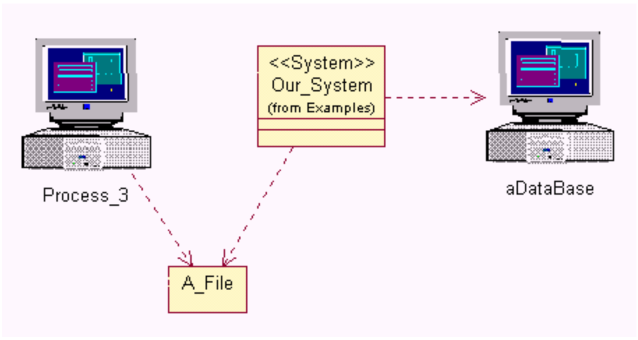
Activity A: Definition of Actors

This activity consists in defining the actors (an actor is "something" **always** out of our system). UML-NEPTUNE suggests two kinds of actors:

- An "active" actor (the standard UML actor) is a participant or a process role, which interacts with the system to be modelled, and actively works in the realisation of the system Use Cases,



- And "external entity" is a stereotyped actor that does not actively participate in the realisation of the system Use Cases (e.g. a process producing or consuming data for our - but not necessary only - system, but **not in direct communication**).



To find these actors, the most important things to wonder are “WHO WILL USE OUR FUTURE SYSTEM”, and, “WHAT (not to say WHO) WILL BE USED BY OUR FUTURE SYSTEM”.

Recommendations:

1. *Each actor must be described through his role and his responsibilities.*
2. *During this activity, it is possible to start the "physical Architecture Description" activity, the two activities running parallel.*

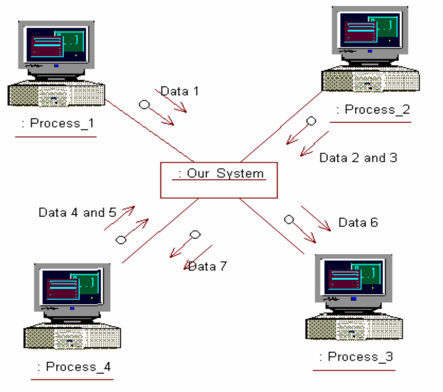
If "External Entities" have been identified during this activity, the next one is then activity B. If not, the following step is activity C.

Activity B: Definition of Context

This activity consists in finding the static and dynamic data flows between the external entities and our system. These data flows show that our system uses or produces data without direct communication.

To find this data flows the most important thing to wonder is "WHAT EACH EXTERNAL ENTITY WILL EXCHANGE WITH OUR SYSTEM?"

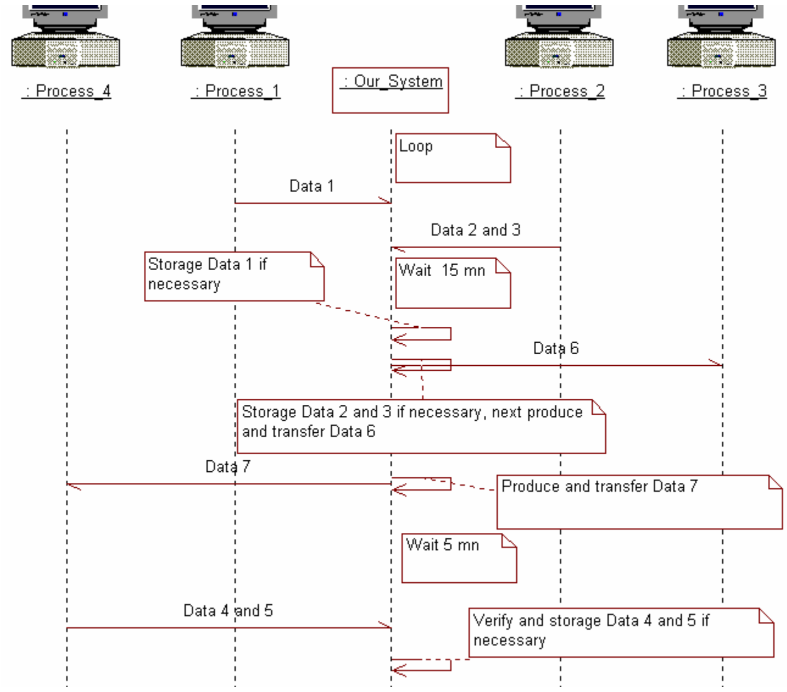
The static data flow is a unique collaboration diagram that illustrates our system with the external entities and the data, whether they are produced or consumed.



Static data flow example

The dynamic data flow is made up of one (or more) sequence diagram(s) that illustrate(s) the chronological relationship between the external entities and our system. It shows, in other words, when the system processes the input data and/or when the system produces and transfers the output data.

Important: *These diagrams are produced if the order of the data exchanged is important. Otherwise, they are not needed.*


: Process_4
: Process_1
: Our_System
: Process_2
: Process_3
Loop
Data 1
Data 2 and 3
Storage Data 1 if necessary
Wait 15 mn
Data 6
Storage Data 2 and 3 if necessary, next produce and transfer Data 6
Data 7
Produce and transfer Data 7
Wait 5 mn
Data 4 and 5
Verify and storage Data 4 and 5 if necessary

Dynamic data flow example

Activity C: System Description

This activity is composed of three "steps" (use case definition, domain analysis and scenario definition) and consists in finding the interactions between the active actors and our system. Each interaction is illustrated with a use case.

Definition: *An interaction is a direct communication between an active actor and the system, this communication embedding an active data flow.*

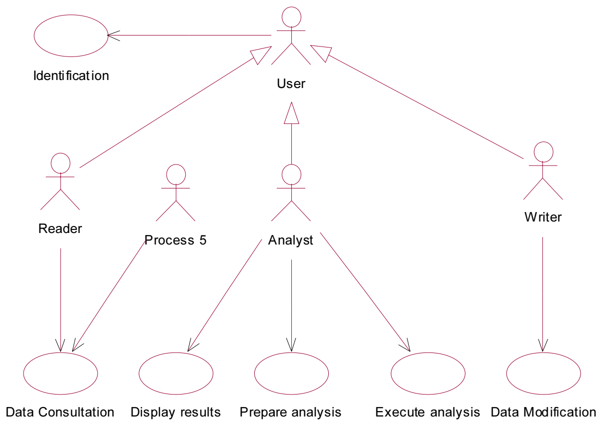
Step 1: Use Case definition

A use case is one essential thing for which the active actor uses our system. To find the use cases, the approach consists in answering the following question "WHAT WILL EACH ACTIVE ACTOR USE OUR SYSTEM FOR?"

Recommendations:

1. *Each Use Case is described in the final documentation as specified by a standard skeleton,*
2. *The role definition of the actors can be used to find the use cases.*

Note: *During this step, one or more use cases diagrams are designed, depending on the model complexity.*



Use Cases diagram example

Step 2: The Domain analysis

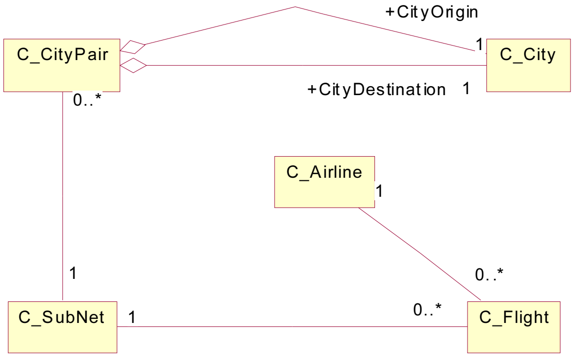
The domain analysis is a "business analysis" made through the use of "class diagrams". It allows the business classes to be found. The domain analysis must be **totally** independent of the solution used to implement it. To make this concept clear, let's imagine that the system to model is an electronic mail system. For this example, the domain analysis is the same as for the next two cases: a "postal mail" or the "pony express mail[1]". For example the class "ADDRESS" is without doubt a domain class in this context.

To find these domain classes, the most important thing to wonder is "IS EACH IDENTIFIED CLASS INDEPENDENT OF THE FUTURE SOLUTION?"

Recommendations:

1. *Each class is described in the final documentation as specified by a standard skeleton,*
2. *The domain analysis **must** be performed by both an analyst and a business expert, working together.*

Note: *During this step, one or more class diagrams are designed, depending on the model complexity.*



Domain analysis example

[1] Solution used in the "far west" during the 19th Century.

Note: *The Use Cases and the domain analysis are the entry points of the following step, namely the scenario definition*

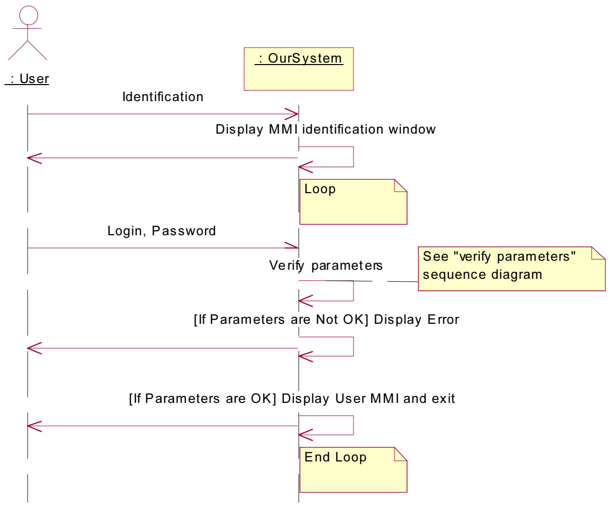
Step 3: The scenario definition

For each use case, one or more sequence diagrams are designed. There are two kinds of sequence diagrams:

(1) Some of them describe the interactions between the active actors and the system. They are called *functional sequence diagrams*. To build this kind of diagram, the most important thing to wonder for each use case is: "WHAT IS THE DATA FLOW BETWEEN THE ACTOR AND OUR SYSTEM IN THE USE CASE CONTEXT?"

Recommendations:

1. *Most of the time, each Use Case is associated with one and only one of these diagrams*
2. *Our discipline recommends to use of "**Notes**" attached to internal events or methods in the sequence diagrams. These notes indicate the next "scenario" or "use case" that will implement this event or method.*

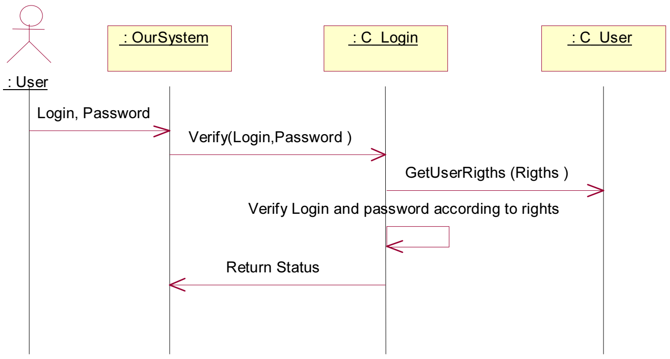

: User
: OurSystem
Identification
Display MMI identification window
Loop
Login, Password
Verify parameters
See "verify parameters" sequence diagram
[If Parameters are Not OK] Display Error
[If Parameters are OK] Display User MMI and exit
End Loop

Functional sequence diagram example

(2) The other sequence diagrams describe the collaboration between the objects (whether they are domain or application related) that implement the actors/system interactions. They are also called "level 2" sequence diagrams.

To build this kind of diagrams, the most important thing to wonder is: "HOW DOES THE SYSTEM IMPLEMENT THE INTERACTION BETWEEN THE ACTOR AND OUR SYSTEM IN THE CONTEXT OF THE USE CASE?"

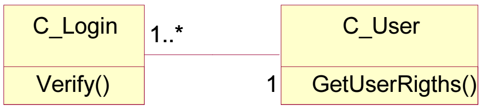
Recommendation: *Most of the time, there is one diagram for each internal event or method extracted from the functional sequence diagram.*



Verify parameters sequence diagram

This second type of sequence diagram **is the link** between the Use Cases and the class diagrams. Indeed, the secondary sequence diagrams are used to initialise the design of the unique class diagram associated with each use case.

To build such a diagram, the most important thing to wonder is: "WHAT ARE THE STATIC LINKS BETWEEN THE CLASSES COLLABORATING TOGETHER, IN THE CONTEXT OF THE USE CASE ?"



Class diagram example

Note: The "level 2" sequence diagrams and the class diagram of the use case are the entry points of the object analysis**. Moreover, this last step can be performed at the end of the requirement analysis or at the beginning of the object analysis.**

IMPORTANT: Naturally, if a class is active (it has a personal behaviour), it is not excluded to design a state diagram.

Object Analysis

The aim of this "Work Definition" is to produce a first logical organisation (using packages) of the classes previously found.

The object analysis "Work Definition" can be seen as one main activity.

Actors:	• The **analyst team**, still wondering "**What?**".
Inputs:	▪ UML Model ▪ MMI prototype ▪ Requirements Analysis Document ▪ Validation Plan Document
Outputs:	▪ UML Model updated ▪ Object Analysis Document
Activities:	▪ Activity D: Object Analysis

At the end of this "work definition", it is recommended to use the NEPTUNE checker, to check the object analysis rules.

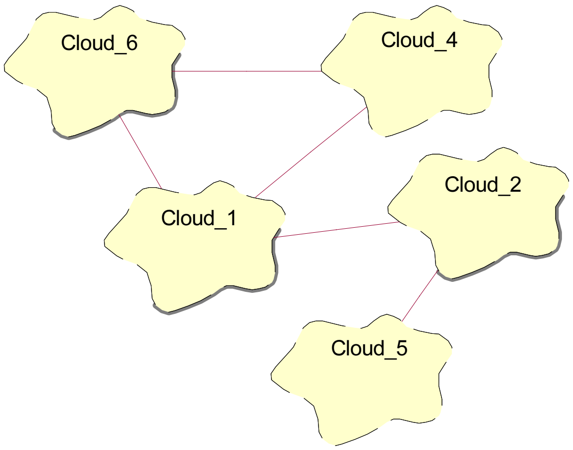
Activity D: Object Analysis

This activity first consists in designing as many class diagrams as use cases (if this was not made during the requirements analysis). Each class diagram contains all the classes involved in the Use Case. Then after, a first logical organisation is settled, using a set of packages.

To find this organisation there is NO ESSENTIAL QUESTION but a set of rules to follow. As examples:

- A package contains a medium set of coherent classes,
- A package contains a medium set of reusable classes,
- Dependency cycles between packages must be avoided,
- …

It is also possible to use an automatic rule to initialise this work. This rule consists in putting all the application classes in one diagram. Naturally this diagram (a dish of ravioli !!) is very hard to understand, but by moving and reorganising the classes (localisation in the diagram), it often appears "**clouds of classes**", only separated by a few links.



"Clouds of Classes" Example

Recommendation: *Each "cloud", can become a package containing all of the "cloud classes".*

Note: *At this stage ,the modeller may find one or more packages, depending on the complexity of the project.*

Once this organisation is effective, a class diagram must be created for each package. It contains all the classes of the package. Next, each class must be completed with its public attributes and the public methods, if these methods appear to be necessary. Of course, the diagrams can also be enriched by adding associations, inheritance...

This work is the last of the object analysis. Its level of achievement is left at the analyst appreciation.

If there is only one package at the end of this activity (object analysis), then the next step of the process is activity F. In any other case, the process moves to activity E.

Architectural Design

The aim of this "Work Definition" is to produce a first architectural view of the modelled system. In this "Work Definition", the object analysis packages are turned into logical components.

Actors:	• The **Design team**, wondering "**HOW?**".
Inputs:	▪ UML Model ▪ MMI prototype ▪ Requirements Analysis Document ▪ Object Analysis Document ▪ Validation Plan Document
Outputs:	▪ UML Model updated ▪ Architectural Design Document
Activities:	▪ Activity E: Software Component Definition ▪ Activity F: Software Component Description ▪ Activity G: Identification of Design Packages ▪ Activity H: Design Pattern Application

At the end of this "work definition", it is recommended to use:

- the NEPTUNE checker, to check the architectural design rules,
- the NEPTUNE document generator to generate the Interface Manual.

Important: *The first logical organisation made during the activity D can be modified here if the logical organisation does not respect the design constraints. For example: the Analyst has decided that a package contains three classes: an applet, an HTML page and a formulary about a particular work, this grouping is a logical choice. For the implementation, the Designer can decide a change, because for him the best solution is to group all the applets in the same package, all the HTML pages in other package ... This kind of change is naturally driven by the technical constraints.*

VERY Important:

1. *The E,F,G and H activities* ***must be made at the same time*** *the sequential presentation made in this book is just a writing facility,*

2. *The activity H is in fact a guideline to help the Designer to build the architecture.*

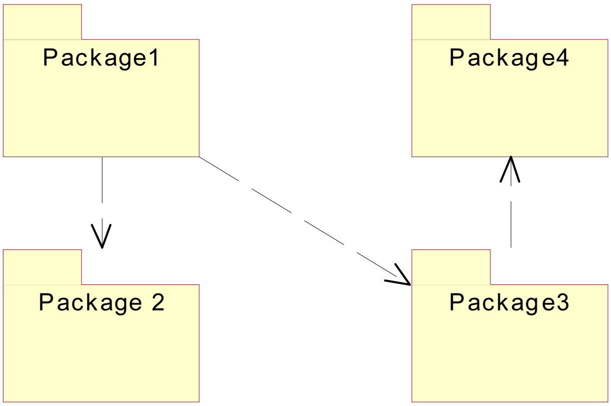
Essential: *This "work definition", with the step 2 of the activity C ARE THE MOST important ones and must consequently be performed with a very special care. The quality of the model, we could even say of the project, are here the real stakes. That's why* ***we recommend a very particular attention during this work.***

Activity E: Software Component Definition

This activity first consists in producing a package diagram showing the client-server view between all packages identified in the activity D.

To build such a diagram, the most important thing to wonder is: "WHAT ARE THE DEPENDENCIES BETWEEN OUR PACKAGES?"

Recommendation: *It is recommended to avoid dependencies cycles.*



Package diagram example

Once the package diagram is designed, time has come to find a **class interface** for each package, which is indeed, the identification of the provided services. Our discipline suggests three kinds of interfaces for different usage:

1. The "JAVA interface" always implemented by **one** concrete class. A package has one or more interfaces of this type.

2. The "FAÇADE interface" (a standard pattern [ref17], [ref9]) always implemented by **several** concrete classes. A package has only one interface of this type.

3. The "public view interface" implemented by all the public parts of all the classes located in the package. This type of interface is used in general when the package is a library.

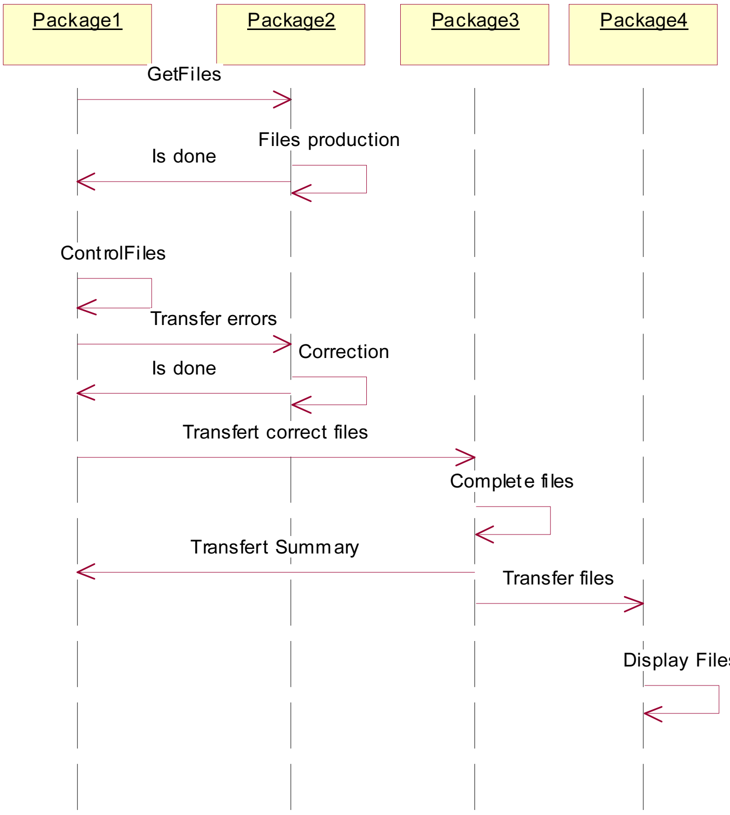
The last step of the Activity consists in identifying the interactions between the packages. This will highlight the exchanged data.

To show the interactions, **a package collaboration diagram** is created. It can either be a sequence diagram or a collaboration diagram. Whatever the choice, each object of the diagram will represent one of the packages.

To build such a diagram, the most important thing to wonder is: "WHAT ARE THE EXCHANGED DATA BETWEEN THE PACKAGES ?"

Recommendation: *It is recommended to make as many diagrams as necessary to explain the different collaborations between the packages.*

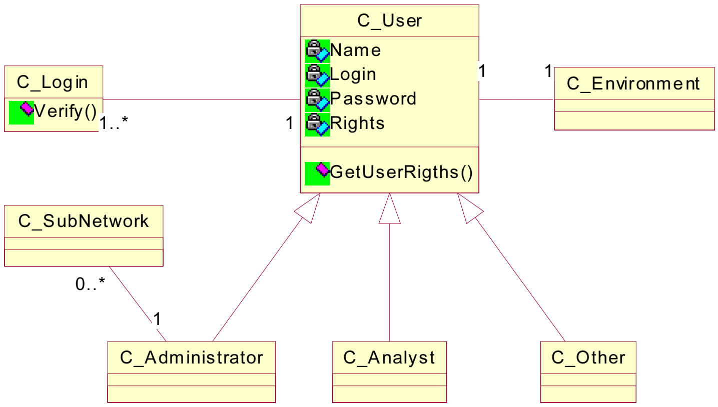
Note: *The choice to use the sequence or the collaboration diagram is made by the project team.*



Package collaboration diagram example

Activity F: Software Component Description

First of all, this activity consists **for each package** in completing its class diagram designed at the end of activity D. For example, if a package implements the use case "*identification*" (at least), we may complete its class diagram as following:



Completed class diagram example

Then, due to the potential introduction of services in the class interface defined in activity E, we may need to produce additional class and sequence diagrams. In that case they are created, still according to the spirit of the use cases implemented by this package.

To build these diagrams the most important things to wonder are: "HOW CAN WE RUN EACH PROVIDED SERVICE and WHICH CLASSES ARE INVOLVED?"

Note: E*ach method of the interface class is considered as a service.*

Recommendations:

1. *Each service is at least described by a pair of diagrams (class and sequence)*

2. *The classes can be completed by addition of public and private methods and/or private attributes.*

3. *At this point, It can be useful to add activity or state diagrams describing any collaboration between the classes by means of methods or events.*

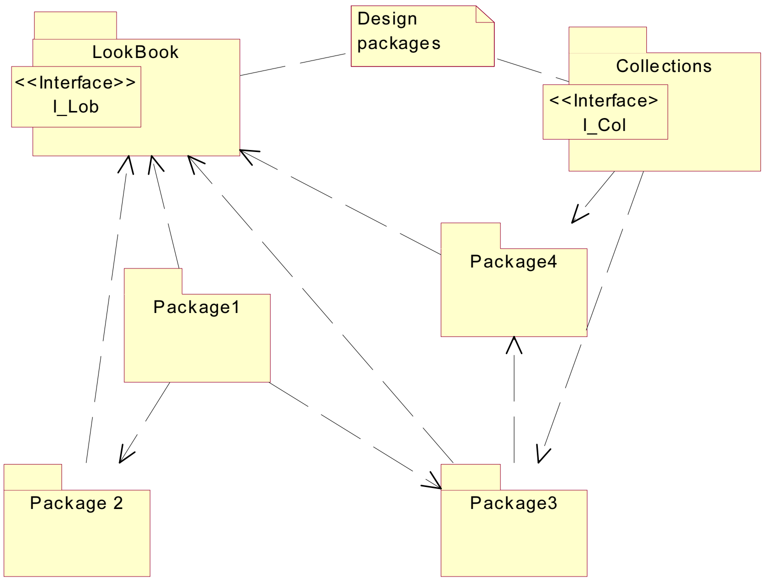
Activity G: Identification of Design Packages

This activity consists in finding the technical packages to complete the architecture by updating the diagram created during the activity E. These technical packages are:

- The existing packages (frameworks, business components…). In this case they are imported in our model.
- New packages that will be built during our work. In this case they are created in our model.

In any case, these packages MUST have a formal interface.

In the next example (the completed architectural diagram), we have added the "Look Book" and "Collections" packages and their interfaces "I_Lob", "I_Col".



Completed package diagram example

To build such a diagram the most important thing to wonder is: "WHAT ARE THE DESIGN PACKAGES (find out from the technical needs) AND THEIR DEPENDENCIES WITH THE EXISTING PACKAGES ? "

Activity H: Design Pattern Application

This activity consists in using patterns to ease the building of the architecture. It can be seen as a guideline. In our discipline, we use business patterns (see example 1 and [ref9]) from operational projects and standard patterns from the literature (see example 2 and [ref17]). The first work consists in analysing the identified problems to be solved. The next is the extraction from the pattern library of the ones that can (once applied) solve the problems aforementioned.

Note: *In a short future, NEPTUNE will be available to generate the source code from any pattern referenced in the tool.*

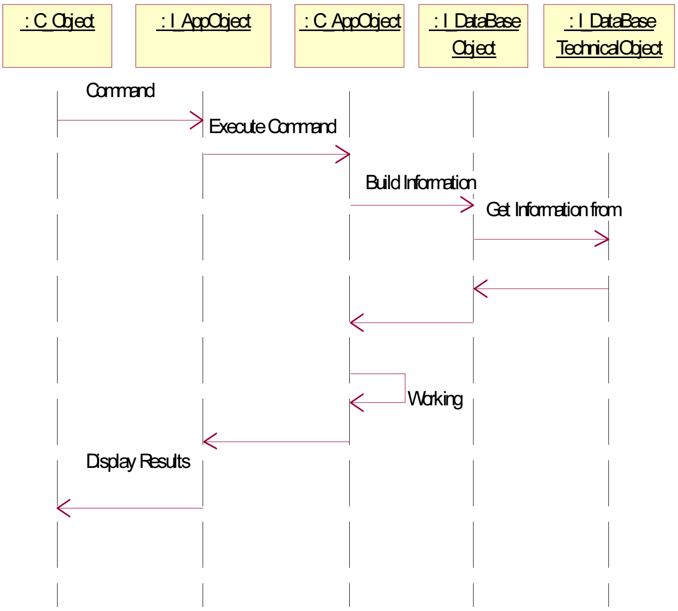
Recommendation: *It is recommended for each new defined pattern to update the "specific pattern catalogue" in use in the development environment.*

Example 1:

This pattern gives the different types of objects (and their collaborations) needed to build up a system composed of three parts known as the application, an MMI and a Data Base.

In the next diagram:

- C_Object is an MMI object,
- I_AppObject is an object interface between the MMI and the application,
- C_AppObject is an application object,
- I_DataBaseObject is an object interface between the application and the data base (e.g.: it allows to construct an "application data" from data base tables),
- I_DataBaseTechnicalObject is a technical object (e.g.: it executes an SQL line, it allows to connect (disconnect) to the database…).



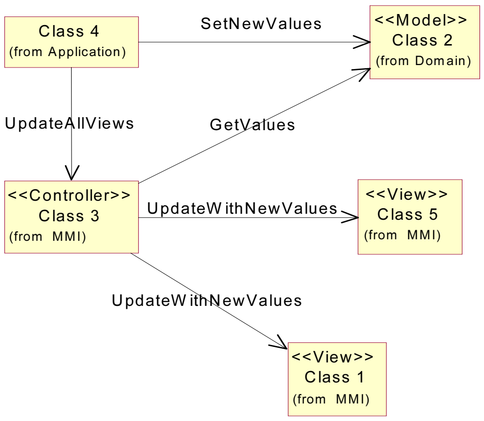
Architecture pattern example

Example 2:

This example shows an implementation of the Observer pattern [ref17] used in general by all the standard MMI libraries.

Note: *This pattern can also be used when the recurrent problem to be solved needs a subscription technique.*

An application class changes the model (*Set New Values*) and notifies the controller of this change (*Update All Views*). The controller then obtains the new values (*Get Values*) and modifies all views associated with this model (*Update With New Values*).



Observer pattern example

Note: *In this example, the different classes are allocated to three packages: "MMI", "Application" and "Domain".*

Object Design

The aim of this "Work Definition" is to produce for each package a detailed design (complete classes, attributes, methods, diagrams *etc.*) which will finalise the package description started in activity F.

Actors:	• The **Design team**, still wondering "**HOW?**".
Inputs:	▪ UML Model ▪ MMI prototype ▪ Requirements Analysis Document ▪ Object Analysis Document ▪ Architectural Design Document ▪ Validation Plan Document
Outputs:	▪ UML Model updated ▪ Detailed Design Document
Activities:	▪ Activity I: Classes Design ▪ Activity J: MMI Classes Design ▪ Activity K: Data Base Classes Design

At the end of this work definition it is recommended to use:

- the UML checker provided by the modelling tool,
- the NEPTUNE checker, to check the object design rules,
- the target language code generator provided by the modelling tool,
- the NEPTUNE code generator (in the future).

When these checks and code generations are executed without errors, time has come to compile in order to check the global interfaces … Then after, the methods can be completed according to the different diagrams.

Note*: In a short future, NEPTUNE will be available to generate the source code of the methods from sequence and activity diagrams.*

Activity I: Classes Design

This activity consists in completing the classes and the associations in order to prepare the code generation.

There is no question, recommendation or specific rules for this activity, but after it's been completed, the model must contain:

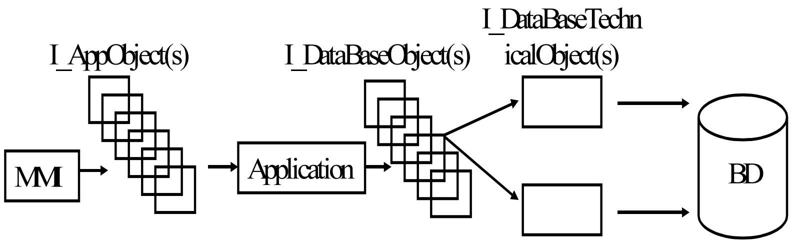
- For each class:
  1. all public and private attributes ,
  2. the type of each attribute,
  3. all public and private methods,
  4. the parameters of each method,
  5. the type of each parameter,
  6. And also (if necessary) for each method one or more sequence or activity diagrams,

- For each association:
  1. all chosen navigability's,
  2. all roles according to the navigability,
  3. all chosen multiplicity's,

- For each state diagram:
  1. its translation on classes, for example with using the pattern "state".

Activity J and/or K: MMI Classes Design and/or Data Base Classes Design

These activities consist in completing the work done in Activity I, and must be done at the same time. The next diagram illustrates the activities J and K according to the design pattern presented for the activity H in the first example.



MMI and DB interface example

In fact we **must complete** our set of application classes by the MMI classes,

- *The minimum is the identification of the interface classes between the MMI and the application* ***(I_AppObject).***

and the Data Base classes

- *The minimum is the identification of the interface classes between the application and the Data Base (**I_DataBaseObjects**).*

Physical Design

The aim of this "Work Definition" is to produce the physical architecture. It is composed of three activities known as α, β and ϒ, respectively for physical architecture description, identification of processes and components, and classes allocation.

IMPORTANT:
The physical design can begin at any activity of the discipline (or never if it is not necessary) However, we recommend a synchronous start with the

activities A for α, *D for* β and after I for ϒ (see the general figure that presents an overview of the discipline at the beginning of this chapter).

Actors:	• The **Physical Design team**, wondering "**WHERE?**".
Inputs:	▪ UML Model ▪ User Requirements
Outputs:	▪ UML Model updated ▪ Physical Validation Plan ▪ Physical Architectural Document
Activities:	▪ Activity α: Physical architecture description ▪ Activity β: Identification of processes and components ▪ Activity ϒ: Class allocation

At the end of this work definition it is recommended to use:

- the NEPTUNE checker to check the physical design rules,
- the NEPTUNE document generator to generate the "Test Cases Forms" from the produced diagrams, and specially from the sequence diagrams that show the processes collaboration .

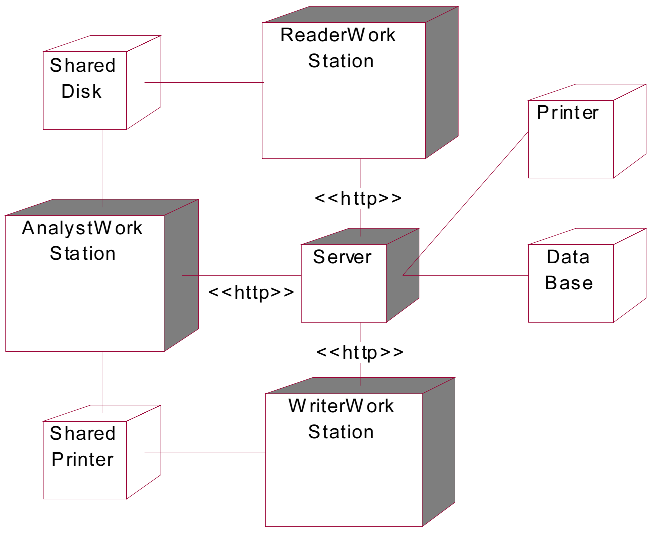
Activity α: Physical architecture description

This activity consists in building the physical architecture that will settle the organisation of the computers, devices, communication layers …

Note: *this concept is similar to the HOOD virtual nodes [ref15].*

Recommendation: The physical architecture is usually made - in its first version - parallel to activity A**.**

Important: *This synchronous start can help the Analyst to find actors.*



Example of a physical architecture

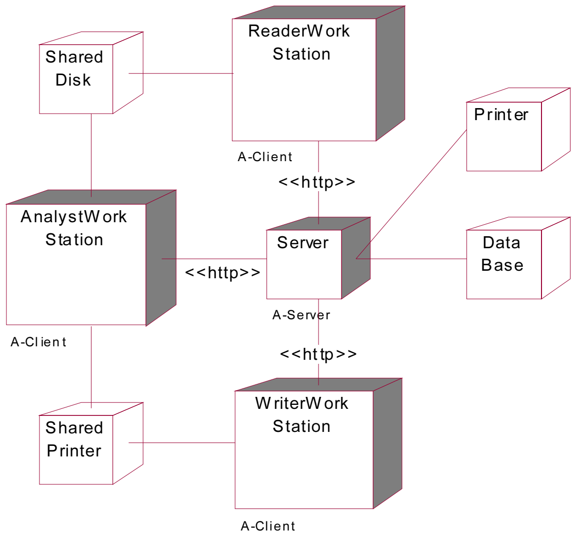
To build such a diagram, the most important things to wonder are: "WHAT ARE THE COMPUTERS AND DEVICES of our system, and WHAT ARE THE COMMUNICATION TECHNIQUES BETWEEN THEM ?"

Activity β: Identification of processes and components

For each computing resource, this activity first consists in identifying *all the processes,* in other words, the main programs of the application. These processes enrich the physical architecture defined in the activity α. Then after, each of the identified processes is split into several smaller entities called physical components. Component diagrams are used to do so.

This step of the process is another (one more) opportunity to complete the logical model.

Recommendation:
This identification is usually made - in its first version - parallel to activity D.



Example of a physical architecture with the processes (A-Client, A-Server ...)

To build such a diagram, the most important thing to wonder is: "WHAT ARE THE PROCESSES THAT RUN ON THE COMPUTERS?"

To ease this activity, it is also interesting to use sequence diagrams to show the collaboration between the processes. On these diagrams, each process is represented by one object, while the events describe the exchanged data. **Important:** The study of the exchanged data between the processes can be a great help for the analyst in finding new classes in the logical model.

To build these sequence diagrams, the most important thing to wonder is: "WHAT ARE THE EXCHANGES BETWEEN THE PROCESSES?"

Activity ϒ: Classes allocation

This activity consists in allocating the classes and/or the packages to its various physical components, this being done for each process.

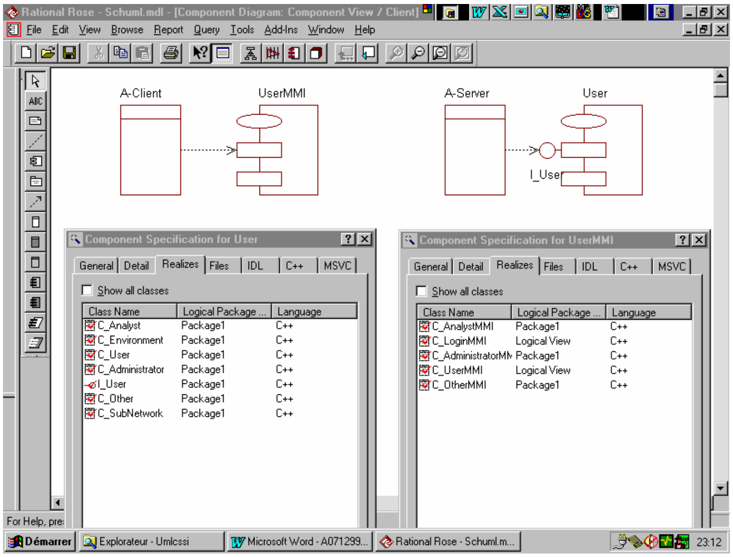
Recommendations:

1. For this work, the default approach consists in trying to map the logical components directly into the physical ones.

2. The classes allocation is usually made - in its first version -right after the class design activity,

3. Searching for the components content highlights the fact that some of them might possibly be implemented by the classes of the actual design while others might not. Among these ones, we will distinguish the components identified as existing frame works, COTS [ref14]…from the components that do not exist in the "shops", which a significant example is a configuration component.

4. *One or more classes can be allocated to each component.*


Rational Rose - Schuml.mdl - [Component Diagram: Component View / Client]
File Edit View Browse Report Query Tools Add-Ins Window Help
A-Client
UserMMI
A-Server
User
I_User
For Help, pres
Démarrer
Explorateur - Umlcssi
Microsoft Word - A071299...
Rational Rose - Schuml.m...
23:12

Component Specification for User

General | Detail | Realizes | Files | IDL | C++ | MSVC

Show all classes

Class Name	Logical Package ...	Language
C_Analyst	Package1	C++
C_Environment	Package1	C++
C_User	Package1	C++
C_Administrator	Package1	C++
I_User	Package1	C++
C_Other	Package1	C++
C_SubNetwork	Package1	C++

Component Specification for UserMMI

General | Detail | Realizes | Files | IDL | C++ | MSVC

Show all classes

Class Name	Logical Package ...	Language
C_AnalystMMI	Package1	C++
C_LoginMMI	Logical View	C++
C_AdministratorMM	Package1	C++
C_UserMMI	Logical View	C++
C_OtherMMI	Package1	C++

Example of classes allocation

To allocate the classes properly, the most important things to wonder are: "WHAT ARE THE COMPONENTS ISSUED FROM OUR DESIGN?"

"WHAT ARE THE COMPONENTS ISSUED FROM OUTSIDE?"

"WHAT ARE THE NEEDS OF EACH PROCESS, IN TERMS OF ENVIRONMENT?"

The component diagrams bring together the physical components coming from the logical design of the actual model and those from other models, frame works, files, libraries…

For many reasons, this work can later be updated, and in particular, during the integration and validation phases of the project. Consequently, the constraints appearing during the implementation lead the physical organisation away from what was originally the logical one, which should never be modified in the context of this activity. This is really important. The optimisation due to physical constraints is always performed on the classes and packages allocation within the physical design, and never on the logical design. Let's take the example of a process "P1" running on a computer "C1" and made of three components, "X", "Y" and "Z". If the execution timing of this physical organisation is not good enough, one solution consists in reallocating the "X" component to another process "P2" running on computer "C2". We naturally suppose that this reallocation implies a better performance of our system.

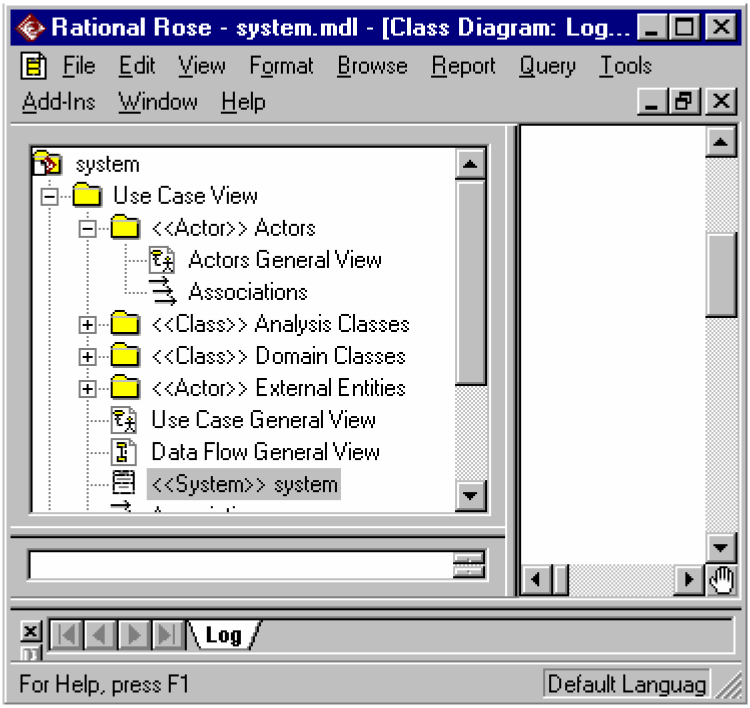
CHAPTER 4: ABOUT THE METHOD AND THE NEPTUNE SOFTWARE

Here below is a chapter showing how easy it is to slip from the NEPTUNE method to the use of the NEPTUNE software. We highlight the key points of the method that will make the use of the software tools even more efficient.

Additional NEPTUNE features for efficient modelling

The method presented in the previous paragraphs sets up the structure of the analysis and design model. As this structure hardly differs from one model to another, we thought that it would be interesting to take benefit of this frozen structure. To do so, we have developed a plug-in that allows the user to have the architecture pre-defined in its development environment [ref13]. This instrumentation allows the user to create a first browser architecture that will next be completed during the various steps of the modelling process.

Here after is a visual example of the use case view of a model (namely system) that has automatically been created with the plug-in aforementioned. Each of the packages or diagrams defined in the method as part of the use case view can be found in the created architecture.


Rational Rose - system.mdl - [Class Diagram: Log...
File Edit View Format Browse Report Query Tools
Add-Ins Window Help
system
Use Case View
<<Actor>> Actors
Actors General View
Associations
<<Class>> Analysis Classes
<<Class>> Domain Classes
<<Actor>> External Entities
Use Case General View
Data Flow General View
<<System>> system
Log
For Help, press F1
Default Languag

The method and the Checker

Our objective here is to focus on the benefits to be taken from the method for the use of the checker. A logical interpretation of this method presented in the previous paragraphs can lead to the identification of many rules, namely the methodological rules. As an example, one of these rules is: “The use case view must contain a package called analysis classes”. Many of these rules, though not formally expressed in the method, have anyway been identified, described textually and then after translated into OCL. These rules are available in the NEPTUNE checker. They can be applied to the model during the software lifecycle.

The method and the Documentation generator

How to take benefit from the method for the use of the documentation generator? This question has an answer in this paragraph.

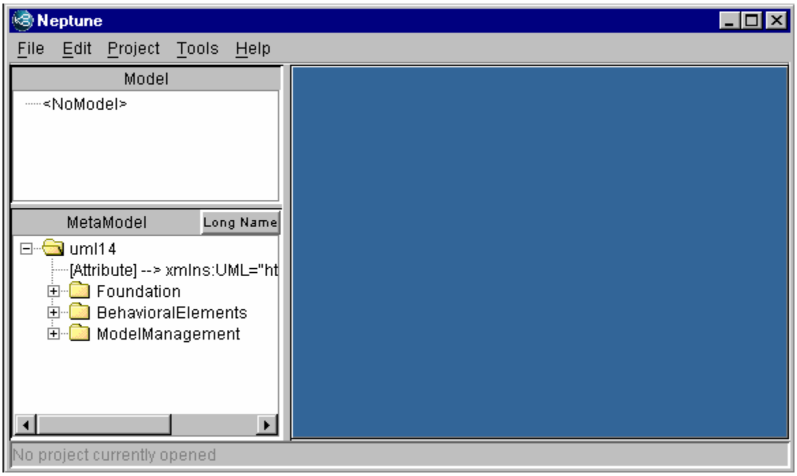
The recommendations given in the process, regarding the use of specific UML diagrams, but also regarding the use of XML format for the documentation associated, have been chosen so that they can be a great help for standard documentation generation. For example, in software engineering, the generation of a validation plan will be very easy with the document generator if the guidelines have been followed during the modelling process. Indeed, the templates used to generate the standard documents basically rely on the guidelines defined in the method. That makes another good reason for documenting the UML elements while modelling.

We have already mentioned the validation plan which generation template is available in the tool. It is also the case for the interface manual and other software engineering standard documents.

Getting started with NEPTUNE software

Within this chapter, the NEPTUNE GUI is presented in a general approach. The aim is to give a description of all the basic GUI features needed to work with the NEPTUNE software. As we have chosen a sequential description of what can be seen on the screen, and though several snapshots are supplied here after, it is even more efficient to run the software in parallel.

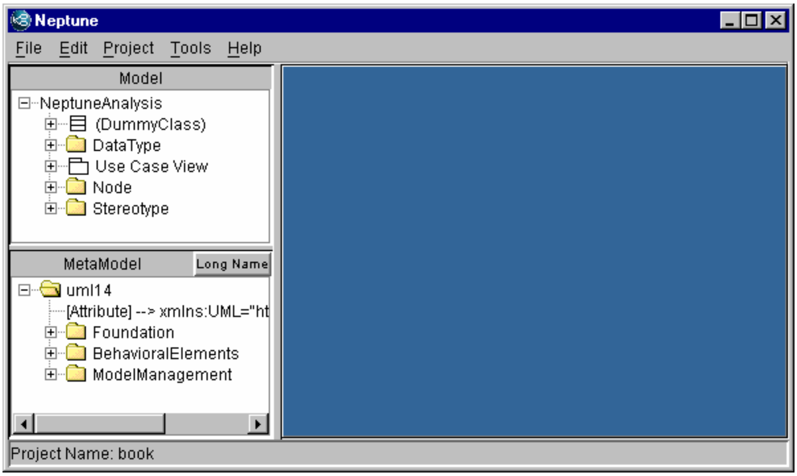
Once NEPTUNE is launched, the main frame is split into three different zones, as shown on the following snapshot.



The model browser (top left) aims at presenting a convivial view of the model (once it's been loaded). The meta-model browser (bottom left) is a representation of the UML meta-model. The last zone (the biggest on the snapshot) is the working zone, which role will be presented further on.

As with many other pieces of software, the highest level objects manipulated by NEPTUNE are the projects. Once a project has been created (use file popup), it can be filled up with a big amount of data. Among these data, the most significant is the model, as the NEPTUNE tools use it as their raw material. Consequently, after the project creation, the import of the model comes next. To attach it to the project, use the File popup again.

The next figure shows the model browser displaying the Neptune analysis model, after it has been attached to the actual project.

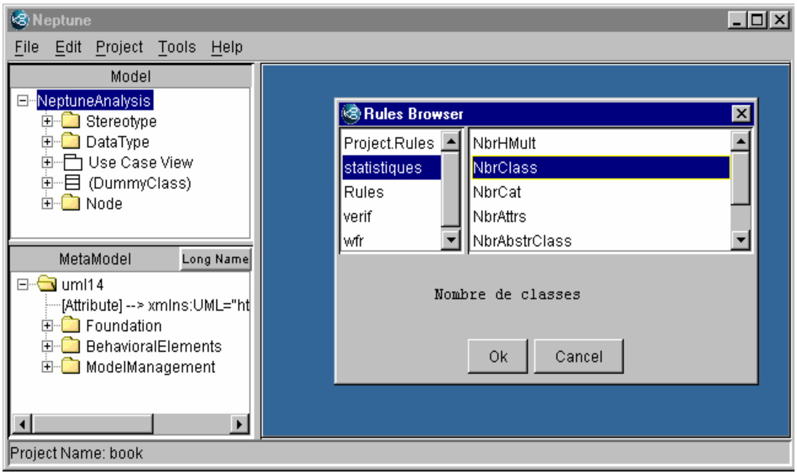


The creation of a project and the choice of a model constitute the starting point of any work done afterwards with the NEPTUNE tools.

What's next then? It depends on what the user wants to do, but whatever its choice, he has to manipulate some small objects, compared to the projects. These objects are the basic bricks necessary for running the checker or the documentation generator.

For checking purpose, the user will manipulate some OCL rules. They constitute the OCL expression of what has to be checked on the chosen model. A sub part of the model can be checked by application of one or more OCL rules, depending on the checking needs. To launch a check, once the user has chosen in the model browser the sub part he wants to check, he has to select a sub set of rules in order to apply them to its sub model selection.

The following figure is an example in which the whole analysis model is used as input of the rule used to count the number of classes.

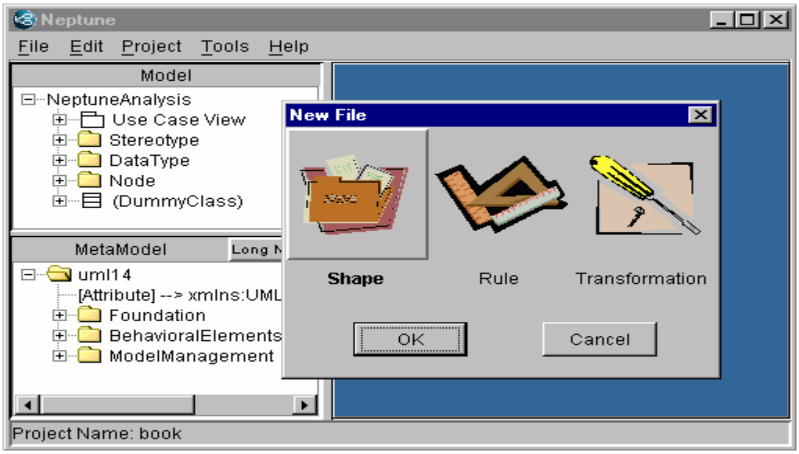


For documentation purpose, the NEPTUNE user can be led to deal with two different sorts of objects. The smallest ones are the XSL transformations. Their role is the extraction of some information from a model in order to transform it into textual information. To make a parallel, we can say that the XSL transformations are for the documentation generator what the OCL rules are for the checker.

Let’s talk now about the second aforementioned object, namely the shape: Several XSL transformations are often needed to achieve the extraction and transformation of the requested information. These transformations, put together and organised, constitute a kind of template. This is what we call a shape. Once processed by the NEPTUNE core, such a template is turned into a complete generated document.

NEPTUNE provides several sets of pre-defined OCL rules, XSL transformations, and shapes. Each object of these sets can be seen as an implementation of a standard need in terms of model checking or documentation generation. The NEPTUNE user can directly use this material. These sets can also be seen as sets of examples. These examples constitute a good basis and can be the starting points of brand new designs. Of course, it is possible to start designing an object from nothing.

The following snapshot shows the interface obtained on object creation request.



For a detailed description of the design features and GUI facilities, please refer to the chapters named “tool usage”. They can be found in part II and III, and are respectively dedicated to the checker and the documentation generator.

To conclude this first approach of the tools, it is important to mention the data storage. As any other software, NEPTUNE offers a backup feature. It is possible to save any opened project. This backup can be seen as contextual backup. In other words, when re opening a previously saved project, all the elements composing this project are opened again, the model, but also all the rules or transformations used inside of the project. Moreover, NEPTUNE also provides an individual backup for any of the elements that can be edited in the tool: rule, transformation or shape.

However, after a while it is possible to have in a project, a certain amount of rules, transformations and shapes that can be considered as generic or standard. It means that they could be used again in another context, in another project. To do so, the user can export these data to a global storage, and then import them into any different project.

PART II

Checking Models

Model checking has an important goal: detect incoherence early in the modelling phase to limit the impact in the development process. This chapter shows how the different kinds of checks can be performed with OCL over any model, and focuses on the different categories of checks needed.

CHAPTER 1: CHECKING PURPOSE

An UML model contains textual, graphical and formal information. There is not at the moment any link between those formalisms nor a recommended way to use them. This implies that the only insurance of the correctness of the model relies on the expertise of the modelling team and the exchanges with the client. Verification with the client cannot be avoided as the starting point of a modelisation is a textual document that no automatic tool can compare with the proposed model. The purpose of UML model checking is thus to offer a help to verify graphical and formal aspects of an UML model. After having defined the different objectives of model checking and the state of art concerning this field, we show how the recommended language OCL must be checked when used as a modelling language and can be used to express checking rules over UML diagrams. As an UML model is continuously evolving, check points must be established in conformance with a modelling process.

The worth of the UML model correctness

UML model correctness is certainly a very important aspect unfortunately ignored by many specialists of the modelling domain. A model is said to be correct if it fulfils the syntactic and semantic constraints of the modelling language. These constraints are defined by the UML specification proposed by OMG and take several forms. First the concrete syntax of UML is presented through the description of the visual form of nine diagrams. Second, the abstract syntax of an UML model is defined using UML class diagrams. It describes the links between UML modelling elements. Third, the usage of the UML modelling elements is restrained through natural language and OCL constraints. These constraints are then reinforced to take into account application domains through the use of methodological rules. The goal of Neptune is to check constraints expressed over the abstract syntax of UML, which can be imposed either by the UML notation or by a given methodology.

Checking UML Models – state of the art

The usage of formalisms such as script languages or formal languages (different from OCL) has some drawbacks. In order to be rigorous, we have to demonstrate for each WFR, the equivalence between its specification in OCL and its specification in the used formalism, supposing that WFR rules are correctly specified in OCL. Another problem, even more embarrassing, can appear when the checks are done using a UML CASE tool. This is because even in the best case, i.e. when the CASE Tool repository fully implements the UML metamodel, the tools do not allow the user to create any model elements or to declare certain relationships among the existent model elements. The most used CASE Tools – Rational Rose, Together, Poseidon, *etc.* have not yet implemented the Inheritance Relationship among packages, the Permission Relationship between packages (including standard stereotypes for this relationship), the Collaboration ModelElement, and other concepts defined in UML 1.4. Moreover, the Repository Interface for the above-mentioned tools is pretty different from the interface that would directly derive from the UML metamodel, completed with additional operations (AO) used to make easier the navigation inside the metamodel. Our position is that a minimal UML repository interface should include at least the AO and the set and get operations. Among the existent CASE Tools offering OCL support (see http://www.klasse.nl/ocl) neither Argo (Poseidon) nor Use or ModelRUN do provide user access to the tool repository by means of AO. Consequently they do not support UML model checking in a straightforward manner.

Checking UML models at different levels

The meaning of this title is very important in understanding the checking mechanism. The UML language uses two different formalisms: a graphical formalism and a textual formalism (OCL). The textual specifications are described in the ModelElement context. The correctness of the UML models thus implies the correctness of all kinds of specifications (graphical and textual). This is why the checking has to be done strictly respecting the above-mentioned sequence. The OCL constraints described at the model level are meaningful and must be checked only after the part of the UML model referred by these constraints has been checked against WFR.

CHAPTER 2: CHECKING CORRECTNESS OF BUSINESS CONSTRAINTS RULES

UML is not sufficient to model a system. The OMG has proposed the OCL to express constraints over model elements. OCL constraints are used either to specify invariants or behavioural properties of the system. This chapter presents the different contexts of OCL use and some extensions proposed by either NEPTUNE or UML 2.0 (though UML 2.0 extensions are not yet supported by NEPTUNE). Verification of such constraints must be understood at two levels:

- Business Constraints Rules are constraints over the dynamics of the system. They act as a partial specification of each dynamic view of the model, which incorporate target language implementation of methods and dynamic diagrams. Thus checking a model also means checking that its dynamics conforms to its (OCL) specification.
- Because they depend on the system being modelled, Business Constraints Rules evolve during the specification and design phases and can be syntactically or semantically incorrect as well as any piece of user written code. Thus, the syntax and the semantics of these rules must be checked.

The OCL syntax and semantics analyser of NEPTUNE can perform the first level of verification. However, the second level needs specific techniques: the validation of the target language code needs either run-time testing of assertions obtained after the compilation of OCL specifications, or theorem proving techniques to guarantee the full covering of execution paths. The checking of dynamic models needs temporal logic model checking techniques.

Consequently, the rest of this chapter will concentrate on the presentation of the different uses of OCL for specification of models.

Designing Business Constraints Rules (BCR)

By Business Constraint Rules we understand the rules expressed at the user model level. Consequently, they access model level information such as class attributes and association roles for navigation. The BCR have to fulfil some design principles. For example, a constraint attached to a class should not access private information of other classes. Otherwise, the fundamental encapsulation principle of object oriented programming would be violated. This principle must be imposed if constraints are used to produce runtime tests, but it is hard to follow in general because the expression of properties for verification purposes often needs to see hidden data.

Invariant and pre- and post- conditions

Invariants and pre- and post- conditions are the key features of the design by contract paradigm popularised by the Eiffel language and supported by formal development methods like the B method. These features are mandatory to allow the large-scale reuse of components. They appear in four major steps of software development:

- They define a guideline for components documentation. The interface of a component, a class in an object-oriented application, defines the operations it exports, the properties that are maintained during the life time of an instance (the invariant), the required conditions for an operation to be invoked (the operation pre-condition), and the enabled property after the operation has returned (the operation post-condition). These features are part of the component and of the operation signature.
- They can be used to help in the development of test suites. The formal specification of a component can serve at the automatic generation of test suites that will be run on the implementation. This point is an active research topic that surpasses the objectives of this book.
- They can be used as assertions inserted in the code of the application for run-time testing. This feature is used for maintenance purpose and allows the earlier detection of run-time errors.
- They can be used to develop a formal verification of the code of the application, thus reducing the need for unit tests. Such verification

can be automatic if the object behaviour is described by finite automata, or if the data model of the application is finite and static. Otherwise, proof techniques must be used, with partial human assistance.

In UML, invariant and pre-post conditions are OCL formula associated respectively with classes and with operations. They are thus part of the model and must be syntactically and semantically checked as any other modelling element. Consequently, the NEPTUNE tool allows the verification of the well-formedness rules (WFR) and the type conformance of OCL constraints that are part of the application. However, inter-formalism verification between OCL constraints and behavioural specifications are not in the scope of the NEPTUNE tool. Furthermore, such verifications are generally not possible. Consequently, we concentrate in the following on the description of the use of OCL for expressing invariants and pre-post conditions.

Class invariants

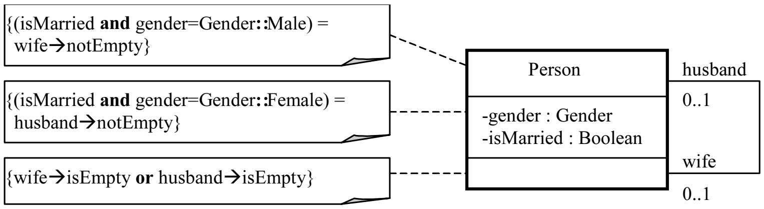
A class invariant is a property about class attributes expressing integrity constraints of class instances. Consequently, a class invariant must be true for each observable state of the whole life time of an object, which means that it is satisfied after the creation of an object and preserved by each public method. A class diagram describes structural invariants of object models. It includes typing constraints of attributes as well as multiplicity constraints over roles. When these notations are not sufficient, UML proposes two ways to attach an OCL-based invariant to a class: a graphical notation where an OCL constraint is linked to a class, and a textual notation in pure OCL where the invariant is specified within a *class* context.

For example, let us consider the class *Person* given in the OCL specification documentation. A person has a gender (male or female) and is possibly married. When married, a reflexive association designates her husband or his wife, depending on the person's gender. Several integrity constraints can be attached to the model and are either expressed graphically or via OCL formula:

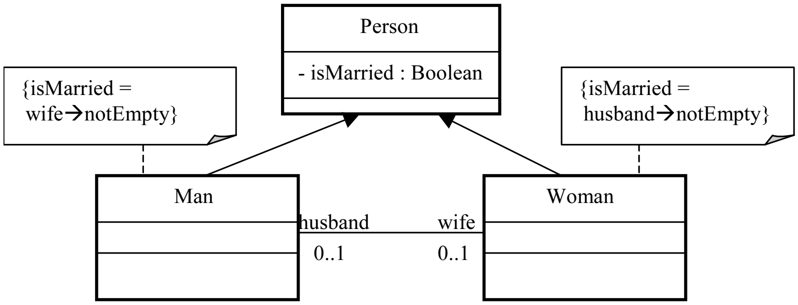
1. A person has at most one husband or one wife. This constraint is expressed by role cardinalities.
2. A married person has exactly one husband or one wife. This constraint is expressed by an OCL property.
3. A male has no husband and a female has no wife. These constraints are expressed by OCL properties.

4. The conjoint of a married person is also married. This property is in fact a consequence of the previous ones and need not be expressed.

The following class diagram defines the class *Person* with the attributes *gender* and *isMarried* and the reflexive association. The cardinalities of its two roles specify the first property. Constraints (2) and (3) are expressed using three OCL formula attached to the class *Person.* The first formula says that a person is married and is a male if and only if he has a wife. The second formula says that a person is a female and is married if and only if she has a husband. The third formula says that a person cannot have both a wife and a husband.


{(isMarried **and** gender=Gender::Male) = wife→notEmpty}
{(isMarried **and** gender=Gender::Female) = husband→notEmpty}
{wife→isEmpty **or** husband→isEmpty}
Person
-gender : Gender
-isMarried : Boolean
husband
0..1
wife
0..1

However, this is not the only possible model of the real world. In fact, the choice of a better representation can avoid the use of too many representation invariants. Here, it is possible to suppress the need for a gender attribute by creating two subclasses of *Person.* This is possible if we suppose that the gender of a person does not change. This is an important property of the model that is exploited by creating an inheritance hierarchy. Now, a person cannot have his gender dynamically changed. The association is no more reflexive but links a *Man* and a *Woman,* defined as the two subclasses of *Person.* This new design implies that the third property is implicitly satisfied. It remains to express the constraint checking the coherency between the value of the *isMarried* attribute and the association role. This is the purpose of the two constraints attached to the classes *Man* and *Woman.*



Pre- and post- conditions

Pre- and post- conditions are attached to an operation. They specify that if the pre-condition is satisfied by the caller then the execution of the operation will ensure the post-condition: pre-post conditions specify what is performed by an operation. The post-condition may make reference to the value assigned to features before the operation has been called. This *before* value is denoted by adding the suffix *@pre* to the name of the feature. There is no graphical notation for the attachment of pre-post conditions to operations. In fact, CASE tools propose text boxes to edit these specifications. On the other hand, pre-post conditions can be specified in pure OCL using the signature of the operation as context.

Let us take the previous example and try to specify the *GetMarried* operation in the class *Person* having the two attributes *gender* and *isMarried.* This operation takes as parameter another person. The pre-condition must say that both persons are not married and of different gender. The post-condition must say that the roles husband and wife designate the correct persons. It is useless to assert that the persons become married because this property is in the class invariant.

```
context Person::GetMarried(p: Person)
pre: self.gender <> p.gender and not
     self.isMarried and not p.isMarried
post: if self.gender = Gender::Male then
         self.wife = p and p.husband = self
      else self.husband = p and p.wife = self
      endif
```

The same operation can be specified in the framework of the second design. Getting the gender of the person is now performed by comparing the dynamic type of the person with *Man* or *Woman.* Then, a type conversion must be performed to access the value of the roles *wife* or *husband.*

```
context Person::GetMarried(p: Person)
pre: self.oclIsKindOf(Man) <> p.oclIsKindOf(Man)
     and not self.isMarried and not p.isMarried
post: if self.oclIsKindOf(Man) then
         self.oclAsType(Man).wife=p and
         p.oclAsType(Woman).husband=self
      else self.oclAsType(Woman).husband=p and
           p.oclAsType(Man).wife=self
      endif
```

Constraints over communications

Constraints over communications specify the messages that can be sent, and possibly in which order messages are sent by an operation or during the life time of an object. Contrarily to invariants and pre-post conditions, such constraints deal with how an operation is implemented and not only with its side effect or its result. A first step in this direction has been done by the UML2.0 proposal of message constraints which specifies the messages that are sent by an operation. As sequence or state chart diagrams are used to specify the behaviour of an operation or of an object, NEPTUNE proposes to use temporal logic [ref4] as a specification language for these diagrams. Temporal logic generalises invariants and allows to express that a property will become true in the future, or will be true infinitely often.

Messages

Message constraints are intended to be included in the incoming OCL 2.0 specification in order to specify properties over the messages that are sent by an operation. As for invariant and pre- and post- conditions, such constraints apply to the behavioural description of the modelled system, which can be defined through the use of sequence, statechart and activity diagrams, or using target language code. Two constructs are mainly introduced, that can be used in the context of an operation:

- **`r^op(args)returns`** true if the considered operation sends the message **`op`** to the object `r` with arguments matching **`args.`**
- **`r^^op(args)returns`** the sequence of messages of name **`op`** sent by the considered operation to the receiver `r,` with arguments matching **`args.`**

In both cases, unconstrained argument positions are identified by question marks. In the second case, a set of `oclMessage` is returned, which gives access to the actual arguments of the message, or to its return value, if any. It is thus possible to check constraints over the arguments of a message.

As an example, in a banking application, we can specify that a grant is always requested to the bank if the amount to be withdraw is greater that some given value.

```
context Client::withdraw(amount: Float)
post: amount > 1000.0 implies
      bank^request(self, amount)
```

Note that at that time, no tool exists either to check the satisfaction of message constraints by user models, or to check the well-formedness of these constraints. Currently, metamodel level constraints can be written to verify restricted forms of message constraints on UML dynamic diagrams. Furthermore, the power of message constraints make them impossible to be statically verified in general, but run time tests could be generated.

Temporal constraints

Invariants and pre-post conditions define business rules attached to elements of class diagrams of the system to be modelled. They constrain the dynamic behaviour of the system, which can be expressed using sequence, activity or state chart diagrams. However, the properties that can be expressed in such a way are somewhat restricted: a pre-post condition specifies the before and after states of an operation while an invariant specifies a property that must be satisfied by the state of an object after each operation. Thus, properties over the sequence of successive states of the system are limited to invariant assertions, and properties over intermediate states traversed during the execution of an operation cannot be expressed. Furthermore, the language used to express pre-post conditions and invariant make them impossible to verify statically in general. The purpose of our proposal is to overcome these negative points. It is based on the use of the temporal logic CTL (Computational Tree Logic) [ref4], to express for example that some state will necessarily be reached in the future or that a property remains always true.

The temporal logic CTL is a state-based logic, which means that atomic properties are predicates over states. It is tree-based because it allows the expression of properties about the different futures of a given current state, which defines a tree of reachable states. The two prefixes **A** and **E** express universal (for All) and existential (there Exists) quantification over the successors of the current state. Then we distinguish immediate successors (or next) states, designated by **X** and future states, designated either by **G** (Globally) or by **F** (Finally), depending on the quantification over the states contained in the considered execution path. Lastly, the binary operator **U** (Until) expresses that a property remains true until a second one becomes true. A temporal atomic property is build by the juxtaposition of a quantification (A or **E),** a path operator (F, **G** or **U)** and a state property. Such atomic properties can be assembled into state temporal properties using usual propositional logic connectors (conjunction, disjunction, negation, implication).

The CTL logic is used to specify safety and liveness properties. Safety properties express that something bad will not occur in the system while liveness properties express that something good will append. For example, a safety property for a traffic light controller is that lights of the two directions will not be green together. This can be expressed in CTL logic, which can be written: **AG not (ns_green and ew_green).** A liveness property may be that the traffic light will always become green in both directions: **(AF ns_green) and (AF ew_green).** This property can be weakened if we consider the presence of a presence detector: the traffic light will become green if the detector is activated: **AG (ns_detector implies AF ns_green).** Note that the **AG** prefix is useful because the property must not only be true at power on but at any time.

Temporal logic and OCL

The insertion of temporal logic to OCL supposes the definition of context declarations and of atomic predicates over states. These two notions, as well as the notion of state, will depend on the kind of dynamic diagram we want to constrain. On the other way, temporal connectors are independent of these choices and can be defined and linked to existing OCL constructs without making references to the target diagram. In order to make temporal formulas more readable, we have renamed the temporal connectors and supposed that the universal quantifier is implicit. Consequently, the following operations can be invoked on a state, with a specific syntax:

```
state-operation-call ::=
              Possibly ? Finally expression
            | Possibly ? Globally expression
            | Possibly ? expression until
              expression
```

which respectively mean:

- The expression will be satisfied by some state of all execution paths (some execution paths if *Possibly* is present) starting from the current state.
- The expression will be satisfied by all states of all execution paths (some execution paths if *Possibly* is present) starting from the current state.
- The first expression will be satisfied by the successive states, until the second one be satisfied, in all (or some) execution paths starting from the current state.

The following paragraphs will describe diagram dependent extensions, mainly the notion of state and atomic predicates over states.

Temporal constraints over sequence diagrams

UML 2.0 [ref5] provides a means to specify properties over the messages sent by an operation. If we can reason about the messages sent by one operation, it is not possible to reason about the whole sequence diagram, which describes a set of the messages exchanged by several objects. Temporal logic will allow the expression of such properties. For this purpose, we have to define what we intend to be a state in the context of a sequence diagram. In UML 1.4, a sequence diagram contains a partially ordered set of messages so that a message has a set of predecessors and a set of successors. The successor relation defines the tree-like structure of messages that may be sent after a given message. Temporal logic is used to specify properties over the possible future messages. For this purpose, we introduce atomic propositions over message attributes such as its name and the name of its source and destination. The transition relation is the successor relation partially ordering messages.

It has to be noted that in UML 2.0, a sequence diagram does not define a partial ordering over messages, but over sending or receiving events. Consequently, a state should be specified in terms of event attributes (is it a sending or a receiving event? What is the corresponding message name? What is the name of the source and destination objects?).

In the following, we conform to UML 1.4 semantics: a state is supposed to be defined using messages.Consequently, next states are immediate successors of the current message. In order to define predicates over such states, we must give access to some information about messages. For this purpose, the OCL library is extended with a **Message** class containing the following declarations:

```
class Message
{
 String name();     // name of the message
 String sender();   // name of the sender’s role
 String receiver();// name of the receiver’s
                   // role
 Set<Message> next(); // immediate successors
                     // of the current message
}
```

Finally, OCL context declarations must be extended to support the attachment of temporal constraints to sequence diagrams. The syntax of context declarations is modified as follows. It allows the attachment of a constraint to all sequence diagrams or to the named diagram.

```
contextDeclaration ::= context (operationContext
      | classifierContext | sequenceContext)
sequenceContext ::= SequenceDiagram : (* | name)
```

Within this context, the current object is a sequence diagram. The **messages()** operation must be called to obtain the first messages of the interaction. Graphically, a constraint over a sequence diagram is represented by a note attached to the sequence diagrams and containing the constraint enclosed within curly brackets.

For example, let us consider a banking application where transactions are described by sequence diagrams. We want to specify that a withdrawal is necessarily preceded by a successful authentication. This property applies to every sequence diagram and is specified by the following constraint expressed in the OCL extension we propose:

```
context SequenceDiagram:* inv:
            self.messages()
            ->forAll ((name()<>'withdraw' until
            name()='ack') and
            globally (name()= 'authenticate'
            implies(not(sender()='bank' and
            receiver()='client')
            until name()='ack')))
```

This constraint says that in every sequence diagram, a withdraw message cannot be sent by the client until he receives an acknowledgement and that the acknowledgement is the reply associated with the request of authentication. It could also be checked that for example a withdraw is emitted by the client and the acknowledge by the bank, but such checks are also performed by type-checking messages.

As a second example, we want to assert that in each sequence diagram of the application, one cannot start a car already started. This constraint can be specified as follows:

```
context SequenceDiagram::* inv StartOk:
     self.messages()->forAll (globally
            (not ((name='Start()' or name=
            'Drive()') and next (name=
            'Start()')))
```

If the current message is *Start* or *Drive,* the next message cannot be *Start.* We only consider three kinds of messages: *Start, Stop* and *Drive* and we suppose the existence of a rule verifying that a car can be driven only after a start.

Temporal constraints over statechart diagrams

Temporal logic states match with the notion of state defined by statechart diagrams, but can also be associated with action states of activity diagrams. Their common metamodel allows the use of the same language to constrain both kinds of diagrams. As for sequence diagrams, states, next states and predicates over states must be defined. A difficulty comes here with the existence of concurrent states and composite states. The state of the system is in fact defined by a so called *configuration* which contains the set of states in which the system can be. In presence of concurrent states, the states of all the components must be mentioned. This only atomic predicate that the user can call is the `oclInState()` predicate which checks that the state given as argument is within the current configuration. The transitions from that state designate the possible next states, which are returned by the `next()` operation.

```
class OclConfig
{
    Boolean oclInState(oclState st);
    Set<OclConfig> next();
}
```

As for sequence diagrams, a new context declaration is introduced to allow the attachment of constraints to all or to named statechart diagrams. Within this context, the current object *self* designates the initial configuration of the selected diagram, containing its initial state. A graphical notation is also introduced, which consists in attaching a note containing the constraint to the initial state of the diagram.

For example, consider a traffic light controller managing two concurrent sub-systems (ns and ew) associated with the two directions, each defined by three states, corresponding to the three usual colours. The safety property specifying that subsystems cannot be in the green state together can be expressed as follows:

```
context StateDiagram::trafficControler
inv safety: globally not (oclInState(ns::green)
            and oclInState(ew::green))
```

Business Constraints Rules Correctness

Business constraints rules apply to dynamic views of the user model, which may be either dynamic diagrams or target language code. Consequently, it is not always possible to verify the coherence between BCR and other parts of the model, and notably if target language code is concerned. However, these rules can be inserted as assertions into the generated code for testing purpose. Thus their syntactic and semantic correctness must be checked. The NEPTUNE tool provides a mean to perform these verifications.

CHAPTER 3: CHECKING UML MODELS

Contrary to chapter 2, this chapter concerns the verification of the structure of UML models. The properties that are checked here are independent from the user defined model, but are linked to the definition of the UML notation itself. Such properties are not expressed at the model level, but at the metamodel level as metaclass invariants. This chapter is concerned with the verification of constraints having three main sources: constraints provided by the OMG, as the UML WFR (mainly intra-diagram rules), inter-diagram rules extending this set of rules and finally, target language dependant rules and methodological rules.

Design principles

The constraints expressed at the metamodel level offer users the opportunity to specify profile rules (rules depending on the kind of problem domain, target programming language, *etc*.). These kind of rules are presented by means of examples.

The UML Metamodel as support for expressing constraints over models

This paragraph focuses on the use of the UML metamodel to define OCL constraints specifying the well-formedness rules of UML models. The meta level allows the definition of universal rules, *i.e.*, rules that are satisfied by any instance of a metaclass. They are expressed using OCL invariants attached to metamodel classes.

Metamodel Navigation and Additional Operation

The UML Additional Operations (AO) have different roles. Firstly, it is very important that OCL specifications are suggestive and easy to understand. Taking into account the AO specification complexity, it is not difficult to guess that without using AO, the WFR understanding will be significantly diminished. For example, some AO return the set of model elements that are visible from a given context and take into account protection and (recursive) importation mechanisms. The second aspect is directly connected to the reusability. The major part of AO is used in specifying several rules. Consequently, AO are reusable entities used to make easier the writing of WFR. Furthermore, replacing direct metamodel navigation by AO calls make WFR less dependent of evolutions of the UML metamodel.

Well-formedness rules

The UML notation is specified by its metamodel that defines the syntax of the language. It is completed by a description of its static semantics that restricts the set of legal constructs. The static semantics is defined by constraints over the metamodel expressed both in natural language and in OCL. This set of constraints is usually called well-formedness rules. A minimal set of well-formedness rules is normalised and proposed in the UML semantics document. They mainly concern intra-diagram rules. The purpose of the following section is to present some extensions of this set of rules, as they are implemented in the NEPTUNE checker. These rules are either general purpose or depend on the use of the UML notation, as defined by an UML profile.

Inter diagrams coherence rules

An UML model consists of several views described using different types of diagrams. Verifying the coherence of the model is supposed to address the problem of inter-diagrams verification. We study several couples of diagrams and how to express their relationships with OCL at the metamodel level. The following table summarises some of the inter-formalism checks that could performed.

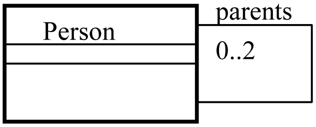
		Checking performed
Class diagram	**Object diagram**	Connections between objects must be compatible with associations
Class diagram	**Sequence diagram**	Messages must be operations or signals of the class of the receiver. The receiver must be known by the sender
Class diagram	**Statechart diagram**	Actions and guards should be valid with respect to the class model.

Sequence diagram	**State chart diagram**	Sequence diagrams should describe behaviours allowed by state chart diagrams. Sent events must be defined in the state chart of the receiver class.

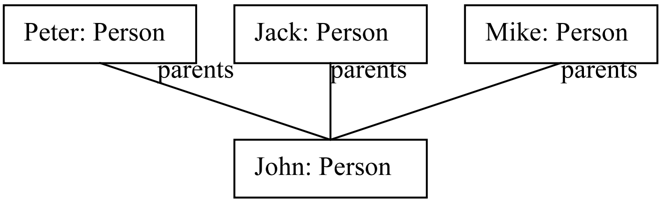
Class and Object diagrams

Object diagrams are not widely used in UML because they are not supported by most UML CASE tools and can often be replaced by collaboration diagrams which also represent class instances and relations between instances. Some coherence checks have to be performed between object diagrams and class diagrams. In fact, class diagrams describe properties that must be verified by their instances. For example, associations between classes correspond to links between their instances. Thus, we can verify that if there is a link between two instances, an association is present between their corresponding classes. We can go one step further if we look at role cardinalities. However, the verification cannot be complete because object diagrams are partial representations of the set of existing objects. Thus, it is only possible to check that the number of links starting from an object is not greater than the upper multiplicity declared in the corresponding association role.

For example, let us consider the class *Person* with an association role named *parents* connecting a person to his/her live parents. The cardinality of this role is thus 0..2. A collaboration diagram can represent one person, say John, and his two parents. An error will be signalled if John is linked to three persons.



Class diagram



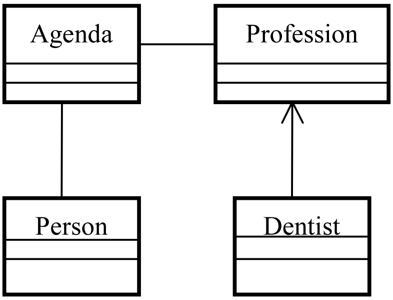
Invalid Collaboration diagram

Class and sequence diagrams

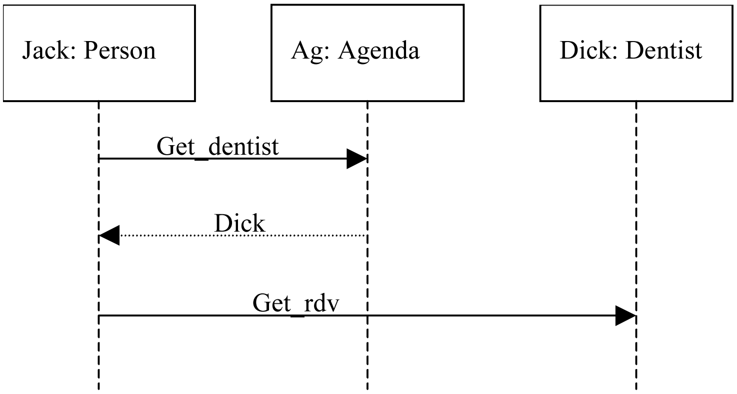
Sequence diagrams describe messages exchanged between class instances. Consequently, they are strongly linked to class diagrams and their coherence should be checked. Messages exchanged between instances are either asynchronous events or operation calls. This leads to two verifications: signals must be accepted by some transition of the statechart associated with the destination class; called operations must be declared in the interface of the class of the destination, or inherited. It is possible to verify that the name of the message is that of an operation. However, such a verification must not be systematic: in analysis phases, the analyst may want to give a textual description of the message. The message becomes an operation class in more advanced phases. This verification is thus dependent on the analysis and design methodologies.

A second verification concerns the destination of sent messages. It must be known by the sender. Its direct knowledge depends on the associations declared in the class diagram: an object can send a message to an instance of a class connected by an association to the class of the sender. This condition can be weakened if messages already sent are taken into account: the destination is valid if it is directly known or if its name may have been communicated by previously received messages.

For example, let us consider the class *Person* with an association to the class *Agenda* containing the phone number of his dentist. An instance of *Person* must first send a message to an instance of *Agenda* before contacting an instance of *Dentist*. An immediate communication with a dentist would be invalid. Such an incorrect behaviour can be detected.



Class Diagram



Sequence Diagram

It has to be noted that the exact contents of messages such as parameters and return values are not taken into account because it is generally not possible to know which information has really been communicated to the destination. Thus, the identity of all the instances of a class is supposed to be potentially transmitted upon message exchange.

Class and statechart diagrams

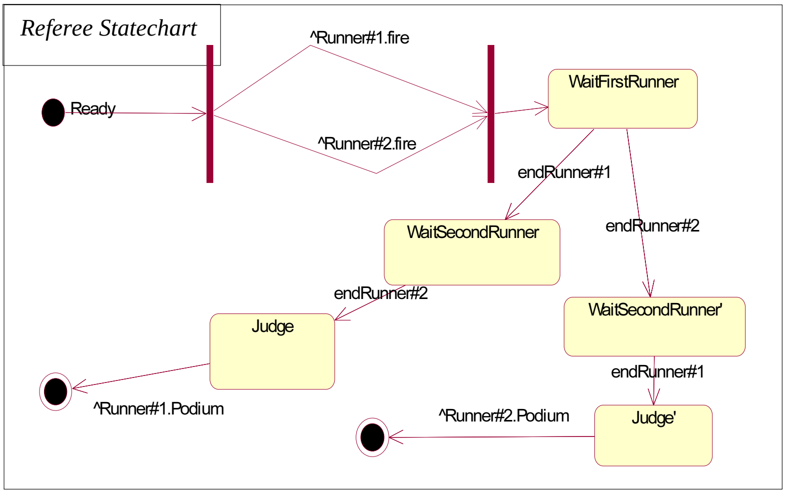
Statechart diagrams define states and transitions, both being annotated by actions or guards. It should be possible to verify the syntactic and semantic correctness of these annotations. More precisely, it has to be checked that expressions contained in actions and guards are first legal with respect to the target language and second that they are semantically correct with respect to class diagrams. However, this verification is hard to perform because guards and actions can be described in any language. This situation will evolve with the diffusion of the UML action language which specifies the abstract syntax of a platform independent language. Consequently, for the moment, the coherence between actions and guards of state/transition diagrams and class diagrams will only be checked at compilation time and no support for this activity is provided by NEPTUNE.

Sequence and statechart diagrams

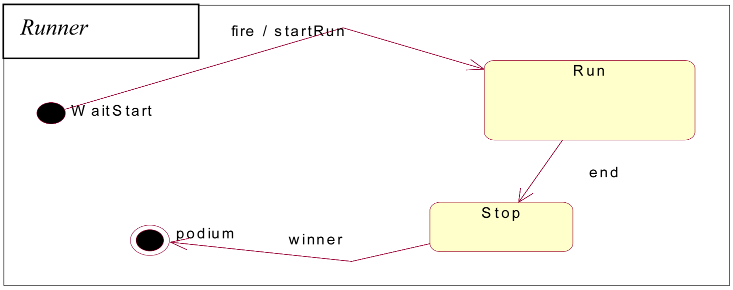
Statechart diagrams describe all legal behaviours of the system while sequence diagrams describe possible interactions between objects of the system. The sequence of messages exchanged should be valid with respect to statechart specifications. Once again, this verification cannot be complete because statechart describe guarded transitions. It is generally not possible to verify these conditions before an effective execution of the application. In fact, this verification requires simulation techniques which are hard to implement in a pure functional language such as OCL.

For example, let us consider the two classes *Referee* and *Runner*. The *Referee* sends both the two *runners* the *start* signal of the run, waits for each runner and sends the winner the order to go to the *podium*. On the other side, the *Runner* waits for an event *fire,* then starts to run until the end. Then the runner can go to the podium if he receives the event *winner*.

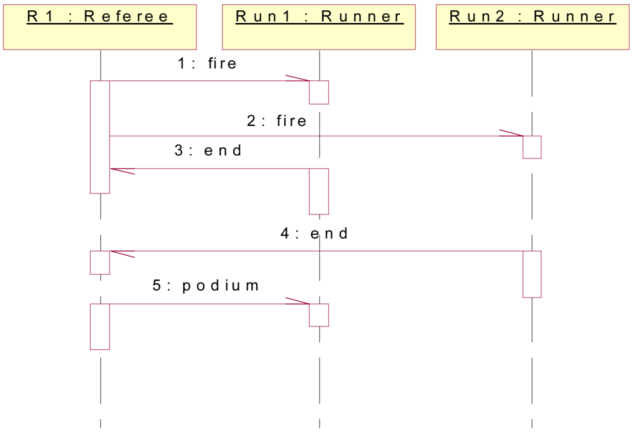
The next statechart diagram specifies the behaviour of the referee. For the sake of simplicity, only two runners can be managed. Furthermore, forks and joins are used to avoid sending two events in the action part of the transition.



The next statechart diagram specifies the behaviour of each instance of the class *Runner*. It is simpler: a runner waits for the *fire* event and performs the *startRun* action. When he crosses the line, he sends the *end* signal and waits for the podium signal.



Now, the next sequence diagram, proposed as an example of the behaviour of the system, must be checked with respect to the effective behaviours specified by the previous statechart diagrams.



Scenario of the run given as Sequence Diagram

Checking for such properties is in fact complex to write in OCL and needs the writing of a simulator of finite state machines. The complexity is reinforced by the incomplete implantation of sequence diagram by the different UML tools and by the existence of various representations as metamodel instances of the graphical notions from one tool to another. On the other hand, a simpler necessary condition can be checked: messages send to the destination that hold signals must be handled by the statechart of the receiver class. In the previous example, all the exchanged messages of the sequence diagram are signals and must be present in the trigger of one of the statecharts.

Profile rules

This paragraph focuses on the use of new stereotypes and patterns, introduced for the benefits of software engineering and business process modelling. Some of the rules constraining the usage of these stereotypes are given.

For example each profile must be described in such a manner:

Stereotype	Base class	Parent	Constraints
Event	Class	NA[2]	Restricted generalisation. No aggregation
Action Event	Class	Event	Restricted generalisation. No aggregation
Communicative Action Event	Class	Action Event	Restricted generalisation. No aggregation
Non-Communicative Action Event	Class	Action Event	Restricted generalisation. No aggregation
NonAction Event	class	Event	Restricted generalisation. No aggregation
Commitment Claim	Class	NA	Restricted generalisation. No aggregation
does	Association	NA	The domain class must be an agent type and the range class must be a non-communicative action event type. Multiplicity is one-to-many.
Perceives	Association	NA	The domain class must be an agent type and the range class must be a non-communicative action event type or a non-action event type. Multiplicity is one-to-many.
Sends	Association	NA	The domain class must be an agent type and the range class must be a communicative action event type. Multiplicity is one-to-many.
Receives	Association	NA	The domain class must be an agent type and the range class must be a communicative action event type. Multiplicity is one-to-many.
HasClaim	Association	NA	The domain class must be an agent type and the range class must be a commitment/claim type. Multiplicity is one-to-many.
HasCommitment	Association	NA	The domain class must be an agent type and the range class must be a commitment/claim type. Multiplicity is one-to-many.

Software engineering profile (Formerly Methodological rules)

This paragraph focuses on the definition of the list of new stereotypes introduced by the NEPTUNE methodology for the benefits of software engineering. Each stereotype is described by a set of constraints expressed both in natural language and in OCL.

[2] Not Applicable

Stereotype	Base Class	Parent	Methodology Phase
Constraint (Natural Language + OCL formulation)			
System	Class	NA	"A"

A class stereotyped 'System' must exist
package Foundation::Core
  context Package inv system_exists:
        name='Use Case View' implies contents->exists(c |
        c.oclIsKindOf(Class) and c.oclAsType(Class).stereotype->
                exists(name='System'))
endpackage			

System	Class	NA	"A"

There must be a unique class with stereotype 'System'
package Foundation::Extension_Mechanisms
  context Class Inv Unique_System_Class:
        (name='System' and baseClass = 'Class') implies
        extendedElement->size = 1
endpackage			

System	Class	NA	"P3"

All the operation of the stereotyped Class 'System' are described by an Activity Graph
package Foundation::Core
  context Stereotype Inv SystemClassOpSpec:
        Self.stereotype->exists(name = 'System') implies
        self.allOperations->forAll(op |Op.behavior->
                exists(oclIsKindOf(ActivityGraph)))
endpackage			

System	Class	NA	“B”

All entities of the static data flow diagram must exclusively communicate with the system

```
package Behavioral_Elements::Collaborations
  context Interaction inv Static_data_flows_to_System
          name='{Use Case View}Data Flow General View' implies
          message->forAll((sender.base.stereotype->
                    exists(name='System') or receiver.base.stereotype->
                    exists(name = 'System'))
endpackage
```

Actor	Package	NA	“A”

A package named 'Actors' must exist in the Use Case View

```
package Model_Management
   context Package inv Main_actors_exists:
          name = 'Use Case View' implies contents->
                    exists(oclIsKindOf(Package) and name='Actors' and
                    oclAsType(Package).stereotype->exists(name =
'Actor'))
endpackage
```

Actor	Package	NA	“A”

The 'Actors' package only contains actors

```
package Model_Management
   context Package inv Main_actors_contents:
          (name = 'Actors' and stereotype->exists(name='Actor')) implies
          contents->forAll(oclIsKindOf(Classifier) implies
                    oclIsKindOf(Actor))
endpackage
```

Actor	Package	NA	“A”

A package of passive entities must exist in the UseCase View

```
package Model_Management
context Package inv Passive_entities_exists:
          name = 'Use Case View' implies contents->
                    exists(oclIsKindOf(Package) and
                    name='External Entities' and
                    oclAsType(Package).stereotype->
                              exists(name = 'Actor'))
endpackage
```

Actor	Package	NA	“A”

The "External entities" package only contains classifiers stereotyped as 'External Entity' or 'Other System'

```
package Model_Management
  context Package inv External_entities_contents
          name = 'External Entities' implies contents->
                    forAll(oclIsKindOf(Classifier) implies
          (stereotype->exists(name = 'External Entity' or
          name = 'Other System')
endpackage
```

StaticDataFlow	Collaboration		

A static data flow diagram exists in the UseCase View

```
package Model_Management
   context Package inv DataFlow_General_View_exists
          name = 'Use Case View' implies contents->
            exists(oclIsKindOf(Collaboration) and
          oclAsType(Collaboration).interaction->
            exists(name = '{Use Case View}Data Flow General View'))
endpackage
```

Actor	Package	NA	“B”

The static data flow diagram must contain all passive entities and the system

```
package Model_Management
   context Package inv Static_data_flow_ok
       let passive =contents->select(oclIsKindOf(Package) and
          name='External Entities').oclAsType(Package)->
          select(stereotype->exists(name='Actor'))

       let static =contents->select(oclIsKindOf(Collaboration)).
          oclAsType(Collaboration) ->select(interaction->
          exists(name = '{Use Case View}Data Flow General View'))
  in name = 'Use Case View' implies (static.contents->
          select(oclIsKindOf(ClassifierRole)).oclAsType(ClassifierRole).
             base->asSet->includesAll(passive.contents->
             select(oclIsKindOf(Classifier)).oclAsType(Classifier)->asSet)
endpackage
```

Facade Library	Class	NA	“E”

A package which is not stereotyped 'library' must have an interface class

```
package Model_Management
   context Package inv PackageInterface
            not (stereotype->exists(name='library')) implies self.contents->
               exists(oclIsKindOf(Class) and stereotype->
                       exists(name='facade'))
endpackage
```

Facade Library	Class	NA	“D”

An association between classes belonging to two different first level packages is allowed only if classes are stereotyped 'facade' or 'interface', unless their package is stereotyped 'library'

```
package Foundation::Core
  context Association inv CrossPackageAssocsRestriction
     let exported(p:Namespace): Set(ModelElement) = p.contents->
           select(oclIsKindOf(Interface) or  (oclIsKindOf(Class) and
          stereotype->exists(name='facade')))
     let ns(me:ModelElement): Namespace =
          me.elementOwnership.namespace
     let path(me:ModelElement):Sequence(Namespace)=
          if me.oclIsKindOf(Model) then Sequence{}
          else path(ns(me))->append(ns(me))
          endif
     let enclosing(me:ModelElement): Set(Namespace) =set{ns(me)}->
          closure(oclAsType(ModelElement).elementOwnership.
          namespace)->including(ns(me))
     let visible(p:Namespace,c:Classifier) = enclosing(c)->includes(p)

     let roots(o1:ModelElement, o2:ModelElement):
          sequence(Namespace) =
            if ns(o1) = ns(o2) then Sequence{ns(o1),ns(o2)}
            else
               sequence{Sequence{1..path(o1)->size.min(path(o2)->
                    size)}->any(i | path(o1)->at(i)<>; path(o2)->at(i))}->
                      collect(i | Sequence{path(o1)->at(i),path(o2)->at(i)})
            endif
     let check(r: Sequence(Namespace), m1: ModelElement, m2:
          ModelElement) = r->at(0) = r->at(1) or (exported(r->at(0))->
              includes(m2) and exported(r->at(1))->includes(m1))
  in connection->forAll(c1,c2 | c1 &lt;&gt; c2 implies check(roots
          (c1.participant, c2.participant), c1.participant, c2.participant))
endpackage
```

Business process profile

A business rule is defined as “a statement that defines or constrains some aspect of the business” [ref6].

Business rules are in the heart of the production of correct models for the business, and in consequence, in the production of good supporting IT systems.

By uncovering the rules behind the scene, and formulating them in such a way that a tool can take the model and check it against them, it is possible to detect in the very first phases of the process critical errors that otherwise would propagate to the design and implementation phases. This leads to less development time and more suitable IT systems.

In a previous chapter we have shown techniques for identifying business rules. This section shows some rule patterns identified after applying those techniques to the analysis of several payment processes within a real organisation. They are part of a more complete repertoire of patterns that can be easily applied to formulate business rules for and to check the corresponding UML activity diagrams modelling the details of business processes. Similar patterns could also be defined for other views of the business modelling after the corresponding work of identification and abstraction.

Rule patters are shown in the tables below. For each one, a business rule example and an OCL expression are given.

Rule pattern: A certain guard must be present in all the existing paths between two points of an activity diagram
Business rule example: Once the order has been received, it will be admitted and checked whether its issuer has been already introduced in the SAP IF its value is less or equal than 1502 Euros. **Business rule and UML activity diagram:** verifis that there is no a path without the gard [value <= 1502] between the “OrderReception” activity and the “DoesProviderExistInSAP?”
--OCL Expression package Behavioral_Elements::Activity_Graphs context ActionState

```
inv ProviderOKCheckBeforePayment:

name="OrderReception" implies Set{self.oclAsType(StateVertex)}-
>closure(e | e.outgoing->select(guard->isEmpty or
guard.expression.body<>"value <= 1502").target->asSet)-
>select(name="DoesProviderExistInSAP")->isEmpty()

endpackage
```

Rule pattern: A certain activity must be present between two other activities if there is a path connecting them

Business rule example: A first funds reservation will be made by the office of the treasure according to the amount indicated by the economic evaluation.

Business rule and UML activity diagram: Verifies that between the start (or an activity) and other activity (PaymentProcessing) all the paths that connect them include an activity ("Reserve"). It does not verify IF THERE IS A PATH

```
package Behavioral_Elements::State_Machines

context StateVertex

inv ProvisionOfFunds:

name="start" implies Set{self}->closure(
e | e.outgoing.target->reject(name="Reserve"))-
>select(name="PaymentProcessing")->isEmpty

endpackage
```

Rule pattern: Existence of a certain object data flow between the start and the end of the activity diagram

Business rule example: An authorization will be signed by the responsible before giving course to the payment.

Business rule and UML activity diagram: Verifies that between the start (or an activity) and other activity (PaymentProcessing) a certain object data flow will be present

```
package Behavioral_Elements::State_Machines
context StateVertex

inv ProvisionOfFunds:

name="Start" implies Set{self}->closure(
e | e.outgoing.target->reject(name="SignedAuthorizaton"))-
>select(name="PaymentProcessing")->isEmpty
endpackage
```

Rule pattern: Presence of a certain activity within a certain swimlane

Business rule example: The Management Unit is the responsible of receiving and processing the orders.

Business rule and UML activity diagram: Verifies that a certain activity appertains to one swimlane

```
package Behavioral_Elements::State_Machines
context StateVertex

inv ActivityPresentInSwimlane:

name="OrderRecption" implies partition-
>select(name="ManagementCenter")->notEmpty

endpackage
```

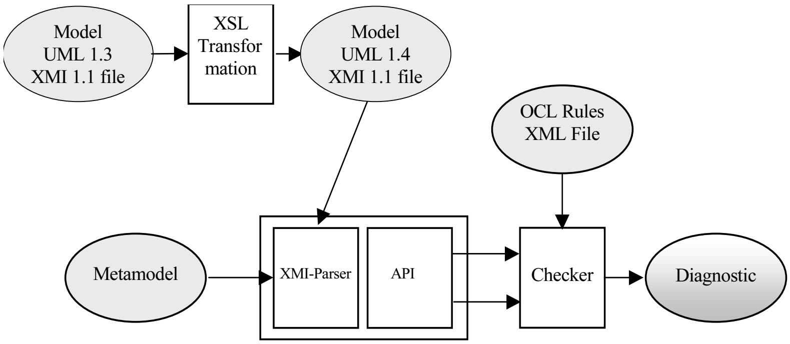
CHAPTER 4: NEPTUNE CHECKER

From many points of view, the NEPTUNE OCL evaluator represents a novelty in CASE tools supporting OCL and (or) checking UML models. The main objective of this part of the book is to support the user in understanding our checker philosophy, helping to an efficient use of this evaluator.

Tool Description

In this paragraph we present the architecture of the NEPTUNE checker. First of all, note that all external documents in input of NEPTUNE checker are expressed in XML. It is for example the case for the UML metamodel, but also for the user's model and at last for the OCL rules to verify. Both model and metamodel are stored in XMI files which DTDs are compatible with version 1.4 of UML.

As described in the figure below, the OCL checker takes for input a database of OCL rules, together with the UML metamodel and a user model. It returns diagnostic information indicating which rules are syntactically or semantically incorrect, and the model element on which rules fail. As most current tools only export UML 1.3, some transformations must be applied to their output XMI so that it becomes compatible with UML 1.4. These transformations are performed using an XSLT processor. XMI files are then parsed and transformed into an internal form that is accessed by the checker through an API compatible with UML 1.4.



The checker loads the metamodel and uses it to validate and compile the OCL rules. The model is then loaded and traversed so that each model element is made an instance of its associated metaclass.

Checking rules

This section presents a classification of the OCL constraints verified by the NEPTUNE checker. As previously mentioned, they can be split into two subsets: rules attached to metalevel classes and constraining the structure of every UML model and, rules included in a user model and constraining the implementation of the model.

Metamodel level rules

The starting point for the specification of this set of rules is the UML semantics document. It uses the OCL language to formalise constraints over metamodel instances. We have enriched this first set of well-formedness rules by including inter-diagrams coherence rules, software engineering rules, target language rules, and business process[3] rules.

- ***Well-formedness rules***: Most of the rules, appearing in the UML semantics reference document [ref7], are intra-diagram rules. They add structural constraints over model diagrams that cannot be expressed graphically at the metamodel level.
- ***Inter-diagram coherence rules***: Inter-diagram rules require checking properties between at least two different kinds of diagrams. They are in general not specified in the UML semantics document and may depend on the UML methodology. Inter-diagram rules include type-checking rules that require verification of method calls encapsulated in messages relative to the class contents and the inheritance hierarchy. At the object level, the number of links between objects in a collaboration diagram must conform to the multiplicity declaration of the corresponding associations. Sequence diagrams describing possible execution traces can also be checked against statechart diagrams describing all possible execution paths.
- ***Target language oriented rules***: Target language rules are rules that require checking properties dependant on the target language chosen for the implementation. For example, one can check the

[3] A description of a set of related activities that, when correctly performed, will satisfy an explicit business goal.

Java requirement disallowing multiple inheritance of classes. Such rules are grouped and can be activated from the NEPTUNE interface.

- ***Software engineering rules***: They address the verification of properties that depend on the methodological process imposed to develop an application and on metric constraints defining modelling rules. The first point concerns the definition of rules applicable at the end of each phase of a development. They are defined in accordance with the NEPTUNE process and concern, for example, the way that external actors must be specified and their interaction with the system. The second point is more project dependent and can express naming conventions, size limitations for classes or packages, *etc.*
- ***Business process rules***: Business process design relies on methodological rules, which describe the mapping of business concepts into the UML notation. The Ericksson-Penker [ref2] business extensions have a stereotyped note for defining rules. The note is stereotyped <<business rule>> and is attached with a dashed line to the model element (class, operation, etc.) to which it applies. Three categories of business rules are extracted:
- Derivation rules define how information in one form may be transformed into another form, or how to derive some information from another piece of information.
- Constraints govern the structure and the behaviour of objects or processes, *i.e.*, the way objects are related to each other or the way object or process changes may occur.
- Existence governs when a specific object may exist – usually inherent in the class model.

Model level rules

Model level rules are defined by the application designer and are associated with user-level model elements in order to constrain the application code. The model element to which a rule applies is called its context and can be a class, a method, or a dynamic diagram:

- ***Class invariant and operations pre- and post- conditions***: This usage of OCL constraints is already allowed by UML. The NEPTUNE tool only checks their syntax and semantics. A more extensive static checking would require proof techniques;

- ***Action clauses proposed by UML/OCL 2.0***: They offer the ability to specify which messages can be sent by an operation, thus allowing some kind of verifications over dynamic diagrams. However, such verification is limited to a one step transition. The current version of NEPTUNE checker only verifies the semantics' correctness of these rules;
- ***Temporal constraints over dynamic diagrams***: The NEPTUNE tool introduces the ability to specify temporal properties over dynamic diagrams through an extension of the OCL language. Temporal operators are borrowed from the Computational Tree Logic (CTL) [ref4] and applied to sequence and state chart diagrams. In the first case, properties over the occurrences of specific messages are asserted. In the second case, properties concern the state of the current object.

Tool Usage

This chapter intends to show how all the checking capabilities can be used within the Neptune tool so that UML models can be checked to verify their consistency and coherence.

To take advantage of all Neptune checker's features, the tool is composed of different features to make it easy to handle:

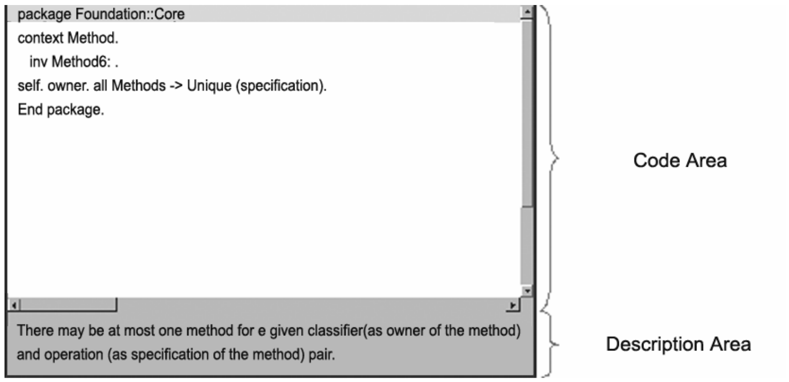
- a **Rule Designer** supporting both edition and creation of new Rules,
- a **Set of Rules** embedding the most useful ones for software design,
- a **Rule Checker** to verify whether the UML model is compliant with the rules or not
- an **Explorer** to browse through the outputs from the Checker.

Usually the user will not be required to create his own rules as Neptune offers a host of them grouped into several categories to make them easy to find. These rules are likely to suit any requirement, even though new ones can also be designed to enlarge this initial host.

The Rules

Overview

Each rule is defined in order to check that the model is compliant with a specific design criteria. The rules can be created or modified by using the Rules Designer.



```
package Foundation::Core
context Method.
  inv Method6: .
self. owner. all Methods -> Unique (specification).
End package.
```

There may be at most one method for e given classifier(as owner of the method) and operation (as specification of the method) pair.

Rules Designer

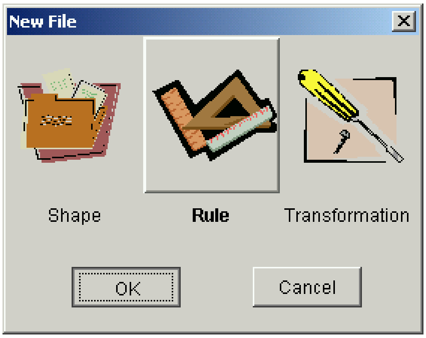
The Rule Designer is divided into two different areas:

 The **Code Area** where the source code of the rule is displayed

 The **Description Area** where the system displays a short description of the checking actions performed by the rule.

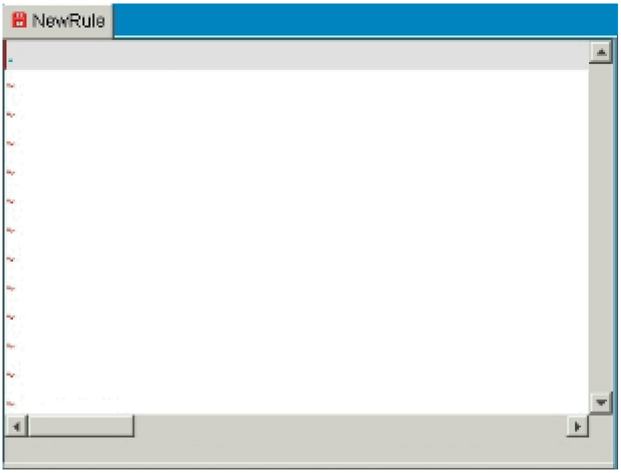
Creation of new Rules

To build new Rules, go to "`File | New`" and select "`Rules`" option from one of the three options available:



Creation of Rules

This action will open a new blank Rule Designer in the Neptune work area:



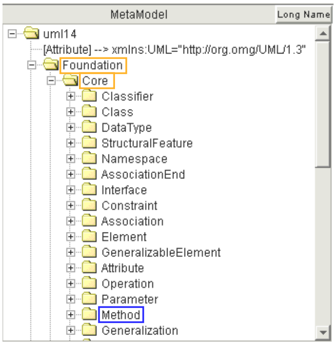
A new Rule Designer

The OCL source code of the rule has to be written within the white area, while the description can be attached when saving the rule or at anytime by using "`File | Properties`".

While writing a rule, the MetaModel Explorer can assist the user in both selection of package and context clauses . For instance:

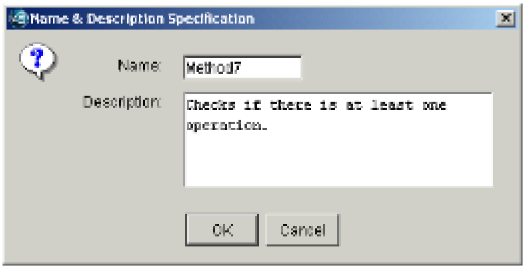
```
Package Foundation::Core
context Method
  inv Method6:
  self.owner.allMethods->isUnique(specification)

endpackage
```

The MetaModel explorer can help us to find the package path and all its possible contexts. The next figure is a snapshot of the MetaModel Explorer with both path and context highlighted.



Using the MetaModel Explorer as an assistant

When the rule has been completely designed, it can be saved into the Project by choosing "`File | Save "RuleName"`". As the rule is unnamed yet, Neptune will ask for a name and a description:



Name: Method7

Description: Checks if there is at least one operation.

Specifying Rule Name and Description

Modification of Rules

To create new rules, you can simply start from an empty one. Nevertheless, this task can be also carried out more easily by editing an already existing one. Later on, this operation to be performed usually only when the rule has not been well defined or has any error (look chapter *Performing the check* for further information).

As the rules being part of the global storage are supposed to be fully functional and tested, they (a priori) can not be modified. However, it is possible to do it, but it is first necessary to import the rule to modify into a project using "`Project | Add Files…`" menu

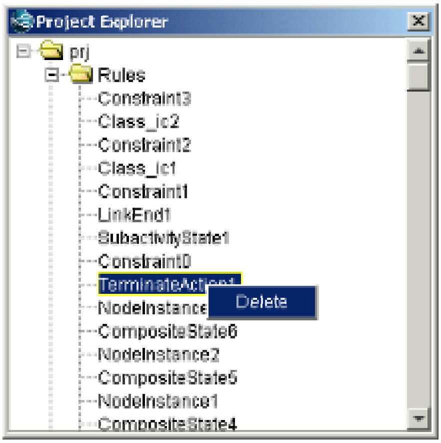
Then, to edit a rule, simply open the Project Explorer, by choosing "`Project | Explorer`" and double-click on the one to modify. This operation will open a new pane with a Rule Designer showing the selected Rule.

To save the Rule simply type Ctrl-S or choose "`File | Save`".

Deletion of Rules

A rule can be deleted with the Project Explorer window, as this is the place where all Project's elements, including Rules, of course, are displayed. To do so, open the Project Explorer using "`Project | Add Files…`".

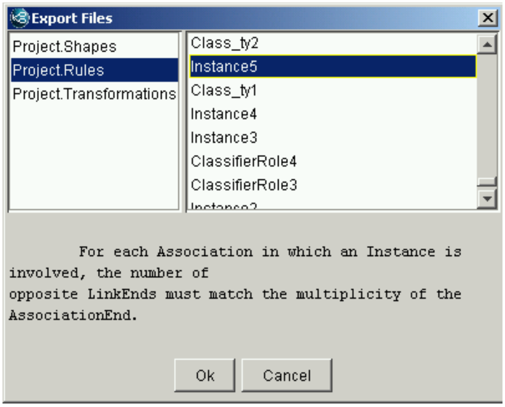
Inside the Project Explorer, right-click on the Rule to delete and choose "`Delete`":



Deleting a Rule from the Project

Exporting a Rule to the Global Storage

The rules created by the user are always project dependent, which means that they only exist in the context of the Project in which they have been created. It is slightly different from the global Rules that are available independently of the current Project.
So that new rules may be used in any other project, they can be exported to the Global Storage. They will so far be shared by all the NEPTUNE projects.



```
        For each Association in which an Instance is
involved, the number of
opposite LinkEnds must match the multiplicity of the
AssociationEnd.
```

To do so, go to "`Project | Export`" from the menu and select the folder "Project.Rules" on the left. All the current Project's rules will then automatically be shown on the right:

The next step consists in choosing a rule destination folder into the global storage.

Invoking the Checker

Once we have the set of rules that express our quality requirements or our design constraints, time has come to run the checker and thus identify any kind of deviation or mistake in the model. The required elements are the XMI file containing the UML model to be checked, the UML corresponding MetaModel and the rules to be checked.

Both XMI file and rules are user dependent. They are chosen according to the user needs, while the UML MetaModel is intrinsically included within the Neptune Tool.

The following section we detail how to apply these rules and then take advantage of the results to improve the model quality.

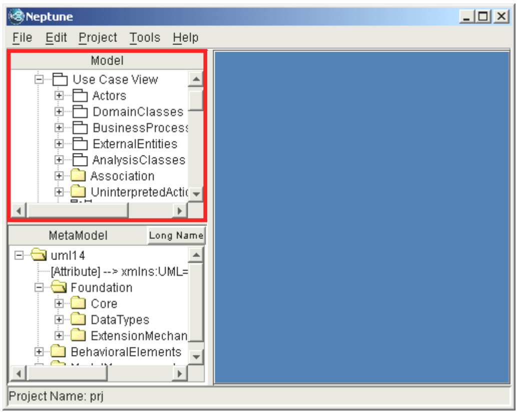
Selection of the Elements

Though the NEPTUNE checker can perform the checking actions on the entire UML model, the user can also choose to run the tool on a specific part of its model, let's say a sub part of the model.

For instance, rules regarding actors and use cases may only be applied to the Use Case View, while Business Rules may be used to check the specific parts of the model dedicated to business process.

These selections can be carried out from the two Neptune's explorers, ie: the Model Explorer and the MetaModel Explorer.

From the model hierarchy



The Neptune's Model Explorer

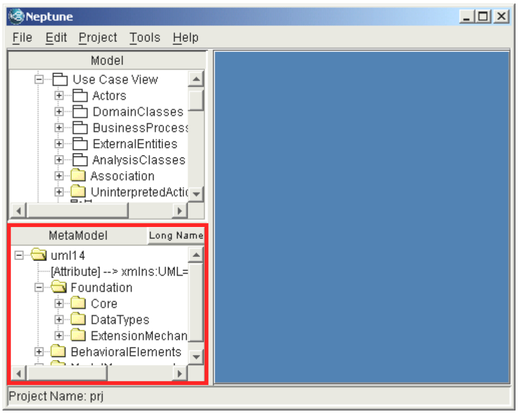
To select one element, click on the corresponding node. Of course, more nodes can be selected by holding the `Ctrl` key and clicking on them. To deselect any previously selected node, the `Ctrl` click does the trick.

To perform a check on the whole UML model, simply click on the root node of the tree.

It is important to notice that, when performing a multiple selection, it is not necessary to explicitly add elements that are represented inside an already chosen parent node as the checker adds recursively all the elements selected.

From the corresponding metaclass

We have seen in the previous chapter how to add the elements searching them in the model explorer, which is the exact view of the actual UML model. Let's now focus on an other view, the MetaModel Explorer one:



The Neptune's MetaModel Explorer

This view is very useful to find model elements using their corresponding metaclass. For instance, looking for a specific actor may be easier by means of the node "BehavioralElements | UseCases | Actors" than having to remember its exact location in the model.

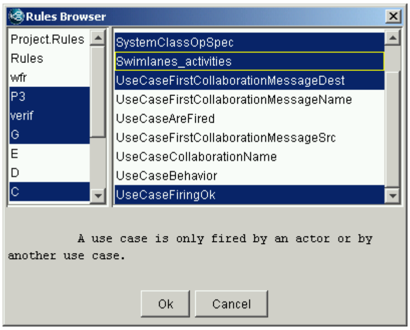
In addition, right click on this element, and choose "`Go To Model`" will highlight it in the Model Explorer, thus showing the exact location where it's been defined. In addition, the button labelled "Long Name" will show the whole path to avoid ambiguity between elements having the smae name in different .namespaces.

Performing the check

Once the elements have been chosen, the menu "`Tools | Rule Checker`…" will launch the checker GUI.

Then first appears a dialog with all the Rules available in the tool, the Project's ones and the global's ones.

The window is divided into two areas: on the left zone are shown the different categories while the right zone details the rules contained in each category. Of course multiple selection is available on both sides of the window : Simply hold the Ctrl key.



```
            A use case is only fired by an actor or by
another use case.
```

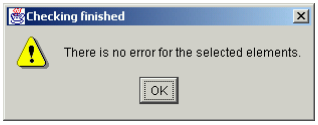
Choosing the Rules to check

Once the rules have been selected, clicking on the "OK" button triggers the Checker. The following splashscreen is a witness of the checker activity. It is displayed until the selected rules have all been evaluated on the UML (sub) model.

The checker is running

Depending on the results of the checking process, NEPTUNE might return various outputs:

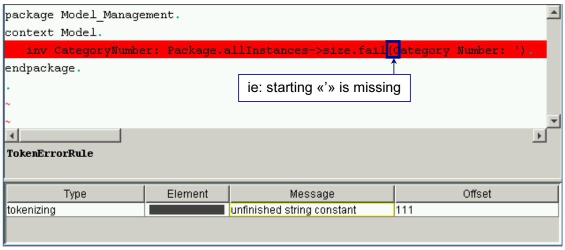
- ➢ No errors: a dialog will show a message informing that no error has been detected for the Rules and elements selected:



There is no error for the selected elements.

No errors dialog

- A list of errors of the following types:
    - Tokenising error: the OCL Rule could not be tokenised:



```
package Model_Management.
context Model.
   inv CategoryNumber: Package.allInstances->size.fail(Category Number: ').
endpackage.
.
~
~
```

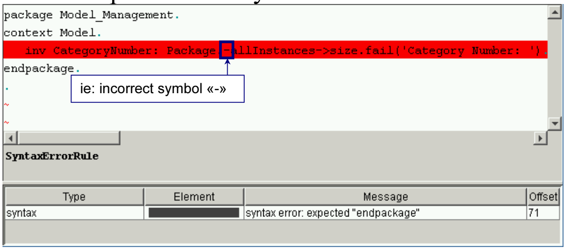
TokenErrorRule

Type	Element	Message	Offset
tokenizing	▬	unfinished string constant	111

An example of tokenising error

Neptune highlights in red the line where the untokenizable word is found.

- Syntax error: the OCL Rule is not compliant with the OCL specification syntax.



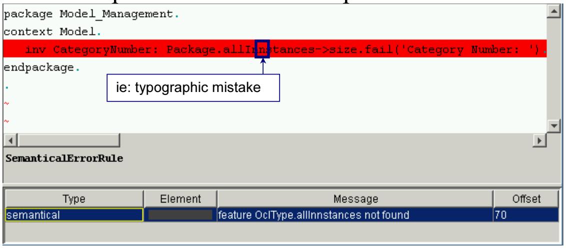
```
package Model_Management.
context Model.
   inv CategoryNumber: Package.-allInstances->size.fail('Category Number: ').
endpackage.
.
~
~
```

SyntaxErrorRule

Type	Element	Message	Offset
syntax	▬	syntax error: expected "endpackage"	71

An example of syntax error

- Semantic error: a feature requested on the OCL rule is not compliant with the OCL specification:



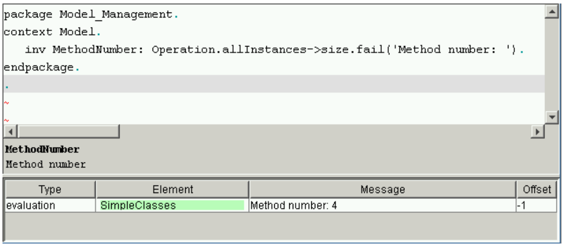
```
package Model_Management.
context Model.
    inv CategoryNumber: Package.allInstances->size.fail('Category Number: ').
endpackage.
```

SemanticalErrorRule

Type	Element	Message	Offset
semantical		feature OclType.allInnstances not found	70

An example of semantic error

- Evaluation error: this is a rule defined message informing about some situation regarding the model loaded.



```
package Model_Management.
context Model.
    inv MethodNumber: Operation.allInstances->size.fail('Method number: ').
endpackage.
```

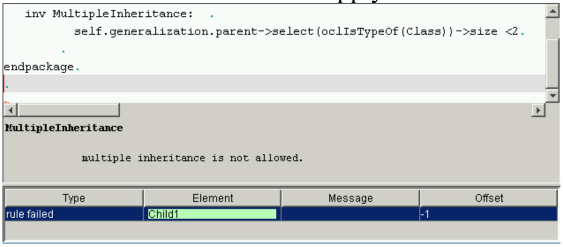
MethodNumber
Method number

Type	Element	Message	Offset
evaluation	SimpleClasses	Method number: 4	-1

An example of evaluation error

In the example above, the rule counts the number of methods of the SimpleClasses model.

- Rule failed: the rule could not apply to the model



```
    inv MultipleInheritance:  .
         self.generalization.parent->select(oclIsTypeOf(Class))->size <2.
           .
endpackage.
```

MultipleInheritance

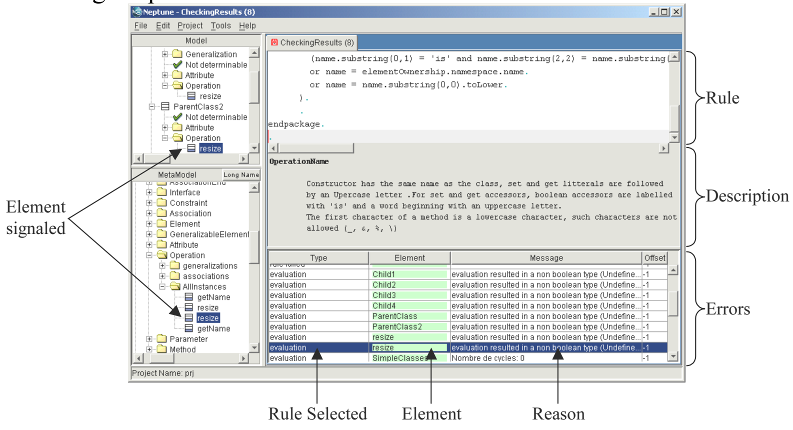
multiple inheritance is not allowed.

Type	Element	Message	Offset
rule failed	Child1		-1

An example of Rule failed

In case of multiple errors, whether it comes from one rule on several elements or from different rules, NEPTUNE displays as many lines as errors found.

Moreover, each click on one of these lines highlights the concerned UML element and its metaclass. This is done thanks to the NEPTUNE explorers. To make the error even more explicit, both the rule and its description are displayed at the top of the window. This is shown on the following snapshot.



```
        (name.substring(0,1) = 'is' and name.substring(2,2) = name.substring(
        or name = elementOwnership.namespace.name.
        or name = name.substring(0,0).toLower.
      ).
      .
endpackage.
.
```

OperationName

```
        Constructor has the same name as the class, set and get litterals are followed
        by an Upercase letter .For set and get accessors, boolean accessors are labelled
        with 'is' and a word beginning with an uppercase letter.
        The first character of a method is a lowercase character, such characters are not
        allowed (_, &, %, \)
```

Type	Element	Message	Offset
evaluation	Child1	evaluation resulted in a non boolean type (Undefine...	-1
evaluation	Child2	evaluation resulted in a non boolean type (Undefine...	-1
evaluation	Child3	evaluation resulted in a non boolean type (Undefine...	-1
evaluation	Child4	evaluation resulted in a non boolean type (Undefine...	-1
evaluation	ParentClass	evaluation resulted in a non boolean type (Undefine...	-1
evaluation	ParentClass2	evaluation resulted in a non boolean type (Undefine...	-1
evaluation	resize	evaluation resulted in a non boolean type (Undefine...	-1
evaluation	resize	evaluation resulted in a non boolean type (Undefine...	-1
evaluation	SimpleClasses	Nombre de cycles: 0	-1

The whole Neptune with checking results

Exporting the Checking Results

Once the check has been performed, it is possible to export the results in an XML file. The checking results are stored and can thus be used anytime. It is easy to understand that the reading of these checking results might not be very easy because of the XML tags. To make it easier, a simple trick consists in transforming the XML output into an HTML or PDF file. This is obviously done with the NEPTUNE document generator. The backup of the checking results is done through the "`File | Save CheckingResults`" menu.

PART III

DOCUMENTING Models

In this third part, we focus on the documentation aspects to be considered during and after the UML modelling process. After a presentation of what is documentation in general and why it is so useful in various business fields, we will introduce the two main concepts of the NEPTUNE automatic documentation generator, namely transformation and shape. Then we will focus on the tool with a detailed description of the documentation generation process followed by a chapter related to the tool usage.

CHAPTER 1: DOCUMENTATION PURPOSE

An important element of a process is its documentation. The documentation associated with a process contains its complete description, and highlights the most important events that occur during its progress.

This is even truer for software as experience has shown that the success of software projects is largely dependent of the quality of their documentation. Because the software projects have a very expanded life cycle going from their specification and their design to their realisation, their tests and their maintenance, a bad documentation or even the absence of documentation cannot assure the continuity between the steps of the software's life.

In addition, software is realised and maintained by engineering teams, often dispersed among various companies, various geographical locations: a bad documentation does not assure a good co-ordination among the different engineers implicated in the software project.

In other words, the documentation is a required quality of modern software. Thus, the constitution and the automatic generation of documentation for software process become more and more necessary.

This chapter can be seen as a global presentation of what is documentation generation. After a sub chapter about the added value of documentation generation, the reader will be presented a state of the art. The very end of the chapter consists in a first introduction of the NEPTUNE document generator.

The worth of documentation generation

This section points up the added value of the automatic documentation generation.

The actual progress of technology (performances, networks…) improves the software performances and can provide new functions and features. It follows that the application domains become more and more large and varied. In business areas like space or automotive, the software is becoming an essential (sometimes the main) part of the systems developed. Consequently, collaboration at all levels in lives of software becomes necessary between software engineers and application domain specialists.

For example, the graphical user interfaces are elaborated by software engineers in tight collaboration with the final users. Of course, the documentation is part of this collaboration, and thus must be accessible by all people implicated in the software projects and not only software engineers. There follows a true (r)evolution of methods, techniques and tools of the documentation because this one must be accessible by all people implicated in the software projects and not necessarily software engineers.

In such conditions, it is important to consider the generation of documentation that can be a great help, saving time and automatically bringing into the documents a normalised formalism. This standardisation makes the communication easier between the people implicated in the project.

Let’s now focus on the use of UML and try to understand how automatic generation naturally fits into model based developments.

UML offers the user different views of process activities, both static and dynamic, based on different technical and functional users’ profiles. Being a standard, UML is now widely used to model software. Its graphical approach facilitates easy use by people. The UML formalism is adapted to users who are not specialised in software development: without having need for a deep understanding of its possibilities, they can follow the modelling of their projects. For this reason the project domain of UML applications is wide open. Consequently, UML is used for modelling diverse information systems, bank and financial services, telecommunications, transportation, military, and distributed Web services. We can easily draw a parallel between software engineering and other domains where modelling also plays an active role in terms of communication: for example, architectural models of houses and buildings nowadays consist for the future occupants in a first opportunity to view the final result.

Pushing the concept even further, we can say that UML use is not limited to software modelling. This language is sufficiently expressive for modelling systems that do not belong to this category. For example, the workflow of a judicial system, the structure and behaviour of a health system, the design of computer system or, more generally, a business process, topic which has been under study in the Neptune project.

In order to set up an exchange process between the organisation and the information system, it is consequently fundamental to have a documentation system accompanying the models. Moreover, it should be based on automatic generation of end user oriented documents.

If documenting a software application is quite common and widely approved to be a great help through the co-ordination of software teams, the documentation of a process model that is not in the framework of software engineering is not yet a usual practice. Because of the specificity of each business, each of them may require a different approach for end user oriented documentation. In this framework, NEPTUNE offers a great support, but this subject will be discussed in details within the next chapters.

Documentation generation: State of the art

In terms of generation around UML, the source code has been more explored that any other type of natural language documentation. This can easily be explained and understood. First of all, even though the radical changes and improvements done in software engineering, the production of source code can not be avoided. On the contrary, documentation rarely appears as the first priority. This is one of the reasons why more efforts have already been done on code generation that on documentation generation. Moreover, the source code can be seen as text only, with a formal syntax, while documentation is composed of text with a particular rendering, a particular style; it can also include some draws, or other extra information that does not automatically appears in a standard UML model.

In spite of this delay, some interesting things have already been done on documentation generation: The evolution of the modelling languages makes them closer and closer from the natural languages. UML seems to be a perfect example, with the opportunity to use graphical stereotypes or with the diagrams that remain quite intuitive even for someone that does not know UML perfectly. Consequently, the diagrams have been widely used by the documentation generation tools. Unfortunately, most of the time, they are simply extracted from the models and copied into a text document. We could give some more examples but most of them would converge to the same conclusion: The tools available on the market do not really provide the flexibility required for a good textual documentation generation. In addition the model parsing is often based upon an internal form which make these tools sharply linked to UML case tool associated. Well, it is clear that the documentation generation topic seems to be an open job where most of the things are still to be done.

These thoughts and studies led us to think that basing our work on a standard was quite interesting. We naturally chose XMI, kind of repository of the information contained in the model. In addition to be able to manipulate a big amount of information from the model, we thought that as for checking, the methodology had to drive the documentation generation. Indeed, it is essential to specify:

- What are the modelling elements to document?

- How can the analysts and the designers document the modelling elements?
- What are the checking rules dedicated to the documentation in order to check the presence of documentation on modelling elements and where these modelling elements are documented (notes, containers, ...). In addition, these checking rules will provide for the analysts or designers an easy way to detect information that is missing or not useful.

Introduction to our document generator

The aim of this chapter is to highlight the essential points that make the NEPTUNE documentation generator different from what had been done previously in the domain of documentation generation around UML.

General concept

The NEPTUNE documentation generator aims at producing some professional documentation that results from the exploitation of UML models. This documentation is an end-user-oriented documentation, which takes into account the professional expertise of the reader. It is the result of transformations applied on both UML elements and documentation available in the actual model or purely external.

Multiple points of view

Among the features of the NEPTUNE documentation generator, its ability to give many views of the same information is essential. If the extraction of information from a UML model leads to one view, transforming this extracted information can produce many other views.

The NEPTUNE document generator is in a way based on this idea of transforming the information. While specifying our tool, the transformation feature has quickly appeared to us as being one of the most essential.

The transformation’s layer is fully customisable in the NEPTUNE document generator, making the number of views as wide as the documentation designer imagination. The access to multiple points of view highly contributes to make the tool different from the others.

Multiple compatibility

Today, each UML case tool has its own proprietary back up format, but due to the increasing number of tools emerging around UML, none of the case tool editors can avoid to supply a model export feature. To do so, the format used is always XMI, which is indeed an XML format using a DTD adapted to UML. This format (described in appendix C) is part of the UML norm. It thus can be can considered as a standard and is said to be the UML exchange format. The NEPTUNE documentation generator is based on this format; the models used as input of our tools must be expressed in XMI. This makes NEPTUNE compatible with any case tool compliant with the norm. As to conclude, we can say that there is no kind of coupling between NEPTUNE and any other UML case tool. NEPTUNE tool is a standalone tool.

Multiple sources of information

Though the UML model is the main source of information used to generate the final documentation, the tool can also make use of other sources. It is possible to extract and transform the information initially located in an external document, whether this document is referred in the model or not. There is however one restriction concerning the external documentation. It must be some XML documentation. NEPTUNE is able to handle any tagged document and make use of any part of such a document, small pieces but also the whole document if needed.

CHAPTER 2: BUSINESS DEDICATED DOCUMENTATION

Depending on the business domain, the look, the use, the life cycle, etc…. of the documents are slightly different. These differences lead to specific definitions of what is the standard documentation for each domain. The following chapters will present the documentation in two different business fields: business process and software engineering

Documentation in business process

This sub-clause aims to focus on the usage of NEPTUNE document generator tool in the domain of business process modelling related documents.

The advantages that models represent for business are well known. Explanatory documents of these models (which in fact are explanatory documents of the business themselves) are a crucial element for building up an agreed and shared knowledge of the business among relevant parties (managers, stakeholders, other staff members, etc).

NEPTUNE process has identified a number of work definitions each one focussed on modelling specific aspects of the business. The models produced there can be accompanied by explanatory documents. What follows is a list of elements that business modellers could take as starting point for defining such a documents. This list is organised by business views, that nicely correspond to the different NEPTUNE work definitions.

For the Vision Business a document should be drafted containing, among others, text with the company mission, some general objectives, the business domain boundary, an opportunity/problem statement describing the opportunity for the business, external major threats, critical factors, core competencies of the business, high level description of the organisation, and a TOWS matrix.

The work definition dealing with business concepts identification and definition should complement the models produced with a lexicon for the business containing, among other things: traceability identifiers, definition of the concepts, short explanation of the relationships that connect concepts in the conceptual model, remarks on those concepts that require additional clarification. The second activity of this work definition deals with the identification of the processes, stakeholders and resources. An accompanying worth document would give information on each process containing its brief description, the estimation, justification and explanation of its required resources as well as the relevant considerations affecting the involved stakeholders.

Goals and problems identification work definition could be complemented with a document with information on them. For goals, suggested items could be: traceability identifier for each goal, its short description, any additional remark (as how critical it is for the wealth of the business), whether it is a qualitative or a quantitative goal (and if it is the later one, information helping to assess its achievement, like goal value, current value, measurement unit, etc), detailed rules for assessing its achievement degree, criteria for classifying the achievement degree, the list of sub-goals, contradictory goals and estimation of desired balanced achievement of them. For problems: traceability identifier, short description, additional remarks, list of causes of the problems (accompanied by explanations of each one), list of actions foreseen to overcome the problem (with additional descriptions of such actions), prerequisites, resources requested by them and processes in charge of solving the problem.

Documents complementing the models for resources could include information on skills required for playing the different roles, availability needs, hours needed and envisaged period of engagement for them, when dealing with people. For physical resources issues like required units, frequent suppliers, costs, conditions, average delivery time, reposition frequency required by processes and foreseen period For the business structure, each organisation unit could be additionally documented giving details on the employees, resources, methods for allocating them, methods for assigning tasks, rules for arrangements and links with other OUs, its expertise, etc.

Documents explaining the business processes can include, among others: traceability identifier, short explanation of the purpose of the process, its scope, how critic it is, the goal that tries to achieve, the list of resources required, the actors involved, its owner, its outputs, its risk, etc. Next clause gives an example of document including most of them.

NEPTUNE document generator tool makes extensive usage of XML files for producing the final documents. Inside the XML files the different pieces of information appear as content of XML elements. The XML tags qualify these pieces of information, which allows their automated processing by XSLT engines as a step in the production of the final document. Future activities on the business modelling and documenting area would focus on producing proposals for files containing information on all the issues mentioned above and other that could be identified as relevant, like NEPTUNE has done for generating the document containing details of a number of business processes and their corresponding business rules.

Examples of business modelling documentation

One of the most relevant documents in medium to big organisations is that one containing detailed description of the processes that take place in that organisation.

In this section, we present the contents of such a kind of document, produced with NEPTUNE tools. It is based on UML, and part of the required information is available in XML files specifically defined for these purposes.

The document itself consisted in a number of chapters, each of one contained the detailed information of one process dealing with payments. All the chapters had the same structure. Sub-clauses below provide information on their contents.

Process identification and scope

This section contains a short sentence explaining where and when the process applies.

Involved entities and their responsibilities

It contains a **table** with two columns: the first one contains the names of the different parties participating in the process and the second one gives a short hint of their roles in the process, as shown below:

Entity	Responsabilities
Management center	Manage good adquisistion, Register accounting phases
Payment authorizer	Order payment

Diagram of the process

This section contains the activity diagram of the process that is part of the whole business model.

Detailed description of the process

This section contains a table of four columns containing the details on the process.

- The first column contains a detailed description of the more relevant phases of the process. Usually, all the activities required to successfully perform the process are explained in this column: it concentrates the core knowledge of the process.
- The second column contains references to forms that during the process the different entities have to manage somehow.
- As payments were supported by an information system, the third column contains references to its user.
- Finally, the fourth column contains references to normative documents (both, internal to the organisation and external – national laws).

Process Description	Forms	User's manual	Normative references

Business rules governing the process

The final section is a list of business rules that govern the process, expressed in natural language.

Documents on-line containing information on the processes, as HTML presentations with navigation capabilities that can be posted in a web service and made available to all members of the staff of an organisation for educational and consulting purposes, are also a good example of how NEPTUNE technology can give support to the business model documentation.

Documentation in software engineering

Any software-engineering project makes use of several specific languages, each one being associated to a particular technology used on the project. Of course, the language of the business area concerned with the information system is also in use on the project. This variety of vocabulary is declined at each step of the process. For instance, the use of UML brings into the project a specific terminology, from the first steps of the development process until the end.

Consequently, there are very few project teams in which each member is totally skilled and familiar with the whole vocabulary in use: if we can expect from a software engineer to read and understand an UML model, it is not the case for the other members of the team. In this particular case, and in a context that always gives more importance to the models (MDA), it is thus fundamental to give an easy and friendly access to the whole information contained by the models. This information should then be understood by any member of the project team, whatever its knowledge of UML. To do this, the textual documentation remains the most efficient tool.

Moreover, the modern software development (process, quality, methods) has brought in big needs of documentation (from specification to validation).

This sub-chapter intends to point up the main role played by documentation in software engineering.

Different types, different means

Documentation as part of the process

Let's start with a piece of history. Once upon a time, there was software development, which principles were mainly based upon algorithmic. Software development has always (and still) been in evolution and today, it has reached kind of adulthood. But in the industrial software domain, the knowledge of algorithmic is not sufficient anymore to assume a good quality of the pieces of software developed. Because they need to be maintained, reused, because of scalability requirements, these pieces of software need to be thought and designed in reference to techniques, guidelines, methods and principles, that all together constitute what is known as software engineering. In such a context, the documentation has obviously found a role to play in the development process. Some documents are now said to be standard documents: from one project to the other, they document the same part of the process, they have the same names, and finally the same objectives. A classical example is the interface manual. These documents are often mandatory on the projects and though their reason of being is the same from one project to the other, what they look like (rendering and content) is often constrained by the software development process in use on the project, in the company.

Documentation as alternative views of the model

Among the objectives of the documentation in software engineering, giving an easy access to the information contained in the models is essential, whether the models are UML models or not. To do so, several approaches can be chosen:

A first option consists in inserting in the model some pieces of documentation. Of course, that means that the modelling tool used on the project supplies such features. Moreover, this way of documenting a model is very often left at the initiative of the project members that design the model. This writing technique is well adapted to the documentation of a precise part of a model, like a class or a package. In this case, we will talk about an object oriented documentation.

Another option consists in writing some documentation beside the model. This documentation is not as close to the model as the one presented in the previous paragraph. That means that it can be dealing with general considerations on the model, or be much more business oriented. Because of this, and often depending on the level of organisation of the company, there can be some templates for this type of documents.

Examples of software engineering documentation

Among the classical software engineering documents, the validation plan appears as a standard. It is on many operational projects a mandatory deliverable. The main objective of such a document is to check that the high level specifications of a project are fulfilled by the software. To do this, the software is used as a black box.

In this section, we present the content of a validation plan, produced on a project on which the NEPTUNE process is the one in use. It is based on UML, which assumes that the whole information needed for the writing of a validation plan should be available in the model itself. The use cases are supposed to be the mirror of the specification. Consequently, we will also give some detailed information on the places in the model where the needed information can be found.

Pre-requisites

First of all, it is important to mention that the UML/NEPTUNE process recommends that each use case should be documented in a textual form, respecting the following categories.

Summary	Brief presentation of a use case
Use context	Conditions of use by triggering elements (frequency of activation, synchronous or asynchronous triggering, etc.)
Triggering element	Actor or use case
Pre-conditions	Stable system condition necessary before use case can be accomplished
Input data	Data used
Description	Detailed description of interaction between triggering elements and the system
Post-conditions	Stable condition achieved by the system at the end of the interaction with a use case
Output data	Data produced
Exceptions	Error condition that the system cannot resolve

The information included in this document will be very useful all along the generation of the validation plan.

Definitions

- **Execution case** : execution for a particular scenario (defined by a sequence diagram)
- **Test case** : sequence of execution cases
- **Validation case**: use case name in validation phase. A validation case makes reference to a test case
- **validation plan** : set of validation cases

The validation plan

A validation plan is composed of several tables. The first one is the table of the validation cases, in which some references are made to test cases. As a test case is a sequence of execution cases, each test case description will include:

- one table for the sequence of execution cases
- as many tables as execution cases in the sequence

The table of the Validation Cases

One raw in the table is needed for each validation case. The table of the validation cases is presented below:

	COLUMS OF VALIDATION CASES				
TEST TYPE					
TEST ID					
USE CASE REFERENCE	hyperlink				
USE CASE VALIDATION TEST DESCRIPTION					
NEXT TEST IF ANOMALY					
TEST TRACEABILITY					

- **TEST TYPE**: functional, robustness, performances, ergonomic, documentation. As this information is at the moment not available in the model, we have to improve the Neptune process to add this information.
- **TEST ID:** Identifier of the test.

- **USE CASE REFERENCE**: Name of the use case described (there is one validation case per use case). This hyperlink gives access to the test case of the current validation case.
- **USE CASE VALIDATION TEST DESCRIPTION:** Description of the use case. It is extracted from the associated documentation mentioned in the former paragraph.
- **NEXT TEST IF ANOMALY**: The name of the test to perform in case of current test failure. As this information is at the moment not available in the model, we have to improve the Neptune process to add this information.
- **TRACEABILITY**: Traceability information from the model, available in the model through the Neptune traceability tagged value.

Test Case

In the first table below is given the sequence of execution cases required for the test case and some additional information describing the test case. All this information is extracted from the use case documentation described in a former paragraph.

This sequence is said to be the primary scenario. Within the model, this information is contained in the main sequence diagram associated to the use case.

	Test case: Use Case name.
OBEJCTIVES	Summary field of the external documentation
PRE-REQUIREMENTS	Pre-conditions field of the Use Case External Documentation
PROGRESS INSTRUCTIONS	Sequence of secondary scenari. (secondary sequence diagrams)
INPUT	Input data field of the Use Case External Documentation
EXPECTED RESULTS	Exceptions field of the Use Case External Documentation

There are as many tables as execution cases in the test case.

There are as many execution cases as secondary scenarios (or sequence diagrams) associated to the use case in the model.

In addition to the descriptive fields of the execution case, we plan to generate an empty filed dedicated to the observed results.

	Execution case : Use Case name.
OBJECTIVES	Secondary scenario name.
PRE-REQUIREMENTS	Sequence from Primary scenario to the described secondary scenario.
PROGRESS INSTRUCTIONS	Secondary scenario description
INPUT	Incoming message to the secondary scenario.
EXPECTED RESULTS	Outgoing message to the secondary scenario.
OBSERVED RESULTS	

CHAPTER 3: THE DOCUMENT GENERATOR CONCEPTS

This section aims at introducing the fundamental concepts of the NEPTUNE documentation generator, namely transformation rule and shape. More than a definition of the concepts, this chapter, matching the path followed by the NEPTUNE team while designing the documentation generator tool, will present the concepts as logical answers to the problematic of documentation generation.

Transformation rules

An UML model is a huge piece of information. It is the repository of the whole modelling work performed by the project team, whatever the state of achievement of the modelling process (specification, design). As a consequence, inside a model, it may not necessarily be obvious to focus on a small part of the system, simply because the information we are interested in is embedded in more generic, wider topic information. To help through this problem, a first important step can consist in extracting from the model the sub parts where the useful information is hidden. However, the product of such an extraction is still carrying some useless information. A filtering operation can be a great help to get rid of this additional information.

Performing both extraction and filtering is the goal of the transformations as we define them in NEPTUNE. As a specific type of filter, each transformation specifies the rendering of the information.

This chapter aims at showing what a transformation is, what it does and how it works.

General transformation rules

The transformations are the smallest entities built and manipulated while designing some documentation. They can be split into two categories, depending on the kind of parameter they take as an input.

On the one hand, we have transformations that take an atomic part of the model as an input, this one being any kind of object, like a class or an association. The number of parameters of the transformation is not limited. This type of transformation is used to document one or more specific elements of the model.

On the other hand, we have the transformations that do not need any kind of model object as input but one single type of UML element. While processed, these transformations will successively deal with all the instances of the specified parameter. The documentation designer will use them for systematic treatment. For example, building up an exhaustive table of the actors defined in the system. These transformations are called meta transformations.

Now that we have justified the existence of the transformations, time has come to think about the formalism chosen in order to give concrete expressions of these transformations.

The UML models manipulated in NEPTUNE are in their exchange format (XMI). As XMI is an XML format, and knowing that XSL is known to be the reference for manipulating some XML documents, it then appeared like an evidence that XSL was the most adapted language for the writing of our transformations. What is also interesting in XSL is the whole set of graphic libraries associated. These libraries can be used while designing a transformation.

Furthermore, the spectrum of XSL dedicated tools is really wide. These tools are often free of charge and open source. Besides, several of these pieces of software are already approved and widely recommended by the community dealing with XSL for a while. Among these tools, we can cite the XSLT transformation engines SAXON or XALAN as significant examples.

The use of XSL has also increased the number of features of the transformations. For example, whether they are meta or not, the transformations can deal with some external pieces of documentation, provided these pieces of documentation are written in XML. For instance we can imagine a transformation which aims at exhaustively extracting the text inside a specified hierarchy of XML tags.

To make things clear, the following chapters present some examples of transformations, organised by business.

The shape

It's been already showed that the transformations are the means used to transform the information contained in a model. We still have to find how to use it and how to organise it. If any document has a semantic content, it also has a structure. If we consider the notion of title, the chapter numbering, or even the styles, as the transformations do not handle these essential features, it is vital to define another concept to do so. This is the role of the shape.

The shape constitutes the higher designing level of documentation. It is the most significant view of the final documentation and the main interface with the documentation designer. If the transformations are the tools, the shape will be the toolbox.

Shape Content

To get an elementary piece of information (The content of a chapter, for instance), a transformation must be associated to at least one UML modelling element and eventually some external pieces of documentation. The NEPTUNE document generator embeds all the facilities to make these associations.

Shape Structure

During the design process of a document, the organisation of the information leads to the definition of a structure, which means titles, subtitles, and styles applied to the various levels. This structural information is handled by the shape.

CHAPTER 4: NEPTUNE DOCUMENT GENERATOR

With the emergence of UML as a standard in software development, software projects are now very often based on this technology. This statement is true in many business fields, like software engineering of course, but also for knowledge management or business process. Although all the participants in a software-engineering project may be able to read and understand an UML model, this does not necessarily happen with people in other domains. Thus, it is useful to transform the information contained in a UML model into different representations, easier to read for someone not necessarily being fully skilled in UML. The main objective of the NEPTUNE [ref11] documentation generator is to produce some professional documentation that results from the exploitation of UML model sub parts. This documentation is an end-user-oriented documentation, which takes into account the professional expertise of the reader. It is the result of transformations applied on UML elements available in the actual model. The transformations turn the information expressed with the UML formalism into textual easy to read information. It is important to say that there is no need to be UML skilled to understand the documentation produced. This makes the tool not specifically dedicated to software engineering, but also to other business fields like knowledge management or business process.

Throughout this chapter is detailed the NEPTUNE documentation generator. The description of the tool is given with a sequential approach of the documentation generation process: inputs, processing chain, outputs…then after is given the tool user approach, based on a wide GUI description.

Tool description

Though the NEPTUNE user may have a more or less clear idea of the look like documentation he wants to produce, he has to provide several inputs to the document generator core. Some of them are mandatory, and some are optional. This set of inputs constitutes the information source used by the document generator core to function. The available information will be manipulated so that it can be organised and transformed as specified by the user. To understand how it works, this section will introduce some fundamental NEPTUNE concepts. The objects presented are the ones to be built and gathered together by the user in order to specify the documentation to be generated. Once these concepts will have been be assimilated, we will focus on the key steps of the generation process. To conclude, we will give a presentation of the outputs of the document generator.

Inputs

This chapter aims at presenting the inputs of the documentation generator. We will successively introduce the UML model, and UML meta model, which are two mandatory inputs. Then after, we will talk about the eventual external documentation and the documentation templates provided with the NEPTUNE tool.

The UML model

The UML model is the most important input. As NEPTUNE is case tool independent, a model should easily be loaded, whatever the case tool used to design it. As a consequence, XMI seems to be the most suitable representation's format, as it is the UML exchange format. Thus, it is the one we chose to use in NEPTUNE. The XMI specification is part of the UML recommendation, which means there are significant differences between the XMI format from one UML version to the other. As the NEPTUNE tool is UML 1.4 compliant, the user will pay attention to the UML1.4 compliance of its XMI file.

The UML metamodel

The UML metamodel is systematically loaded during the NEPTUNE sessions and displayed in the metamodel browser. Thus, for systematic treatments (over all actors for example), the documentation designer will not have to select all actors one by one in the model but only the actor concept in the metamodel browser.

The existing documentation

The NEPTUNE document generator gives the opportunity to make use of XML external documentation. While designing the final documentation, the user can make references to this external documentation, which content will then appear inside the produced documentation, bringing in additional information. This external documentation can be pure external documentation or documentation associated to the model in the tool case itself, thus embedded in the XMI representation of the model.

Of course, it may not be relevant to extract and use the whole document. Any user may some day needs to pick up small pieces of documentation inside of a bigger document. To do so, NEPTUNE offers a filtering feature, allowing the selection of document parts. To do so, the user has to specify a hierarchy of tags among the ones used in the external documentation specified as input of the document generator. Then during the generation process, only the text contained in this hierarchy of tags will be extracted for contribution to the generated document.

The standard models of documentation

Though each project has its own specificity, if we focus on the documentation topic, for each business field, we can identify a standard set of documents. These documents are the product resulting from the sharing of a particular knowledge, a common culture, a common vocabulary….

Regarding software engineering, a good example is the validation plan. Even if the form may change from one organisation to the other, the content is always based on the same information. Indeed, in a UML context, the use cases will always define the validation cases and it is true that there should be as many sequence diagrams as test cases.

What the NEPTUNE tool offers is a selection of templates (we call them shapes in the tool) corresponding to these standard documents. Feeding the documentation generator core with one of these templates properly parameterised and then applied on the UML model will lead to the generation of the awaited standard document.

The shape as an input organiser

In the previous paragraph, we have mentioned the standard models of documentation. If we use the NEPTUNE vocabulary, these models are called standard shapes. In the following section, we will develop much further the concept of shape, whether it is considered as standard or fully designed by the NEPTUNE user. We will detail each of the shapes' components, so that the reader becomes familiar with the concepts of style, documentary element, all widely developed here after.

To fully describe the shape, we chose to adopt a bottom top approach. In other words, we will first describe the atomic constitutive objects of a shape, before gathering them together to build up a bigger element, itself merged with one other object to finally constitute a full shape.

XSL elementary transformation:

Though we have already presented them in a previous section, it can be worthy refreshing the reader’s memory.

These transformations, written in XSL, are able to perform the extraction of information from the model and to define the way this information will be presented. It can be with the means of arrays, with a particular size of column, etc. These transformations can also be seen as a way to generate different views of a same piece of information, and by extension, different views on a model.

Documentary element

A documentary element is composed of the whole information needed for a full paragraph generation, expressed at the shape level. To define a documentary element, the reader has to associate one or more elements of the UML model with an XSL transformation. The UML model elements are the parameters of the transformation. A documentary element can be defined at the metamodel level – meta documentary element – or at the model level – documentary element -. The main difference is the level of appliance of the XSL transformation. A model level documentary element needs an instance of object as parameter of the transformation, whereas a meta documentary element only needs a type of UML element.

Whether it is meta or not, the documentary elements constitute a raw brick of a higher level concept in NEPTUNE, namely the shape.

Structural elements

The structural elements are used to insert titles or sub titles, with text and styles. They define organisation of the documentary elements all together, with features such as chapters, sub chapters… The access to the structural elements is done during the shape design. They can easily be inserted, removed, or modified.

Shape

The shape is a frame containing the structure of the documentation the user will produce, together with some documentary elements. The shape can be defined at metamodel level – the generic shape - in order to provide generic models of documents, and at model level – the user shape - in order to be fully adapted for the user’s needs. A user shape may be defined from a generic shape and extended by the user. The generic shapes will lead to the generation of standard documents, often dedicated to a particular business.

Internal Generation process

The previous paragraph has shown what the shape was made of. Now that we have it built or loaded from a library, it is time to launch the automatic generation. Even if this generation is hidden behind a single mouse click, the process is indeed performed in several sequential steps, all described throughout this section.

From a Shape to a User Document Definition (UDD)

The first step consists in a first interpretation of the shape as specified by the user. It is turned into an XML document called UDD. The tags of this document are pre defined in the tool. This operation aims at making the shape information compatible with a second interpretation made by an XSL processor.

From a UDD to an XMI processing style sheet, using the generic UDD processing style sheet

As the UDD is an XML document, it can be transformed using the XSL technology. This is what we do thanks to a generic style sheet, applied on the shape in its UDD form. The main object of this operation is to replace the references to XSL transformations by the real XSL text of the transformation. The document obtained is thus a real XSL style sheet, let's call it the XMI processing style sheet.

From the XMI original file to the first XML output document using the XMI processing style sheet

The XMI style sheet previously obtained is ready to be applied on the model. Both XMI style sheet and model (XMI file) are inputs of the XSL processor, which output is a first XML version of the document to generate as specified by the NEPTUNE user.

From an XML output document to any available output format using the NEPTUNE format transformer feature.

The last operation consists in transforming the first XML document into a smart document. This last product can be generated into html; but also rtf or pdf.

Outputs

Let's first remind the reader that the NEPTUNE technology is based upon XML: The input format is XMI, which is nothing else but XML, the transformations are written in XSL (XML again), the internal forms are also XML (see previous chapter)… In other words, the whole stuff is relying upon XML. Consequently, the generated documentation should logically be pieces of XML. It is true, and we can even add that the XML format was the first one to be supported by the tool while it was still being developed.

Then after came some additional formats, and today, the tool is an enabler HTML, RTF, and PDF. The production of these new formats does not constitute a different chain of generation from the one presented in the previous paragraph but an additional step in this process: Indeed, the XML produced by the last step is transformed with the help of XSL (one more time). For the readers the most interested in XSL, it can be interesting to know that the libraries used to do that transformations are FO.

Tool Usage

Now that we have already seen all the elements needed to produce the final document, we are ready to use this knowledge within the NEPTUNE tool, so that we will be able to configure the documenting process to make it generate documents, which will fit whatever requirement.

As previously shown, the NEPTUNE document generator makes use of four different elements to produce the final document. All these inputs can be designed within the NEPTUNE tool thanks to an easy, integrated and ergonomic interface, described hereafter. Moreover, we will see that NEPTUNE has been developed to let the user manage each of these elements independently from the others, so that each of them can be used for any future generation of document. This is way, the final efficiency increases as there is reusability of all components and the final configuration of all the generated documents can be readjusted implying only modifications in its minor parts. As an example of this, changing a Transformation's behaviour will also be taken into account in the next generation, or one shape may be applied with as many UML models as wished.

This section is, consequently, organised covering all subjects of the NEPTUNE document generator feature's to be consulted independently. But before going on, it is important to remark right now that all Neptune's elements initially provided has been also designed by means of the Neptune Tool, so the interface has been fully tested and it is proved to fulfil all the possible requirements.

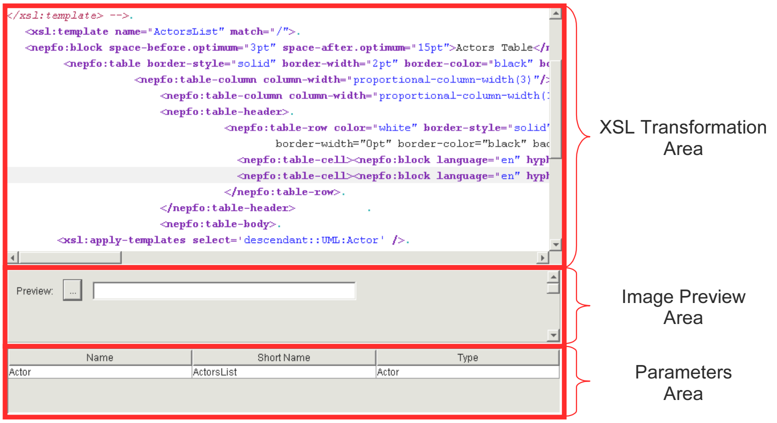
The transformations

Overview

The transformations are the basic elements that give all the functionality to the NEPTUNE document generator. In other words, we can say that the efficiency of the tool will improve respectively to the number of transformations available.

Consequently, although NEPTUNE provides a set of transformations, it also offers facilities to re-design them or create brand new ones, fully adapted to the use needs.

An overview of the transformation designer is shown below:



```
</xsl:template> -->.
   <xsl:template name="ActorsList" match="/">.
   <nepfo:block space-before.optimum="3pt" space-after.optimum="15pt">Actors Table</
          <nepfo:table border-style="solid" border-width="2pt" border-color="black" b
                      <nepfo:table-column column-width="proportional-column-width(3)"/
                          <nepfo:table-column column-width="proportional-column-width(
                          <nepfo:table-header>.
                                    <nepfo:table-row color="white" border-style="solid"
                                           border-width="0pt" border-color="black" bac
                                      <nepfo:table-cell><nepfo:block language="en" hyp
                                      <nepfo:table-cell><nepfo:block language="en" hyp
                                    </nepfo:table-row>.
                          </nepfo:table-header>           .
                          <nepfo:table-body>.
         <xsl:apply-templates select='descendant::UML:Actor' />.
```

Name	Short Name	Type
Actor	ActorsList	Actor

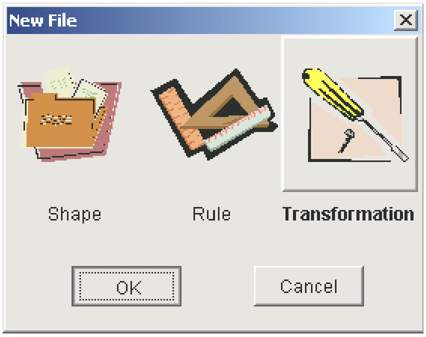
Transformations Designer

As we can see, the Transformation Designer is divided into three different areas:

- **The XSL transformation area**: it is the component area where to write the XSL transformations. Neptune will automatically highlight the text according to the syntax.
- **The image preview area**: this area is designed to provide a preview of what the transformation does. The user will thus be aware of the rendering of the transformation even before applying it through the documenting process.
- **The parameter area**: this area is the edition zone of the parameters of the transformation.

Creation of new transformations

To build new transformations, go to "`File | New`", select the "`Transformation`" option., and click on the "`OK`" button.



Creation of Transformations

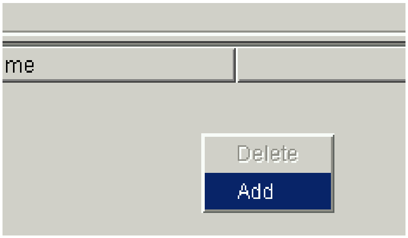
This operation will open a new blank transformation designer with its three zones. The first required step then consists in filling the XSL transformation area. The syntactic XSL analyser embedded in the tool is running in the background, which is a great help while designing a transformation.

Due to the fact that no preview of the Transformation's result has been caught yet, usually no image cannot be attached in the creation process, although it can also be assigned later when having any output result. This Transformation's image preview is useful when designing the shape, because it gives an idea on what results the Transformation produces. However, assigning an image preview is not mandatory and it can be done after the transformation has been executed or even it can be omitted.

To attach an image, click on the ".." button and select the image in the local file system.

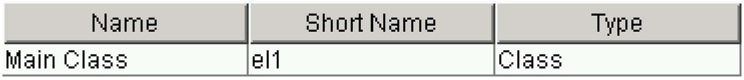
Within the parameter area, the transformation designer has to specify the parameters of its transformation. They which are the UML elements required to correctly run the transformation. If the transformation designer has properly defined these parameters, once used in the document generation process, the transformation will need to be associated to UML elements which types match the ones previously defined.

To add or delete one parameter, simply use the right click button. This action will pop-up the following contextual menu item.



Adding new Transformation's Parameters

Once it's been chosen to add a parameter, the corresponding area is enhanced with one additional row. Each parameter (each row) has tree columns:



Name	Short Name	Type
Main Class	el1	Class

Parameters' Fields

- **Name:** it is the name of the parameter and should be human-understandable, as it will be the main parameter's identifier when using the transformation in future.
- **Short Name:** it is the exact string used to identify the parameter used in the core of the XSL transformation.
- **Type:** it is the UML type of the parameter.

Depending on the parameters' types that the Transformation has, it can be grouped into two different categories:

- **Meta Transformation**: This sort of Transformation has only parameters of type "XML Files" or no parameter at all. As it does not have any model-dependent parameter (such a Class or an Actor, for example), the Transformation applies to meta-classes as defined in the XSLT, thus implying that all elements on the model are processed.
- **Normal Transformation**: This other sort of Transformations, on the contrary, contains at least one parameter different from XML files and the user has to provide some model element of its type when using the Transformation.

Once all these fields have been filled, the transformation is fully defined. It can be saved by choosing "`File | Save`" from the menu. The user has to give both a name and a description of the transformation, shortly explaining what it's been designed for. From that point, the transformation becomes part of the actual project. Consequently, it will appear inside of the "Transformations" node of the project explorer (`Ctrl-E`).

Modification of transformations

Modifying a transformation is a task that can be necessary to adjust its final behaviour to the user needs. The aim can also be to attach the preview image once the transformation has been processed, and its output turned into a relevant view of the rendering.

As it happens with all the NEPTUNE's elements being part of the global storage, the transformations can not be modified, simply because the global storage location means that they are supposed to be fully functional and tested. However, the modification of a transformation is possible, but it is first necessary to import it into a project using "`Project | Add Files…`" menu.

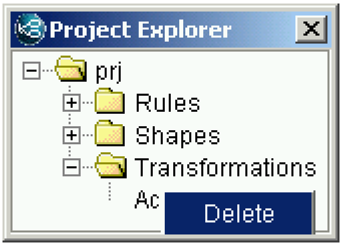
To edit a transformation, use the"`Project | Explorer`" menu, and double click on the transformation to modify.

Then, the transformation designer is opened showing the content of the transformation. If an image has previously been attached to the transformation, the preview is displayed though its path field remains empty. This is because the image is fully part of the transformation. It is not only a link to the file.

Any modification can then be performed on the transformation, that can be saved at any time by pressing Ctrl-S.

Deletion of transformations

On the Project Explorer window (`Ctrl-E`), look for the transformation to be deleted and right click on it. In the pop-up menu choose "Delete".



Deletion of Transformations

Exporting a Transformation to the Global Storage

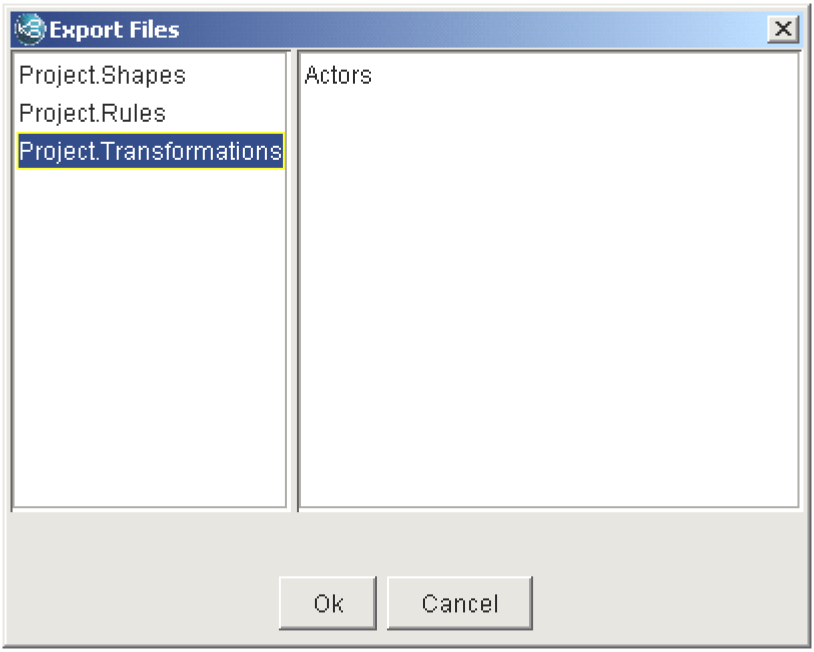
All the transformations created by the user are project dependent. This means that when closing a project and opening another one, the UML model and the various elements available in the project interface (Transformations, Shapes and Rules) are the ones attached to the last opened project

However, the elements stored in the Global Storage, which the ones provided initially with NEPTUNE are part of, will remain completely project-independent and available all the time. This notion of project dependence is a first justification of the export feature available in the tool

Indeed, it seems quite easy to identify other good reasons for transferring some elements from a local (or project) storage into the global storage. For instance, when a user has designed a transformation that should be shared (it may be useful for another person in the company, not necessary involved in the same project), then the transformation should be exported into the Global Storage.

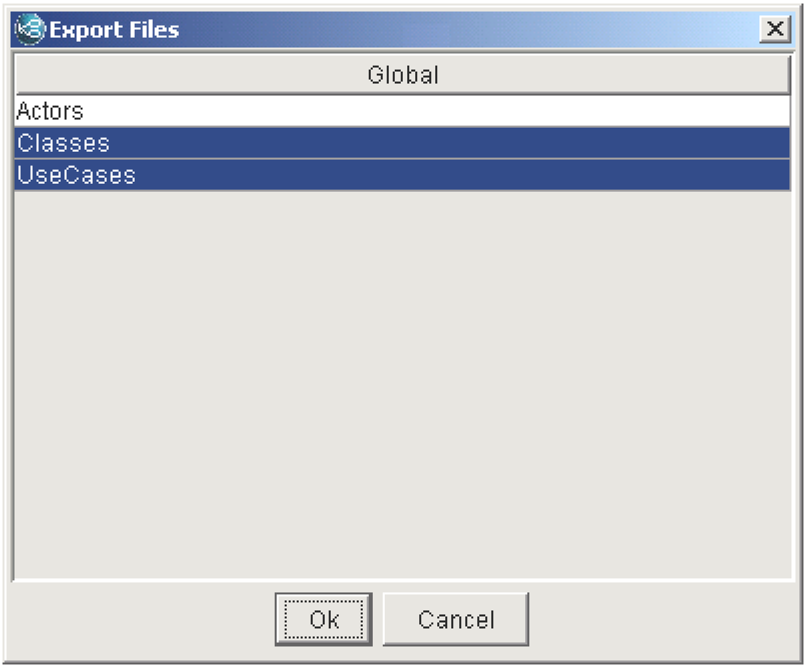
This Global Storage is categorised, the information is organised, easing the access to the elements

To export a transformation, use the "`Project | Export`" menu and then select the transformation (or several transformations holding the `Ctrl` key) to transfer to the global storage.



Browsing the Local Elements

The next step consists in clicking on the "OK" button. All the available categories of transformations then appear on the screen An other choice has now to be made in order to select the destination category of the transformation. It is interesting to notice that several categories can be selected, thus defining multiple locations of the transformation to export.



Global Storage Categories selector

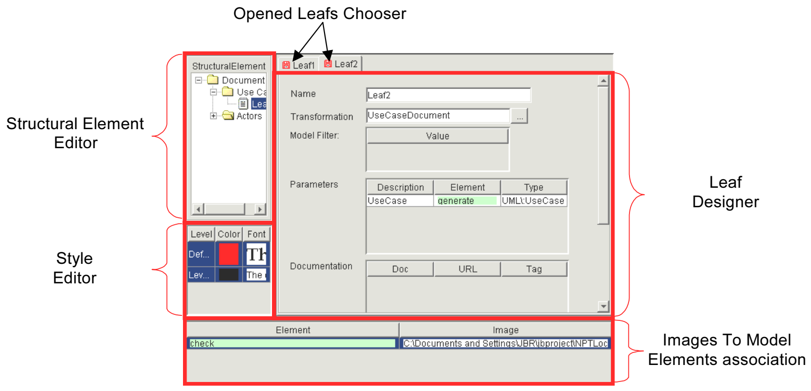
Of course, the creation of new categories is also available: Right clicking on the window above will launch the devoted interface.

The Shapes

Overview

The shapes configure the formatting of each document. They are interpreted by the core of the NETUNE document generator. They contain the entire set of instructions needed to launch the information extraction from the user's UML model. Indeed, the shape regroups the singular functionality carried out by each transformation. They also embed the information needed to organise the output document in chapters, sub chapters…

As for the transformations, a design feature devoted to the notion of shape is available in the tool (see next figure).



Name: Leaf2

Transformation: UseCaseDocument

Model Filter:

Value

Parameters

Description	Element	Type
UseCase	generate	UML::UseCase

Documentation

Doc	URL	Tag

Level	Color	Font
Def...	🟥	Th
Lev...	⬛	The

Element	Image
check	C:\Documents and Settings\JBR\jbproject\NPTLoc

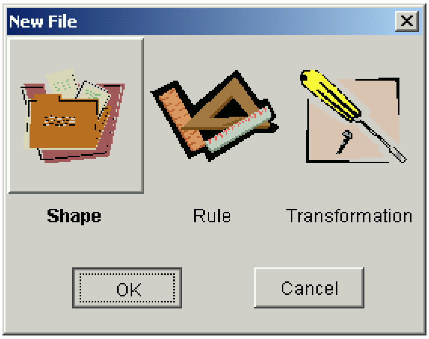
Shapes Designer

As we can see, the shape designer is divided into five different areas:

- **Structural Element Editor:** in this area the shape's structure is defined, thus allowing the creation of new chapters and the selection of leafs where to attach the transformations.
- **Style Editor:** this is the interface dedicated to the styles' definition for each level (chapter, sub chapter…) For each level, the user can choose a size, a font, a colour…
- **Leaf Designer:** this area defines which transformation is attached to each leaf, with its set of parameters (if any).
- **Images to Model Elements Association:** it contains a table where to assign icons to each UML model element in particular, in order to be represented as specified from here.
- **Opened Leafs Chooser:** all of the opened leaves appear here as different tabs, so that the user is given a quick access to each of them.

Creation of new Shapes

To build new Shapes go to "`File | New`", select the "`Shape`" option, and click on the "`OK`" button.



Creation of Shapes

When the shape Designer is opened, an empty tree in the structural element editor appears. No Leaf is opened, no image is neither associated and only the default level is available within the style editor. This state of the shape designer is typical of a blank shape. It is the starting point of the creation of a new shape.

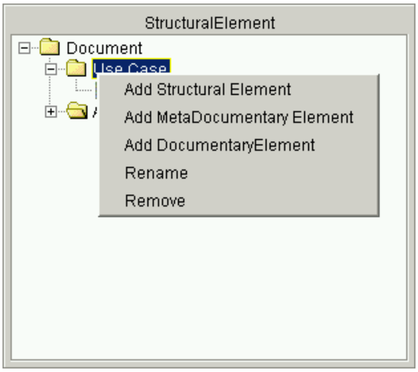
The first thing to do is to build the document structure by means of the Structure Element Editor. Each folder in the tree represents a chapter level, which can contain:

- **A Structural Element:** it represents a new chapter of the document structure.
- **A MetaDocumentary Element**: it contains a Meta Transformation, applied to all the metaclasses of the UML model.
- **A Documentary Element:** it contains a Normal Transformation that will apply to any particular user-selected model elements.

In the next chapters we will see how to handle these elements.

Adding Structural Elements

To add, remove or rename Structural Elements, right click on the parent folder (a Structural Element, in fact), and choose among the available options of thethe pop-up menu:



Modifying the Document's Structure

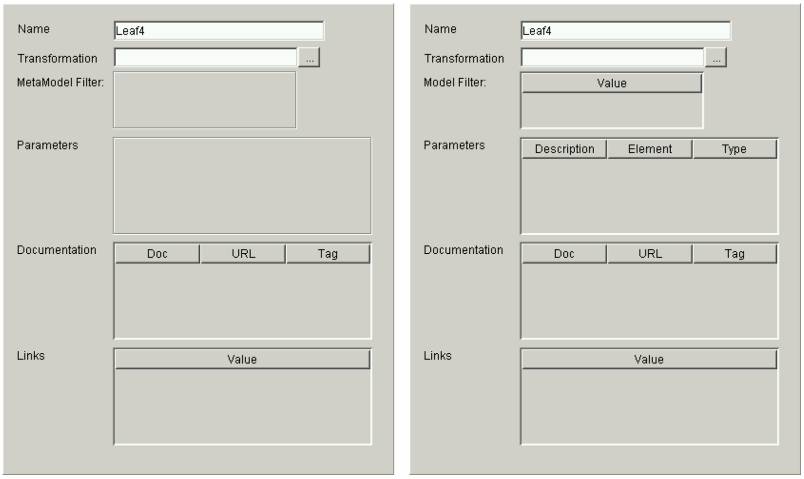
In this case, we will select “Add Structural Element”.

Remark: the rename operation can be performed by pressing `F2` key when the node to change has already been selected (highlighted in blue).

Creating a Documentary Element

New Documentary or MetaDocumentary Elements (Leaves in any case), can be created by choosing the corresponding option after doing a right-click on the parent Structural Element (see previous chapter to get more details).

When the new leaf is created, a new Leaf Designer will open.


Name: Leaf4

Name: Leaf4

Meta & Normal Leaf Designers respectively

Both Meta and Normal Leaf Designer work in a similar way, but due to the fact that they address different targets, it would for instance not be relevant to insert a Meta level transformation inside of a Normal Documentary Element. Because of this, the content of the window dedicated to the choice of a transformation is already filtered and the Transformations displayed in there are the ones belonging to the appropriate family.

The Leaf Designer is composed of the following fields:

- **Name:** The string chosen here also appears on the Structural Element Editor tree.
- **Transformation:** it is the field where to attach the Transformation to be used within this Leaf.
- **MetaModel & Model Filter:** it is used to filter the results of the Transformations Chooser, the previous field indeed.
- **Parameters:** it is the field which lists all the required parameters of the previously chosen transformation As already mentioned, these parameters refer to specific UML model elements and only exist in case of Normal Transformation.
- **Documentation:** this field is the location where to attach the external documentation file possibly required by the Transformation.

In the subsequent subchapters we will focus on the parameter fields, and detail how they can be filled.

Entering the Name

To change the name of a Leaf (usually named LeafX when created), select the former one with the mouse and type the new one.

Choosing the Transformation

The assignation of a Transformation is a required step of the Leaf definition. The available Transformations are either located in the Project or the global storage.

To launch the interface giving access to the Transformations, use the "…" button on the right. This action shall pop-up the Transformation Explorer, where all the available Transformations will be grouped by categories displayed on the left column.

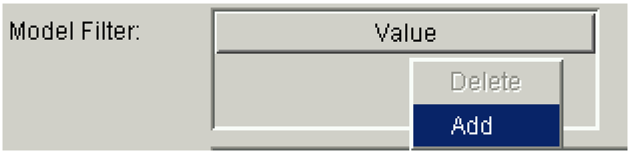
Once a category has been selected, the Transformations it contains will be displayed on the right. Select any of them and click on the "OK" button.

The Transformation Explorer will be closed and the Transformation will be transferred to the Leaf designer.

Adding Transformation Filters

The set of transformations available for attachment (previous section) can be the result of a filtering action. To makes things clear, let's see that, for instance, it is possible to set a filter which effect is to restrict the set of transformations to the ones that have a use case as parameter.

To add a filter, perform a right-button click on the "Model Filter" box and select "Add":



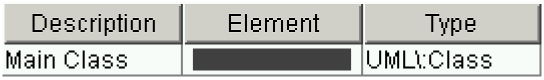
Adding Filters

A new dark grey row will appear in the table. Dropping an UML element into that zone is the same as choosing a filter on the transformations. To do so, simply drag from either Model or MetaModel explorers, any Model Element into that row. As an example, we can analyse the effect produced by dragging from the MetaModel explorer one of the elements contained under `ModelManagement/Package/AllInstances`. As a result, the only Transformations listed in the Transformation Explorer are the ones that have packages as parameters.

Adding Transformation Parameters

When a Transformation applies to specific Model Elements, these Model Elements must be selected by means of this table.

There are as many Model Elements to specify as parameters defined in the Transformation chosen. Here is an example:



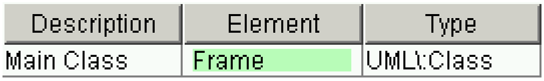
Description	Element	Type
Main Class	■	UML\\:Class

Transformation Parameters Field

The columns are detailed hereafter: **Description:** The text displayed is the one typed by the Transformation creator. It is thus under its responsibility to make this text self-explanatory.

- **Type:** specifies the required type of each parameter. In the previous example, only instances of Classes are allowed to be dragged into the field.
- **Element:** this is the column where to drop the parameters required. Their type must be the one given in the "Type" column. They are picked up from either the Model or MetaModel Explorers.

Once an element of the correct type has been dropped into the dedicated field, it is displayed on a green background.

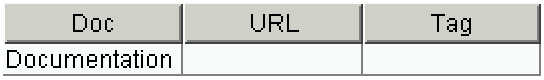


Description	Element	Type
Main Class	Frame	UML\\:Class

An UML Element attached

Adding Documentation

Concerning the Transformations that are enriched with external XML documentation, a table is provided by the GUI, which number of rows depends on the chosen transformation, as some require one single piece of documentation while some others need more. In other words, there are as many rows as pieces of documentation specified by the Transformation creator.



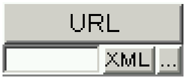
Doc	URL	Tag
Documentation		

Documentation Field

Each columns stands for:

- **Doc:** contains the label for each document required and is shown as typed when defining the Transformation.
- **URL:** is the field where to type the path the XML file containing the document.
- **Tag:** contains the path within the XML tree where to extract the information for this Leaf.

To choose the XML file, perform a click in the right of the field. Two buttons will appear:

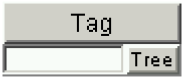


Choosing the XML file

The "…" button will open a File Chooser window and will browse the file system to attach the XML file.

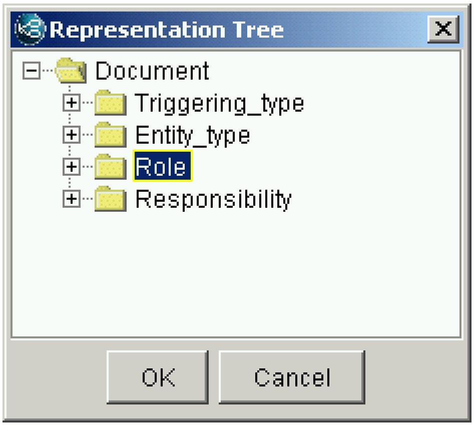
The "XML" button will open the external XML Editor as configured in "Tools | Configure Neptune".

Once the file has been attached, its XML structure can be browsed by clicking on the right side of the "Tag" field. This will make appear a "Tree" button:



Opening the XML tree window

The "Tree" button will open a new window showing a tree with the XML structure:



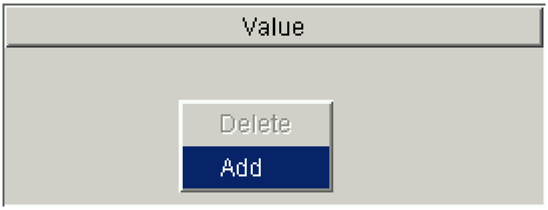
Choosing the documenting tag

When chosen the folder where stands the root of the information to attach, you can click on the OK button and this path will be transferred to the field.

Adding Links between Leafs

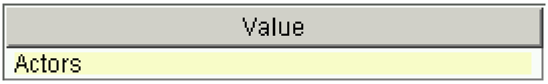
The links are the last missing element concerning the Leaf and let joining information; so similar Leafs can be grouped.

To add links, simply perform a right-click in the links box and choose "Add":



Adding links between Leafs

This way, a new dark grey row will appear. To assign another Leaf, drag the related one from the Structural Element Editor and drop it into the row. The Leaf's name will be cached and displayed in yellow:

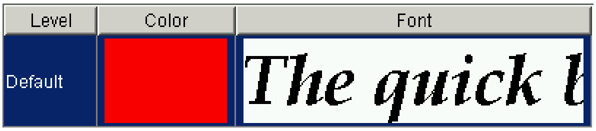


A link between Leafs

Defining the Shape's Style

By default, when creating a new Shape only one style level is available, which is named as "Default". Each of these levels applies to the corresponding level of the Shape's structure starting from the root node.

Once the Shape's Structure has changed, it is habitual to define a new style for the new coming chapters. This task is accomplished by means of the Style Editor:

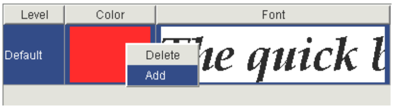


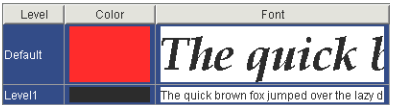
Style Editor row

The Style Editor has three columns:

- **Level:** contains information on the style level it is defining. It must be either "Default" or "LevelX", where X is a number greater than 0 pointing to the right structural level.
- **Colour:** assigns the font foreground colour.
- **Font:** let the user specify the font type and shows a preview.

To add or delete rows, click on the table with the right button and choose the wished option:

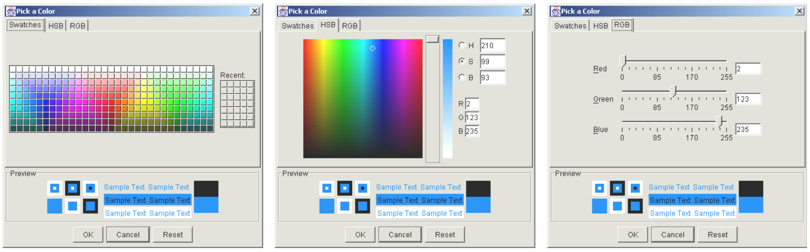




Adding new styles

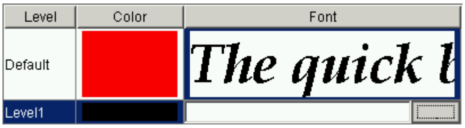
To edit the colour, click on the colour cell and a colour picker dialog will appear. You will be given the opportunity to choose colours from a range of three different ways:

- From a swatch, thus choosing one of available colours.
- Choosing the hue, saturation and brightness.
- Choosing the red, green and blue components.



Choosing colours

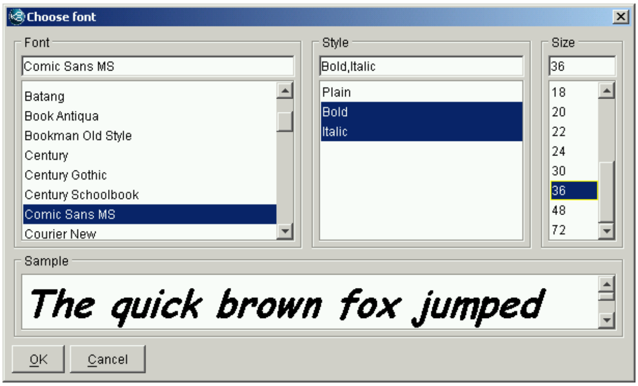
The font can be defined performing a double-click on the right side of the font cell. This operation will make appear a button:



Level	Color	Font
Default	🟥	The quick b
Level1	■	

Choosing the font face

A Font Chooser dialog will also appear showing all system available fonts:



Choose font

Font	Style	Size
Comic Sans MS	Bold,Italic	36

Font: Batang, Book Antiqua, Bookman Old Style, Century, Century Gothic, Century Schoolbook, Comic Sans MS, Courier New

Style: Plain, Bold, Italic

Size: 18, 20, 22, 24, 30, 36, 48, 72

Sample

The quick brown fox jumped

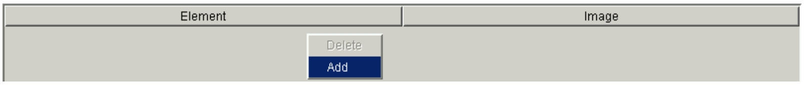
Defining the font appearance

The Font can be chosen on the left, the style on the middle (if allowed by the font multiple styles can be selected, as in the image by holding the Ctrl key) and the size on the right.

Attaching images to elements

This step lets the user specify specific icons to metaclasses or even particular elements on the model and the definition is performed by using the Images to Model Elements bottom part of the Shape Designer.

The way to add new associations is to with-click into the table and choose “Add”:

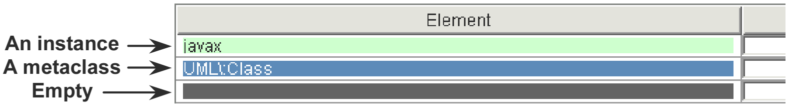


Adding images associations

The meaning of the two columns is the following:

- **Element:** contains the metaclass or the particular UML Model Element associated to the image.
- **Image:** contains the image associated.

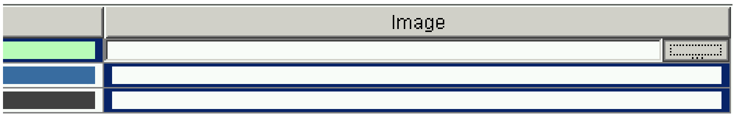
This will create an empty dark grey row. The way to attach the elements (an instance or a metaclass) is dragging and dropping them into the Element cell from either Model or MetaModel Explorer. Depending on the colour the dragged element can be:



The three sorts of rows

If we use the previous figure as example, there is attached the package “javax” and the metaclass “Class”, so the association will be used in any Class instance of the current UML model.

To attach the image, we must perform a double click on the right side of the corresponding image cell. This action will make a button:



Attaching images

After pressing the button an Image Browser will pop up showing the file system:

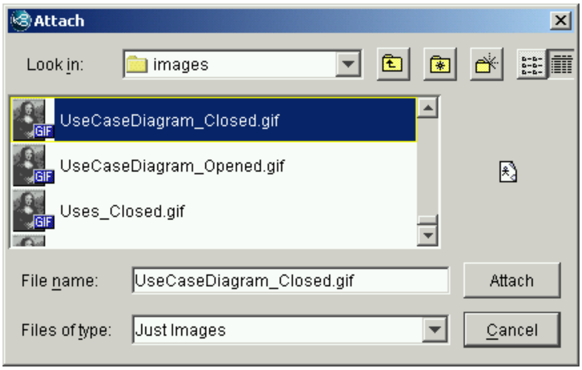


Image Browser

When selecting the image and clicking “Attach”, the image selected will be transferred. This image can be created temporally and there is no need to make it available in the generation process, so it can be deleted after attaching it.

Modification of Shapes

The process of modifying a Shape is to be carried out usually when creating new Shapes in order to fit its behaviour.

Only Project's elements can be modified as the others, as they are in the Neptune Storage, are supposed to be well working. Anyway these global elements can also be modified by importing them into the project by choosing "`Project | Add Files…`".

The edition of Shapes requires it to be opened. This action is performed by using the Project Explorer (choosing "`Project | Explorer`" from the menu) , and double-clicking onto the Shape.

Note that images associated path will not be visible because they do not need to exist in its former location.

Deletion of Shapes

To delete a Shape from the Project, open the Project Explorer (press Ctrl-E) and choose "Delete" from the contextual menu that will appear after doing a right-click to the Shape.

Exporting a Shape to the Global Storage

The process of exporting Shapes to the Global Storage can be useful to share user-defined shapes in other Projects, as they are available always to any project. Anyway, we must take into consideration that the Shapes that are to be exported cannot make any references to particular UML model elements, as it would have no sense to generate documentation when changing the UML model.

The working way is as early explained in chapter called "Exporting a Transformation to the Global Storage", so the first step is to choose "`File | Export`" from the menu and chose the Shapes (several can be selected holding the `Ctrl` key).

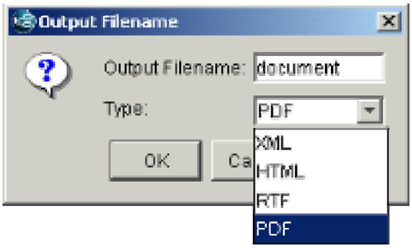
After clicking on the "OK" button, you will be given the Shapes categories on the Global Storage. In this case, as before, the procedure is the same as in chapter called "Exporting a Transformation to the Global Storage", so several categories can be selected (indicating that the Shape can be regrouped into several categories), and new ones can also be created.

Invoking the Generator

When having already all elements concerning the generation process in the Neptune system (it is the Transformations, the Shapes and the UML Model) , the generation of documents can be performed.

To do it, go to "`Tools | Generate Documentation`". Neptune will show a Shapes Explorer with all Shapes in the Project and with all the global ones categorised. When each Shape is selected on the bottom of this window will be displayed a description of it.

The next step is to type the output filename and choose the output format:



Selecting output document

Neptune can export the results in the following ones:

- XML
- HTML
- RTF
- PDF

The pressing the "OK" button, Neptune will work for a while and will show a window indicating that the Document Generator is working:

Document Generator working

The document will be created inside the folder configured for output documents into the Project's directory. The Project's directory can be known by choosing "`Project | Properties`", and the output directory by means of "`Tools | Configure Neptune`".

APPENDIX A

Overview of UML META model and MOF

INTRODUCTION

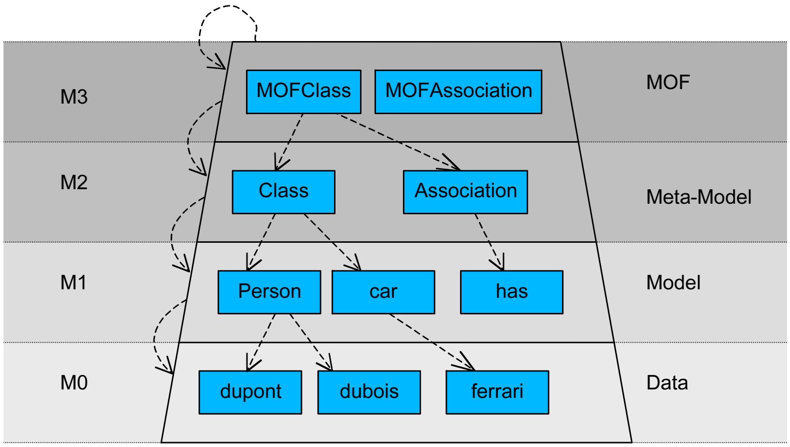
The UML meta model is presented in terms of UML concepts [ ], giving a metacircular definition of this modelling language. In order to have this description, only some concepts of UML have been considered as relevant. The set of these concepts constitute the kernel of the definition of the so-called Meta Object Facility (abbreviated MOF). The MOF has been adopted and standardised by OMG, and is intended to be the universal meta model language *i.e.* the language capable of describing languages such UML, relational models and so on.

Let us recall that MOF is resting on a four-level standardised architecture; these four levels conventionally denoted M0, M1, M2, and M3 are defined below:

- M0: information level
- M1: model level
- M2: meta model level
- M3: meta meta model level

M0 is devoted to objects and links. The M1 level concerns a model, for instance in UML, it consists of class diagrams, sequence diagrams, use cases, *etc.* Then higher up, appears the M2 level which is any modelling language; in our approach the UML language, itself. The top level M3 contains the language whose purpose is to describe any modelling language that appears at level M3. Thus, each level can be considered as described by the next level up. In order to avoid an infinite number of levels, it was decided by OMG to make M3 reflexive. The consequence is that the MOF must be able to describe itself.

The four level architecture proposed by OMG can be summarised by the following scheme.



THE BASIC CONCEPTS OF MOF

Each MOF element is represented by a meta entity. Globally, the MOF possesses about twenty meta entities. However, four of them are sufficient to describe the entire MOF. Consequently, four meta entities are enough to describe the entire M3 level of the architecture.

These meta entities are:

- Class
- Association
- Data type
- Package

Let us now give a more detailed description of the above meta entities.

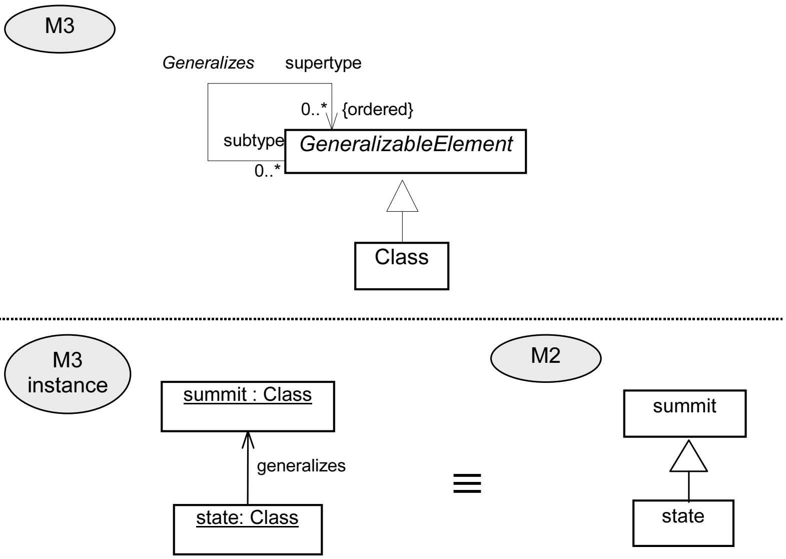
The concept of a MOF class

At the M3 level, a MOF class is a meta entity that allows the definition of meta objects *i.e.* objects that occur at level M2. At the M3 level, a MOF class permits the definition of all the meta-entities at level M3. A MOF class is similar to an UML class. It contains attributes, methods, associations, *etc.* A MOF class can be abstract. MOF supports single and multiple inheritance hierarchies as well.

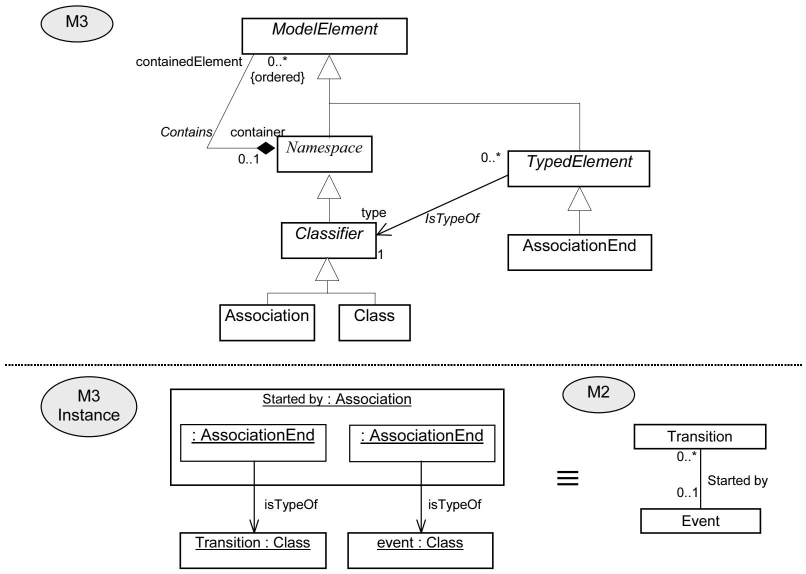
The concept of a MOF association

A MOF association is a meta entity that is intended to express binary relations between a pair of MOF classes. Is is important to distinguish the MOF associations that describe the relations between MOF meta entities and the so called meta association Association which is devoted to implement relations between instances of the MOF meta entities.

At the M3 level, the MOF associations describe relations between MOF meta entities. For instance, the association "Generalizes" implements the inheritance relation between the so called "GeneralizabeElements".



The MOF meta entity "Association" is used to establish binary relations between instances of MOF entities at level M2. For instance, in UML, a transition that occurs in a state-transition diagram can be fired by an event. This particular relation is instantiated by the meta entity "Association".



As the MOF is reflexive, the inheritance relation between the M3 entities “Namespace” and “ModelElement” is also implemented by the meta entity “Association”. In this situation, this relation belongs to level M3.

At any level, modularity is an essential concept which is intended to package scattered information. Among the benefits of modularity, appear the following points:

- regrouping and classifying elements
- simplification of inheritances, here inheritance processes a group of elements
- simplification of dependence links between groups

MOF modularity is ensured by the meta entity “Namespace” which can contain ModelElements through the relation “Contains”. This relation is depicted by the next figure:

Containing Contained	Packa ge	*Class*	*Datatype*	*Associatio n*	*Operatio n*	*Exceptio n*
Package	X					
Class	X	X				
Datatype	X	X				
Association	X					
Attribute		X				
Reference		X				
Operation		X				
Exception	X	X				
Parameter					X	X
Association End				X		
Constraint	X	X	X	X	X	
Constant		X				
Alias			X			
Import	X					
Tag	X	X	X	X	X	X

MOF generalization is very similar to the one in UML. Inheritance in the MOF is implemented by the so-called relation “Generalizes” between “GeneralizableElements” (package or class). This relation allows the GeneralizableElements to inherit from other ones.

When dealing with associations, the MOF offers two mechanisms. The first one rests on associations. An association possesses two ends that define type and cardinality of the elements related by the association. With such elements, it is possible to reach all the classes related by this association.

The second way uses the concept of a “ reference ”. A reference is defined within a MOF class for an association. Using two links so-called “ RefersTo ” and “ Exposes ” the reference makes it possible to describe the two ends of an association and the attached cardinalities.

The concept of a MOF package

The package is an element involved in the MOF construction that allows grouping meta model elements in a single entity. Thanks to packages, it is possible to partition the model space.

A package can contain the following concepts:

- class
- association
- data type
- package

The concept of a MOF package supports four forms of construction: nesting, inheritance, composition, and importation.

Data type

The MOF offers the concept of Data type to define the values of the attributes and parameters of operations. Data type can be used to represent two kinds of types:

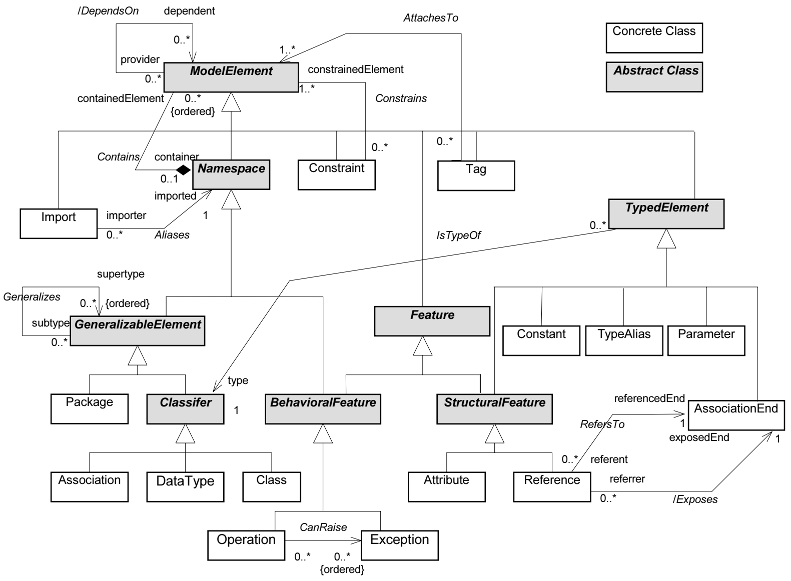
- scalar types which do not denote any object, and
- external types that are not defined by the MOF specifications.

It is to be taken note that neither instantiation, nor inheritance, nor composition apply to Data types.

A class may contain Data types. In fact, a class is a Namespace that can contain ModelElements; since Data type is a ModelElement it can be included in a class.

MOF GENERAL ARCHITECTURE

The general architecture of the MOF is summarised in the next diagram.


/DependsOn
dependent
0..*
provider
0..*
AttachesTo
1..*
ModelElement
constrainedElement
1..*
Constrains
containedElement
0..*
{ordered}
Concrete Class
Abstract Class
0..*
0..*
Contains
container
0..1
Namespace
Constraint
Tag
imported
Import
importer
1
0..*
Aliases
TypedElement
0..*
IsTypeOf
supertype
Generalizes
0..*
{ordered}
subtype
GeneralizableElement
0..*
Feature
Constant
TypeAlias
Parameter
type
Package
Classifer
1
BehavioralFeature
StructuralFeature
referencedEnd
AssociationEnd
RefersTo
1
exposedEnd
1
0..*
referent
Association
DataType
Class
Attribute
Reference
referrer
0..*
/Exposes
CanRaise
Operation
Exception
0..*
0..*
{ordered}

APPENDIX B

Overview of OCL

INTRODUCTION

This chapter presents the Object Constraint Language (OCL), a formal language used to refine UML specifications. OCL typically specify invariant conditions that must hold for the system being modeled. OCL is a pure expression language and its evaluations do not have side effects. In addition, to specifying invariants of the UML metamodel, UML modelers can use OCL to specify application-specific constraints in their models.

A UML diagram, such as a class diagram, is typically not refined enough to provide all the relevant aspects of a specification. There is a need to describe additional constraints about the objects in the model. Such constraints are often described in natural language that this will always result in ambiguities. In order to write unambiguous constraints, so-called formal languages have been developed

OCL can be used for a number of different purposes. It allows to specify invariants on classes and types in the class model, type invariant for stereotypes, and constraints on operations. It can be used as a navigation language and to describe pre- and post-conditions on operations, methods, and guards.

RELATION TO THE UML METAMODEL

Specifying the UML context

The context of an OCL expression within a UML model can be specified through a so-called context declaration at the beginning of an OCL expression.

```
context Person inv:
context Person::age():integer
...
```

If the constraint is shown in a diagram, with the proper stereotype and the dashed lines to connect it to its contextual element, there is no need for an explicit context declaration in the test of the constraint. The context declaration is optional as in figure 1.

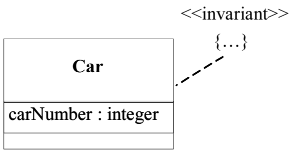


Figure 1: graphical representation of an invariant

Invariants, pre- and post-conditions

The OCL expression can be part of:

- An Invariant which is a Constraint stereotyped as an “invariant”. When the invariant is associated with a Classifier (for example a class), the latter is referred to as a “type”. An OCL expression is an invariant of the type and must be true for all instances of that type at any time. All OCL expressions representing invariants are of the type Boolean ;
- A pre-condition or post-condition, corresponding to «pre-condition» and «post-condition» stereotypes of Constraint associated with an Operation or Method. The context declaration in OCL uses the *context* keyword, followed by the type and operation declaration. The stereotype of constraint is shown by putting the labels ‘pre:’ and ‘post:’ before the actual pre-conditions and post-conditions.

The reserved word *result* denotes the result of the operation, if there is one. The names of the parameters (*param1*) can also be used in the OCL expression.

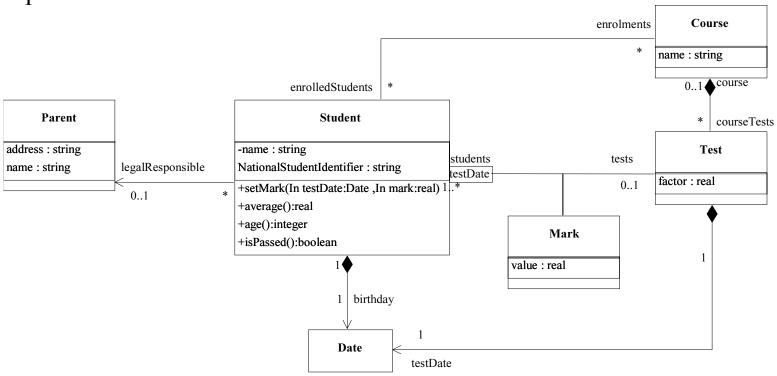


Figure 2: example management of the students tests

Optionally, the name of the pre-condition or post-condition may be written after the *pre* or *post* keyword, allowing the constraint to be referenced by name. In the example of diagram **2**, we can express the following pre- and post-condition:

```
context Student::setMark(testDate: Date,
                                 aMark: Real)
pre validMark : aMark>=0 and aMark<=20
post: self.mark[testDate].value=aMark
```

In a post-condition, the expression can refer to two sets of values for each property of an object:

- the value of a property at the start of the operation or method,
- the value of a property upon completion of the operation or method.

The value of a property in a post-condition is the value upon completion of the operation. To refer to the value of a property at the start of the operation, one has to postfix the property name with the keyword '*@pre*':

```
context Student::setMark(testDate: Date, aMark:
Real)
pre validMark : aMark>=0 and aMark<=20
post: self.mark[testDate].value=aMark and
    average()= average@pre()+
      (self.tests[testDate].factor*aMark)/
          self.tests.factor->sum()
```

In this example, when a new mark is added for a specific date, this mark as an impact on the average of the student. The post-condition expresses that the value of the mark for the tests of the *testDate* is equal to the parameter *aMark* and the average of the student is impacted by the adding of this new mark.

When the pre-value of a property evaluates to an object, all further properties that are accessed of this object are the new values (upon completion of the operation) of this object. So:

```
a.b@pre.c   -- takes the old value of property b
            -- of a, say x and then the new
            -- value of c of x.
a.b@pre.c@pre  -- takes the old value of
               -- property b of a, say x and
               -- then the old value of c of x.
```

The ‘@pre’ postfix is allowed only in OCL expressions that are part of a post-condition. Asking for a current property of an object that has been destroyed during execution of the operation results in Undefined. Also, referring to the previous value of an object that has been created during execution of the operation results in Undefined.

Self

An OCL expression can be associated with the context of an instance of a specific type. The reserved word *self* is used to refer to the contextual instance. For instance, if the context is *Student*, then *self* refers to an instance of *Student*. In the definition of an invariant, the keyword *self* can be dropped. An explicit name can be used instead of *self*. For example in figure **2**, consider that all student younger than eighteen must have a legal responsible. It is possible to express this constraint using *self* or an explicit name like *aStudent*:

```
context Student  inv legalResponsible:
self.age()<18 implies
self.legalResponsible->notEmpty()
```

or an explicit name like aStudent :

```
context aStudent: Student inv legalResponsible:
aStudent.age()<18 implies
  Student.legalResponsible->notEmpty();
```

In the definition of pre- or post-condition, *self* is an instance of the type which owns the operation or method as a feature. It is used in the expression referring to the object on which the operation was called.

Package context

When the context is ambiguous, it is possible to precise clearly the package containing the classifier where the invariant, pre- or post-condition is defined. To specify explicitly the package, these constraints can be enclosed between ’package’ and ’endpackage’ statements. The package statements have the syntax:

```
package Package::SubPackage
     context X inv:
     ... some invariant ...
     context X::operationName(..)
```

```
    pre: ... some pre-condition ...
endpackage
```

BASIC VALUES AND TYPES

In OCL, a number of basic types are predefined and available to the modeler at all time. These predefined value types are independent of any object model and part of the definition of OCL. The most basic value in OCL is a value of one of the basic types.

Types

Each OCL expression is written in the context of a UML model, a number of classifiers (types/classes, ...), their features and associations, and their generalizations. All classifiers from the UML model are types in the OCL expressions that are attached to the model.

Enumerations are datatypes in UML and have a name, just like any other Classifier. An enumeration defines a number of enumeration literals, that are the possible values of the enumeration. Within OCL one can refer to the value of an enumeration.

Let expressions and “definition” constraints

When a sub-expression is used more than once in a constraint, the *let* expression allows one to define an attribute or operation which can factorize the sub-expression. The scope of a let expression is limited to the specific constraint.

```
context Student inv:
let testsPassed: Set(Course) =
   self.tests.course->asSet() in
self.testsPassed->includesAll(self.enrolments)
```

A let expression may be included in an invariant or pre- or post-condition. To enable reuse of let variables/operations, one can use a constraint with the stereotype “definition”, in which let variables/operations are defined. This “definition” Constraint must be attached to a Classifier and may only contain let definitions.

All variables and operations defined in the "definition" constraint are known in the same context as where any property of the Classifier can be used. In essence, such variables and operations are pseudo-attributes and pseudo-operations of the classifier. They are used in an OCL expression in exactly the same way as attributes or operations are used. The textual notation for a "definition" Constraint uses the keyword *def* as shown below:

```
context Student def:
let testsPassed : Set(Course) =
self.tests.course->asSet()
```

The names of the attributes/operations in a let expression may not conflict with the names of respective attributes/associationEnds and operations of the Classifier. Also, the names of all let variables and operations connected with a Classifier must be unique.

Type conformance

OCL is a typed language and the basic value types are organized in a type hierarchy. An OCL expression in which all the types conform is a valid expression. An OCL expression in which the types don't conform is an invalid expression. The OCL expression contains a type *conformance error*. A type *type1* conforms to a type *type2* when an instance of *type1* can be substituted at each place where an instance of *type2* is expected. The type conformance rules for types in the class diagrams are simple:

- each type conforms to each of its supertypes,
- type conformance is transitive: if *type1* conforms to *type2*, and *type2* conforms to *type3*, then *type1* conforms to *type3*.

Precedence rules

The precedence order for the operations, starting with highest precedence, in OCL is:

1. @pre
2. dot and arrow operations: '.' and '->'
3. unary 'not' and unary minus '-'
4. '*' and '/'
5. '+' and binary '-'
6. 'if-then-else-endif'
7. '<', '>', '<=', '>='

8. ‘=’, ‘<>’
9. ‘and’, ‘or’ and ‘xor’
10. ‘implies’

Parentheses ‘(’ and ‘)’ can be used to change precedence.

Keywords

Keywords in OCL are reserved words. That means that the keywords cannot occur anywhere in an OCL expression as the name of a package, a type or a property. The list of keywords is shown below:

```
and    context  def    else    endif    endpackage
if     implies  in     inv     let      not
or     package  pre    post    then     xor
```

Comment

Comments in OCL are written following two successive dashes (minus signs). Everything immediately following the two dashes up to and including the end of line is part of the comment.

Undefined values

Whenever an OCL expression is being evaluated, there is a possibility that one or more of the queries in the expression are undefined. If this is the case, then the complete expression will be undefined. There are two exceptions to this for the Boolean operators:

- True OR-ed with anything is True,
- False AND-ed with anything is False.

The above two Boolean expressions are valid irrespective of the order of their arguments and whether or not the value of the other sub-expression is known.

OBJECTS AND PROPERTIES

OCL expressions can refer to Classifiers and datatypes. Also all attributes, association-ends, methods, and operations without side-effects that are defined on these types, etc. can be used. In a class model, an operation or method is defined to be side-effect-free if the *isQuery* attribute of the operations is true. For the purpose of this document, we will refer to attributes, association-ends, and side-effect-free methods and operations as being *properties*. A property is one of:

- an Attribute,
- an AssociationEnd,
- an Operation with *isQuery* being true,
- a Method with *isQuery* being true.

Properties

The value of a property on an object that is defined in a class diagram is specified by a dot followed by the name of the property.

```
context AType inv:
self.property
```

If *self* is a reference to an object, then *self.property* is the value of the *property* property on *self.*

For example, the *factor* of a *Test* is written as *self.factor*:

```
context Test inv:
self.factor > 0
```

The value of the subexpression *self.factor* is the value of the *factor* attribute on the particular instance of *Test* identified by *self.* The type of this subexpression is the type of the attribute *factor*, which is the basic type Real. Using attributes, and operations defined on the basic value types, we can express calculations. over the class model.

Operations may have parameters. For example, as shown earlier, a *Student* object has marks set using the function *setMark*. This operation would be accessed as follows, for a particular *Student aStudent,* the date of the test *12/12/2003* and a mark *12*:

```
aStudent.setMark(12/12/2003, 12)
```

The operation itself could be defined by a pre-condition constraint. This is a constraint that is stereotyped as «pre-condition»:

```
context Student::setMark (  testDate: Date,
                            aMark: Real)
pre: aMark>=0 and aMark<=20
```

The right-hand-side of this definition may refer to the operation being defined (i.e., the definition may be recursive) as long as the recursion is not infinite. To refer to an operation or a method that doesn't take a parameter, parentheses with an empty argument list are mandatory:

```
context Student inv:
self.age() > 0
```

Starting from a specific object, we can navigate an association on the class diagram to refer to other objects and their properties. To do so, we navigate the association by using the opposite association-end:

```
object.rolename
```

The value of this expression is the set of objects on the other side of the *rolename* association. If the multiplicity of the association-end has a maximum of one ("0..1" or "1"), then the value of this expression is an object. In the example of the class diagram of figure **2**, when we start in the context of a *Student*, we can write:

```
context Student
inv: self.legalResponsible.address = "Toulouse"
inv: self.enrolments->notEmpty()
```

In the first invariant, *self.legalResponsible* is a *Parent*, because the multiplicity of the association is zero or one. In the second invariant, *self.enrolments* calculates in a Set of *Course*. By default, navigation will result in a Set. When the association on the Class Diagram is adorned with {ordered}, the navigation results in a Sequence.

Missing rolenames

When a rolename is missing at one of the ends of an association, the name of the type at the association end, starting with a lowercase character, is used as the rolename. If this results in an ambiguity, the rolename is mandatory. This is the case with unnamed rolenames in reflexive associations. If the rolename is ambiguous, then it cannot be used in OCL.

Navigation over associations with multiplicity zero or one

Because the multiplicity of the role manager is one, *self.legalResponsible* is an object of type *Parent*. Such a single object can be used as a Set as well. It then behaves as if it is a Set containing the single object. The usage as a set is done through the arrow followed by a property of Set. This is shown in the following example:

```
context Student inv:
self.legalResponsible ->size() = 1
```

Combining properties

Properties can be combined to make more complicated expressions. An important rule is that an OCL expression always evaluates to a specific object of a specific type. After obtaining a result, one can always apply another property to the result to get a new result value. Therefore, each OCL expression can be read and evaluated left-to-right.

Association classes and navigation

To specify navigation to association classes (*Mark* in the example), OCL uses a dot and the name of the association class starting with a lowercase character:

```
context Student inv:
self.mark
```

The sub-expression *self.mark* evaluates to a Set of all the marks of a student has with the tests that he takes. In the case of an association class, there is no explicit rolename in the class diagram. The name *mark* used in this navigation is the name of the association class starting with a lowercase character, similar to the way described in the section "Missing rolenames" above. In case of a recursive association, that is an association of a class with itself, the name of the association class alone is not enough. We need to distinguish the direction in which the association is navigated as well as the name of the association class.

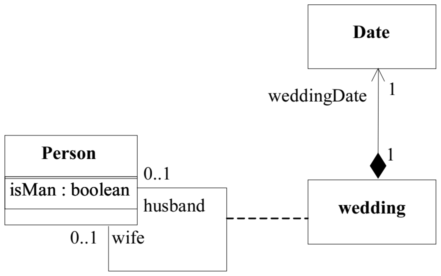


Figure 3 : example of recursive association

In the figure 3, when navigating to an association class such as *wedding* there are two possibilities depending on the direction. We may navigate towards the *wife* end, or the *husband* end. By using the name of the association class alone, these two options cannot be distinguished. To make the distinction, the rolename of the direction in which we want to navigate is added to the association class name, enclosed in square brackets. In the expression:

```
context Person inv:
let theWife : Person=self.wedding[wife]
let theHusband : Person=self.wedding[husband]
in
  self.isMan and self.theWife->notEmpty()and
  self.theHusband->isEmpty() implies
    theWife.wedding[husband]=self
or
  not self.isMan and self.theHusband->
    notEmpty()and self.theWife->isEmpty()
     implies theHusband.wedding[wife]=self
or
  self.theWife->isEmpty()and
  self.theHusband->isEmpty()
```

the *self.wedding[wife]* evaluates to the wife belonging to the association-end of *wife*. And the *self.wedding [husband]* evaluates to the husband of *wedding* belonging to the association-end of *husband*. The unqualified use of the association class name is not allowed in such a recursive situation. Thus, the following example is invalid:

```
context Person inv:
self.wedding->isEmpty()
```

In a non-recursive situation, the association class name alone is enough, although the qualified version is allowed as well. Therefore, the examples of the figure 2 could also be written as:

```
context Student inv:
self.mark[tests]
```

We can navigate from the association class itself to the objects that participate in the association. This is done using the dot-notation and the role-names at the association-ends.

```
context Mark
inv: self.tests.factor …
inv: self.student.age() …
```

Navigation from an association class to one of the objects on the association will always deliver exactly one object. This is a result of the definition of AssociationClass. Therefore, the result of this navigation is exactly one object, although it can be used as a Set using the arrow (->).

Navigation through qualified associations

Qualified associations use one or more qualifier attributes to select the objects at the other end of the association. To navigate them, we can add the values for the qualifiers to the navigation. This is done using square brackets, following the role-name. It is permissible to leave out the qualifier values, in which case the result will be all objects at the other end of the association. The result of the evaluation of the following expression is a set of persons containing all tests of the student

```
context Student inv:
self.tests…
```

The result of the evaluation of the following expression is the test took by the student the 12 December 2003.

```
context Student inv:
self.tests[12/12/2003]
```

If there is more than one qualifier attribute, the values are separated by commas, in the order which is specified in the UML class model. It is not permissible to partially specify the qualifier attribute values.

Using pathnames for packages

Within UML, different types are organized in packages. OCL provides a way of explicitly referring to types in other packages by using a package-pathname prefix. The syntax is a package name, followed by a double colon:

```
Packagename::Typename
```

This usage of pathnames is transitive and can also be used for packages within packages:

```
Packagename1::Packagename2::Typename
```

Accessing overridden properties of supertypes

Whenever properties are redefined within a type, the property of the supertypes can be accessed using the *oclAsType()* operation. Whenever we have a class B as a subtype of class A, and a property p1 of both A and B, we can write:

```
context B inv:
self.oclAsType(A).p1  -- accesses the p1
                      -- property defined in A
self.p1    -- accesses the p1 property defined
           -- in B
```

Features on classes themselves

The types are either predefined in OCL or defined in the class model. In OCL, it is also possible to use features defined on the types/classes themselves. These are, for example, the *class*-scoped features defined in the class model. Furthermore, several features are predefined on each type. A predefined feature on each type is *allInstances*, which results in the Set of all instances of the type in existence at the specific time when the expression is evaluated. If we want to make sure that all instances of *Person* have unique national student identifier, we can write:

```
context Student inv:
Student.allInstances->isUnique
    (s:Student|s.nationalStudentIdentifier)
```

The *Student.allInstances* is the set of all students and is of *type Set(Student)*. It is the set of all students that exist at the snapshot in time that the expression is evaluated.

Collections

Single navigation results in a Set, combined navigations in a Bag, and navigation over associations adorned with {ordered} results in a Sequence.

The type Collection is predefined in OCL. The Collection type defines a large number of predefined operations to enable the OCL expression author (the modeler) to manipulate collections. Consistent with the definition of OCL as an expression language, collection operations never change collections; *isQuery* is always true. They may result in a collection, but rather than changing the original collection they project the result into a new one. Collection is an abstract type, with the concrete collection types as its subtypes. OCL distinguishes three different collection types: Set, Sequence, and Bag. A Set is the mathematical set. It does not contain duplicate elements. A Bag is like a set, which may contain duplicates (i.e., the same element may be in a bag twice or more). A Sequence is like a Bag in which the elements are ordered. Both Bags and Sets have no order defined on them. Sets, Sequences, and Bags can be specified by a literal in OCL. Curly brackets surround the elements of the collection, elements in the collection are written within, separated by commas. The type of the collection is written before the curly brackets:

```
Set { ‘a’, ‘b’, ‘c’, ‘d’ }
Sequence { 1, 2, 3, 2, 3 }
Bag {‘a’, ‘b’, ‘a’, ‘b’ }
```

Because of the usefulness of a Sequence of consecutive Integers, there is a separate literal to create them. The elements inside the curly brackets can be replaced by an interval specification, which consists of two expressions of type Integer, *Int-expr1* and *Int-expr2*, separated by ‘..’. This denotes all the Integers between the values of *Int-expr1* and *Int-expr2*, including the values of *Int-expr1* and *Int-expr2* themselves:

```
Sequence{ 1..(6 + 4) }
Sequence{ 1..10 }
-- are both identical to
Sequence{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
```

Within OCL, all Collections of Collections are flattened automatically; therefore, the following two expressions have the same value:

```
Set{Set{‘a’, ‘b’}, Set{‘c’, ‘d’}, Set{‘e’,
‘f’}}
Set{‘a’, ‘b’, ‘c’, ‘d’, ’e’, ‘f’}
```

Collection type hierarchy and type conformance rules

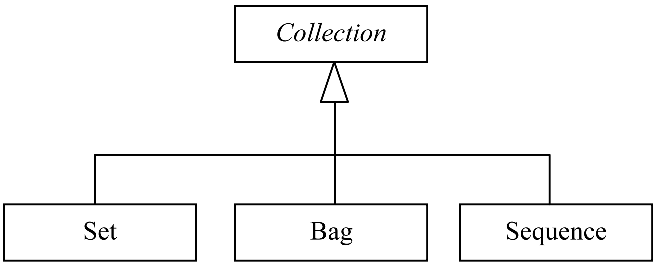


Figure 4: collection hierarchy

Type conformance rules are as follows for the collection types:

- *Type1* conforms to *Type2* when they are identical (standard rule for all types).
- *Type1* conforms to *Type2* when it is a subtype of *Type2* (standard rule for all types).
- Collection(Type1) conforms to Collection(Type2), when Type1 conforms to Type2.
- Type conformance is transitive: if *Type1* conforms to *Type2*, and *Type2* conforms to *Type3*, then *Type1* conforms to *Type3* (standard rule for all types).

PREDEFINED OCL TYPES

This section contains all standard types defined within OCL, including all the properties defined on those types. Its signature and a description of its semantics define each property. Within the description, the reserved word 'result' is used to refer to the value that results from evaluating the property. In several places, post-conditions are used to describe properties of the result. When there is more than one post-condition, all post-conditions must be true. The basic types used are Integer, Real, String, and Boolean. They are supplemented with OclExpression, OclType, and OclAny.

OclType

All types defined in a UML model, or pre-defined within OCL, have a type. This type is an instance of the OCL type called *OclType*. Access to this type allows the modeler limited access to the meta-level of the model. This can be useful for advanced modelers. Properties of *OclType*, where the instance of *OclType* is called t*ype*.

Property	Description
type.name() : String	The name of *type*
type.attributes() : Set(String)	The set of names of the attributes of *type*, as they are defined in the model
type.associationEnds() : Set(String)	The set of names of the navigable associationEnds of *type*, as they are defined in the model.
type.operations() : Set(String)	The set of names of the operations of *type*, as they are defined in the model
type.supertypes() : Set(OclType)	The set of all direct supertypes of *type*.

OclAny

Within the OCL context, the type *OclAny* is the supertype of all types in the model and the basic predefined OCL type.

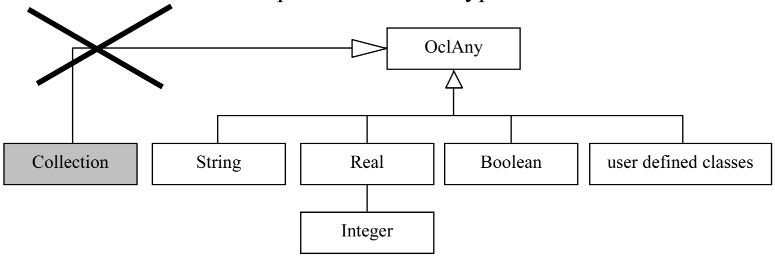


Figure 5: type hierarchy

The predefined OCL Collection types are not subtypes of *OclAny*. Properties of *OclAny* are available on each object in all OCL expressions. All classes in a UML model inherit all properties defined on OclAny. One can also use the *oclAsType()* operation to explicitly refer to the *OclAny* properties.

Property	Description
type.allSupertypes() : Set(OclType)	The transitive closure of the set of all supertypes of *type*

type.allInstances() : Set(type)	The set of all instances of *type* and all its subtypes in existence at the snapshot at the time that the expression is evaluated
object = (object2 : OclAny) : Boolean	True if *object* is the same object as *object2*
object <> (object2 : OclAny) : Boolean	True if *object* is a different object from *object2*
object.oclIsKindOf(type : OclType) : Boolean	True if *type* is **one**[4] **of** the types of *object*, or one of the supertypes (transitive) of the types of *object*.
object.oclIsTypeOf(type : OclType) : Boolean	True if *type* is equal to **one of** the types of *object*

OclState

The type *OclState* is used as a parameter for the operation *oclInState*. There are no properties defined on *OclState*. One can only specify an OclState by using the name of the state, as it appears in a statechart. These names can be fully qualified by the nested states and statechart that contain them. The operation *oclInState(s)* results in true if the object is in the state *s*. Values for *s* are the names of the states in the statechart(s) attached to the Classifier of *object*. For nested states the statenames can be combined using the double colon '::' . In the example of statechart diagram of figure 6, values for *s* can be *closed*, *opened*, *inProcess*, *inProcess::opening*, *inProcess::closing*.

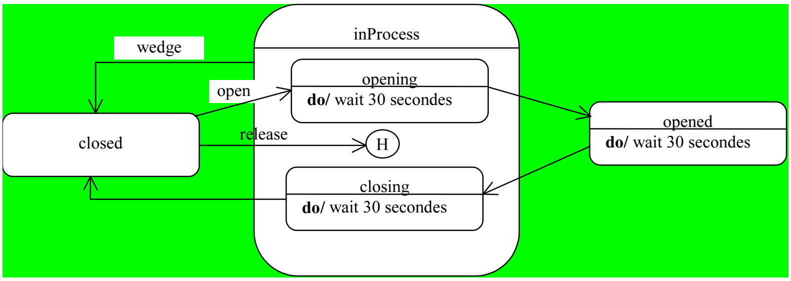


Figure 6: example of statechart

If the classifier of *object* has the above associated statechart, valid OCL expressions are:

[4] In UML, an object can have simultaneously several types. This is a UML specificity.

```
object.oclInState(closed)
object.oclInstate(inProcess::opening)
```

OclExpression

Each OCL expression itself is an object in the context of OCL. The type of the expression is *OclExpression*. An *OclExpression* includes the optional iterator variable and type and the optional accumulator variable and type. For any operation name *op*, the syntax options are:

```
collection->op( iter : Type | OclExpression )
collection->op( iter | OclExpression )
collection->op( OclExpression )
```

Property	Description
object.oclAsType(type : OclType) : type	Results in *object*, but of known type *type*. Results in Undefined if the actual type of *object* is not type or one of its subtypes
object.oclInState(state : OclState) : Boolean	Results in true if *object* is in the state *state*, otherwise results in false. The argument is a name of a state in the state machine corresponding with the class of *object*.
object.oclIsNew() : Boolean	Can only be used in a post-condition. Evaluates to true if the *object* is created during performing the operation. I.e. it didn't exist at pre-condition time.
expression.evaluationType() : OclType	The type of the object that results from evaluating *expression*.

For example in figure 2, the method *isPassed* return true if all marks of the student are greater or equal to 8 and the average of all these marks is greater than 10.

```
context Student::isPassed()
post: result=self.average()>10 and self.mark->
         forAll(m : Mark| m.value>8)

    context Student::setMark(testDate: Date,
                                 aMark: Real)
    pre validMark : aMark>=0 and aMark<=20
    post: c.oclIsNew() and c.oclIsTypeOf(Mark)
          and c.value=aMark and  self.mark=
          self.mark@pre->including(c)
```

Real

The OCL type Real represents the mathematical concept of real.

Property	Description
r = (r2 : Real) : Boolean	True if *r* is equal to *r2*
r <> (r2 : Real) : Boolean	True if *r* is not equal to *r2*
r + (r2 : Real) : Real	The value of the addition of *r* and *r2*
r - (r2 : Real) : Real	The value of the subtraction of *r2* from *r*
r * (r2 : Real) : Real	The value of the multiplication of *r* and *r2*
- r : Real	The negative value of *r*
r / (r2 : Real) : Real	The value of *r* divided by *r2*
r.abs() : Real	The absolute value of *r*
r.floor() : Integer	The largest integer which is less than or equal to *r*
r.round() : Integer	The integer which is closest to *r*. When there are two such integers, the largest one
r.max(r2 : Real) : Real	The maximum of *r* and *r2*
r.min(r2 : Real) : Real	The minimum of *r* and *r2*
r < (r2 : Real) : Boolean	True if *r1* is less than *r2*
r > (r2 : Real) : Boolean	True if *r1* is greater than *r2*
r <= (r2 : Real) : Boolean	True if r1 is less than or equal to r2
r >= (r2 : Real) : Boolean	True if r1 is greater than or equal to r2

Integer

The OCL type Integer represents the mathematical concept of integer.

Property	Description
i = (i2 : Integer) : Boolean	True if *i* is equal to *i2*
- i : Integer	The negative value of *i*
i + (i2 : Integer) : Integer	The value of the addition of *i* and *i2*
i - (i2 : Integer) : Integer	The value of the subtraction of *i2* from *i*
i * (i2 : Integer) : Integer	The value of the multiplication of *i* and *i2*
i / (i2 : Integer) : Real	The value of *i* divided by *i2*
i.abs() : Integer	The absolute value of *i*
i.div(i2 : Integer) : Integer	The number of times that *i2* fits completely within *i*
i.mod(i2 : Integer) : Integer	The result is *i* modulo *i2*
i.max(i2 : Integer) : Integer	The maximum of *i* an *i2*
i.min(i2 : Integer) : Integer	The minimum of *i* an *i2*

String

The OCL type String represents ASCII strings.

Property	Description
string = (string2 : String) : Boolean	True if *string* and *string2* contain the same characters, in the same order
string.size() : Integer	The number of characters in *string*
string.concat(string2 : String) : String	The concatenation of *string* and *string2*
string.toUpper() : String	The value of *string* with all lowercase characters converted to uppercase characters
string.toLower() : String	The value of *string* with all uppercase characters converted to lowercase characters
string.substring(lower : Integer, upper : Integer) : String	The sub-string of *string* starting at character number *lower*, up to and including character number *upper*

Boolean

The OCL type Boolean represents the common true/false values.

Property	Description
b = (b2 : Boolean) : Boolean	Equal if *b* is the same as *b2*
b or (b2 : Boolean) : Boolean	True if either *b* or *b2* is true.
b xor (b2 : Boolean) : Boolean	True if either *b* or *b2* is true, but not both.
b and (b2 : Boolean) : Boolean	True if both *b1* and *b2* are true.
not b : Boolean	True if *b* is false.
b implies (b2 : Boolean) : Boolean	True if *b* is false, or if *b* is true and *b2* is true.
if b then (expression1 : OclExpression) else (expression2 : OclExpression) endif : expression1.evaluationType()	If *b* is true, the result is the value of evaluating *expression1*; otherwise, result is the value of evaluating *expression2.*

Enumeration

The OCL type Enumeration represents the enumeration defined in an UML model.

Property	Description
enumeration = (enumeration2 : Boolean) : Boolean	Equal if *enumeration* is the same as *enumeration2*.
enumeration <> (enumeration2 : Boolean) : Boolean	Equal if *enumeration* is not the same as *enumeration2*.

Collection-related types

Each collection type is actually a template with one parameter. 'T' denotes the parameter. A real collection type is created by substituting a type for the T. So Set (Integer) and Bag (Person) are collection types. All collection operations with an *OclExpression* as parameter can have an iterator declarator.

Collection

Collection is the abstract supertype of all collection types in OCL. Each occurrence of an object in a collection is called an element. If an object occurs twice in a collection, there are two elements. This section defines the properties on Collections that have identical semantics for all collection subtypes. Some properties may be defined with the subtype as well, which means that there is an additional post-condition or a more specialized return value.

Property	Description
collection->size() : Integer	The number of elements in the collection *collection*
collection->includes(object : OclAny) : Boolean	True if *object* is an element of *collection*, false otherwise
collection->excludes(object : OclAny) : Boolean	True if *object* is not an element of *collection*, false otherwise
collection->count(object : OclAny) : Integer	The number of times that *object* occurs in the collection *collection*
collection->includesAll(c2 : Collection(T)) : Boolean	Does *collection* contain all the elements of *c2*?
collection->excludesAll(c2 : Collection(T)) : Boolean	Does *collection* contain none of the elements of *c2*?
collection->isEmpty() : Boolean	Is *collection* the empty collection?

collection->notEmpty() : Boolean	Is *collection* not the empty collection?
collection->sum() : T	The addition of all elements in *collection*. Elements must be of a type supporting the + operation. The + operation must take one parameter of type T and be both associative: (a+b)+c = a+(b+c), and commutative: a+b = b+a. Integer and Real fulfill this condition.
collection->exists(expr : OclExpression) : Boolean	Results in true if *expr* evaluates to true for at least one element in *collection*
collection->forAll(expr : OclExpression) : Boolean	Results in true if *expr* evaluates to true for each element in *collection*; otherwise, result is false
collection->isUnique(expr : OclExpression) : Boolean	Results in true if *expr* evaluates to a different value for each element in *collection*; otherwise, result is false
collection->sortedBy(expr : OclExpression) : Sequence(T)	Results in the Sequence containing all elements of *collection*. The element for which *expr* has the lowest value comes first, and so on. The type of the *expr* expression must have the < operation defined. The < operation must return a Boolean value and must be transitive i.e. if a < b and b < c then a < c
collection->iterate(expr : OclExpression) : expr.evaluationType()	The *iterate* operation is slightly more complicated, but is very generic. An accumulation builds one value by iterating over a collection. *collection->iterate(elem : Type; acc : Type = <expression>|expression-with-elem-and-acc).* The variable *elem* is the iterator, as in the definition of *select, forAll,* etc. The variable *acc* is the accumulator. The accumulator gets an initial value *<expression>*. When the iterate is evaluated, *elem* iterates over the *collection* and the *expression-with-elemand-acc* is evaluated for each *elem*. After each evaluation of *expression-with-elem-and-acc*, its value is assigned to *acc*. In this way, the value of *acc* is built up during the iteration of the collection.

collection->any(expr : OclExpression) : T	Returns any element in the *collection* for which *expr* evaluates to true. If there is more than one element for which *expr* is true, one of them is returned. The pre-condition states that there must be at least one element fulfilling *expr*, otherewise the result of this operation is Undefined
collection->one(expr : OclExpression) : Boolean	Results in true if there is exactly one element in the *collection* for which *expr* is true
collection->select(expr : OclExpression) : collection(T)	Results in a collection that contains all the elements from *collection* for which the *expr* evaluates to true.
collection->reject(expr : OclExpression) : collection(T)	Results in a collection that contains all the elements from *collection* for which the *expr* evaluates to false
collection->collect(expr : OclExpression) : Bag(expr. evaluationType())	Results in a Bag that contains the result of the evaluation of *expr.* Because navigation through many objects is very common, there is a shorthand notation for the collect that makes the OCL expressions more readable. In figure 2, instead of *self.legalResponsible-> collect(address)* we can also write *self.legalResponsible.address*

Set

The Set is the mathematical set. It contains elements without duplicates

Property	Description
set->union(set2 : Set(T)) : Set(T)	The union of *set* and *set2*
set->union(bag : Bag(T)) : Bag(T)	The union of *set* and *bag*
set = (set2 : Set(T)) : Boolean	Evaluates to true if *set* and *set2* contain the same elements
set->intersection(set2 : Set(T)) : Set(T)	The intersection of *set* and *set2* (i.e, the set of all elements that are in both *set* and *set2*)
set->intersection(bag : Bag(T)) : Set(T)	The intersection of *set* and *bag*
set – (set2 : Set(T)) : Set(T)	The elements of *set*, which are not in *set2*
set->including(object : T) : Set(T)	The set containing all elements of *set* plus *object*
set->excluding(object : T) : Set(T)	The set containing all elements of *set* without *object*

set->symmetricDifference(set2 : Set(T)) : Set(T)	The sets containing all the elements that are in *set* or *set2*, but not in both
set->select(expr : OclExpression) : Set(T)	The subset of *set* for which *expr* is true
set->reject(expr : OclExpression) : Set(T)	The subset of *set* for which *expr* is false
set->collect(expr : OclExpression) : Bag(expr.evaluationType())	The Bag of elements which results from applying *expr* to every member of *set*
set->count(object : T) : Integer	The number of occurrences of *object* in *set*
set->asSequence() : Sequence(T)	A Sequence that contains all the elements from *set,* in undefined order
set->asBag() : Bag(T)	The Bag that contains all the elements from *set*

Bag

A bag is a collection with duplicates allowed. That is, one object can be an element of a bag many times.

Property	**Description**
bag = (bag2 : Bag(T)) : Boolean	True if *bag* and *bag2* contain the same elements, the same number of times
bag->union(bag2 : Bag(T)) : Bag(T)	The union of *bag* and *bag2*
bag->union(set : Set(T)) : Bag(T)	The union of *bag* and *set*
bag->intersection(bag2 : Bag(T)) : Bag(T)	The intersection of *bag* and *bag2*
bag->intersection(set : Set(T)) : Set(T)	The intersection of *bag* and *set*
bag->including(object : T) : Bag(T)	The bag containing all elements of *bag* plus *object*
bag->excluding(object : T) : Bag(T)	The bag containing all elements of bag apart from all occurrences of *object*
bag->select(expr : OclExpression) : Bag(T)	The sub-bag of *bag* for which *expr* is true
bag->reject(expr : OclExpression) : Bag(T)	The sub-bag of *bag* for which *expr* is false
bag->collect(expr: OclExpression) : Bag(expr.evaluationType())	The Bag of elements which results from applying *expr* to every member of *bag*

bag->count(object : T) : Integer	The number of occurrences of *object* in *bag*
bag->asSequence() : Sequence(T)	A Sequence that contains all the elements from *bag*, in undefined order
bag->asSet() : Set(T)	The Set containing all the elements from *bag*, with duplicates removed

Sequence

A sequence is a collection where the elements are ordered.

Property	Description
sequence->count(object : T) : Integer	The number of occurrences of *object* in *sequence*
sequence = (sequence2 : Sequence(T)) : Boolean	True if *sequence* contains the same elements as *sequence2* in the same order
sequence->union (sequence2 : Sequence(T)) : Sequence(T)	The sequence consisting of all elements in *sequence*, followed by all elements in *sequence2*
sequence->append (object: T) : Sequence(T)	The sequence of elements, consisting of all elements of *sequence*, followed by *object*
sequence->prepend(object : T) : Sequence(T)	The sequence consisting of *object*, followed by all elements in *sequence*
sequence->subSequence(lower : Integer, upper : Integer) : Sequence(T)	The sub-sequence of *sequence* starting at number *lower*, up to and including element number *upper*
sequence->first() : T	The first element in *sequence*
sequence->at(i : Integer) : T	The *i-th* element of sequence
sequence->including(object : T) : Sequence(T)	The sequence containing all elements of *sequence* plus *object* added as the last element
sequence->excluding(object : T) : Sequence(T)	The sequence containing all elements of *sequence* apart from all occurrences of *object*. The order of the remaining elements is not changed
sequence->select(expression : OclExpression) : Sequence(T)	The subsequence of *sequence* for which *expression* is *true*
sequence->reject(expression : OclExpression) : Sequence(T)	The subsequence of *sequence* for which *expression* is false
sequence->collect(expression : OclExpression) : Sequence (expression.evaluationType())	The Sequence of elements which results from applying *expression* to every member of *sequence*

sequence->iterate(expr : OclExpression) : expr.evaluationType()	Iterates over the sequence. Iteration will be done from element at position 1 up until the element at the last position following the order of the sequence

APPENDIX C

Overview of XML and XMI

THE CHOICE OF XML

Information processing systems are more and more and more build from distributed software components which need to exchange information over networks. It is the case of the electronic mail which makes it possible for the user to build a message which will be sent to the recipient site and managed by a software having various suitable functions (display, storage, answer, ...). Component software can also need to communicate between them without being distant. For example, a user can be brought to extract information from a database and communicate it to a software specialised in statistical processing. These examples show that there is a significant need for communication between heterogeneous systems.

The XML standard was developed to define a format for information exchange independent of any vendor solution.

In the same way, UML related tools (model editors, model checkers, code generators, documentation generators) must manage UML models. The use of heterogeneous, vendor independent, tools can be made possible if some standard way of storing UML models is used. This is the purpose of the XMI format which is based both on XML and on the definition of a MOF-based metamodel describing the structure of the information to be stored. This is also why Neptune takes as input models stored in the XMI format.

XML PRINCIPLES

To facilitate the exchange of information between two applications, a solution consists in conveying with each data, the set of information which describes it - which is generally called meta-data.

An XML document consists of meta-data associated to some value. Meta-data can be either elementary and valued by a string or composite and in turn described by valued meta-data. In both cases, meta-data are associated to additional properties which qualify the data.

For example, if a user wants to send the contents of the message: "Hello, how are you?", the mail client will send the contents of the message with its metadata, which could be: sender, receiver, Message-content, text, police, size.

```
Sender:
            Name="dupont"
            FirstName="Jean"
Message-Content:
text="Hello, how are you?"
police="time"
size="12"
Receiver:
            Name="durand"
            FirstName="Paul"
            e-mail address="durand-paul@… .fr"
```

XML elements

Thus in a first approach one can define an XML document in the following way:

```
XML_document::= <name > (elementary_information
| XML_document*) </name>
```

XML attributes

XML defined another container of information called "Attribute" which cannot exist alone but must be attached and included in an XML element, empty or not. The difference of use of elements or attributes is not clearly established because each of us can contain the same information. The w3c norm recommends to use attributes to define a set of attribute to an element, to establish type constraint for the attribute, or to provide default values. In practice, attributes are used to qualify the nature of the element like in versioning, to contain a value specified by a pre-defined list, to precise that the information is optional or mandatory or to contain references towards others documents or elements. But attributes are very useful to reduce the length of XML documents allowing to give the value directly to the attribute, avoiding the use of an opening and a closing tag as for an element.

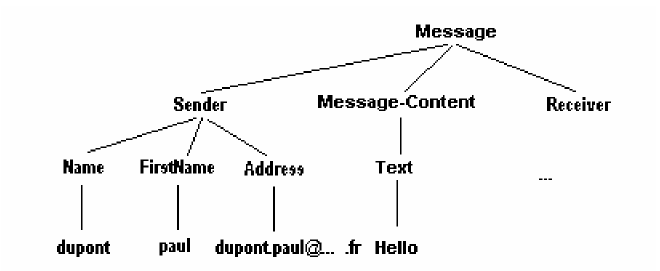
We can thus complete the definition of the syntax of an XML document with attributes qualifying the nodes of XML trees:

```
XML_document::= <name qualifier*>
    (elementary_information | XML_document*)
     </name>
qualifier::= attribute=value
```

REPRESENTATION OF AN XML DOCUMENT

The syntactic tree

The definition of an XML document confers to it a tree-like structure, where each leaf corresponds to elementary information and where each node defines the metadata associated to the sub-tree. As an example, an electronic message could be represented according to the following

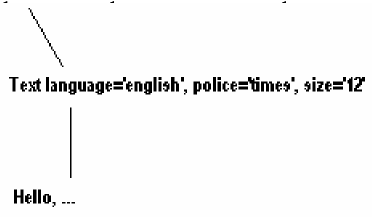


structure:

Each leaf representing an elementary information can be either a text or an URL given access to an unspecified document (image, …).

Non-structuring metadata

Each metadata can have qualifiers also named attributes. Those qualifiers, which are a ure information or to bring a structur add additional information to nodes o



A classical top-down and left to right traversal of the tree can be performed to univocally transform it into a marked character string. Each sub-tree is parenthesised by opening and closing markers encapsulating the metadata. This is the way XML documents are represented and exchanged by components.

```
<Message-Content>
      <Sender>
            <Name>dupont</Name>
            …
      </Sender>
      …
</Message-Content>
```

One can notice that each element (simple or composite) is delimited by an opening and a closing marker. Then attributes, defined by couples (meta-data, quoted-value), can be attached to opening markers as follows:

```
<Text language=’english’ police=’times’size=’12’>
      Hello, …
</Text>
```

ID and IDREF attributes

ID and IDREF are very useful attributes respectively to identify/refer an XML element inside a document. Expression of both attributes are based on XML Names. While ID allows to construct a direct access table for the identified elements, this table is made during the parsing-validation phase, see further, IDREF give the capability to access an element without searching it in the ancestor tree. This mechanism speeds a lot any XSL process.

The ID attribute must store an unique value although the IDREF may be expressed as distinct elements or as a list of reference using IDREFS type.

Example: “Construct your Toc”

```
  <?xml version='1.0' encoding='ISO-8859-1'?>
    <!DOCTYPE Book SYSTEM 'book.dtd'>
      <Book>
        <Chapter idElt=’Chap1’
```

```
                 Title='FirstChapterTitle'
                 startpage='10'>
            <Section idElt='Section1'/>
        </Chapter>
        <Chapter idElt ='Chap2'
                 Title='SecondChapterTitle'
                 startpage ='30'>
            <Section idElt='Section2'/>
         </Chapter>
         <Toc>
            <!--First expression using IDREF-->
              <TocEntry chap.idref='1'/>
              <TocEntry chap.idref='2'/>
            <!--Second expression using IDREFS-->
              <TocEntry chap.idref='1 2'/>
          </Toc>
      </Book>
```

WELL-FORMED XML DOCUMENTS

A marked document is considered as an XML document if it is well formed, i.e. well parenthesised: each opening marker must be associated to a matching closing marker. This document must have only one entry-point called its root. Thus, a minimal document contains at least the root marker: `<rootMark></rootMark>`. A more compact notation can be used if a node does not have sub-trees: `<rootMark/>`. Such a marker is both opening and closing.

Document Type Definition (DTD)

Well formed documents can be exchanged between components, but components understand their contents if the sender and the receiver agree on the meta-data used to represent information, and on the way meta-data are structured. Meta-data define the mark-ups used in the XML document. The set of markups and their organisation are described in the Document Type Definition (DTD). Thus, an XML document can be understood by the receiver if the sender and the receiver both agree on its DTD, i.e. on the vocabulary and on the syntax used to exchange XML-based information. Furthermore, in order to make communication between applications easier, their exists predefined DTD for some domains. For example, models of document (DTDs) are defined for banking or commercial data exchange.

Coming back to the previous message exchange example, the DTD of messages can be the following:

```
<!Element Message-Content (Sender, text?,
                                  Receiver+)>
<!Element Sender(Name,First-Name,Adress)
<!Element Name #PCDATA>
<!Element First-Name #PCDATA>
<!Element Address #PCDATA>
<!Element text #PCDATA>
<!Attlist text
          language CDATA #REQUIRED "english"
          police CDATA #IMPLIED ("times" |
              "helvetica") "times"
          size CDATA FIXED "12">
      …
```

The content of Message-Content specifies that Sender is the first element followed by text and finally followed by Receiver. In this declaration text is optional (0..1), Sender is unique, and one Receiver is mandatory but several are authorised. #PCDATA indicates that the content of the element is XML string type relevant.

While Attlist declaration specifies a default value “english” for language attribute which is required in the definition of text, the precision of the police can be forgotten or its value can be chosen in a list of given possibilities with a default value “times’. At least the size of the police is fixed at the value “12” and never change.

Concerning our study, there is a standardised DTD whose instance in an XML form encode an UML model.

Valid documents

Valid documents are well-formed documents obeying to the structure given in a DTD. The checking of validity can be made using dedicated tools, called parsers, whose goal is to verify that an XML document can be considered as an instance of the DTD. The validity check imposes the parser to analyse first the DTD and verify the presence of all required information respecting order containment, types, links towards existent internal or external referred information. Validity of documents ensure that publication of a document is possible conforming to a standardised way to write.

THE XML METADATA INTERCHANGE FORMAT (XMI)

As explained before, XML documents can be validated according to a DTD thus ensuring that they are both well-formed and that the XML document is an instance of the DTD. A DTD specifies the organisation of meta-data. The same information can in fact be described using an UML class diagram where classes represent meta-data, the information metamodel. The purpose of XMI (XML Metadata Interchange) is to link these two ways of representing the structure of meta-data. More precisely, given an UML model of meta-data, the XMI norm, proposed by the OMG, standardises through a set of rules the way to produce a DTD and to encode information structured according to this meta-data in an XML file that complies with the DTD. In fact, XMI defines how to produce a DTD and an XML instance from a MOF representation of the metadata, MOF defining an interface to metamodel repositories that can be represented by an UML class diagram. Consequently, UML CASE tools are used to edit metamodels and tools exist (the XMIFramework of IBM for example) to transform metamodels specified in UML into DTD.

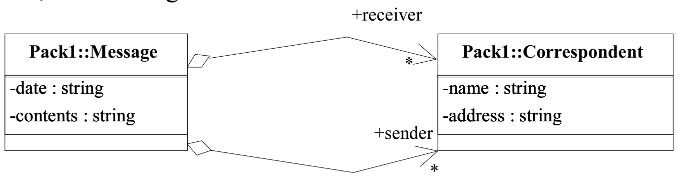
The XMI norm can be used to encode any information whose structure can be specified with an UML class diagram, and thus to serialise instances of UML class diagrams. As the abstract syntax of UML itself is specified by an UML class diagram -- the UML metamodel -- UML models can be encoded in XMI and exchanged. This is why most UML CASE tools propose an XMI import and export facility which should allow the communication of UML models between CASE tools, model checkers, code generators, document generators...

Example of metamodel and its associated DTD

An XML document must be validated according to a DTD (or an XML schema) to ensure that it is both well-formed and a correct instance of the DTD. In this framework, XMI brings an important help: it permits an automatic generation of a DTD from any UML class diagram. In fact, XMI is a bridge from the UML-based technologies and the XML world which increases the popularity of the two domains.

DTDs are needed to check that XML documents are valid, i.e. that they conform to a model of document. This model of document can be written directly in the DTD language, but reading of some DTDs informed us about the painfulness of this language. Another way to do is to model the considered class of documents with an UML class diagram, thus using an easily understandable graphical notation. Then producing a DTD becomes automatic and makes UML an important technology for communities which exchange structured data in XML.

As an example, the following UML class diagram specifies the structure of a message: it has a date, a contents, one sender and several receivers, both having a name and an address.



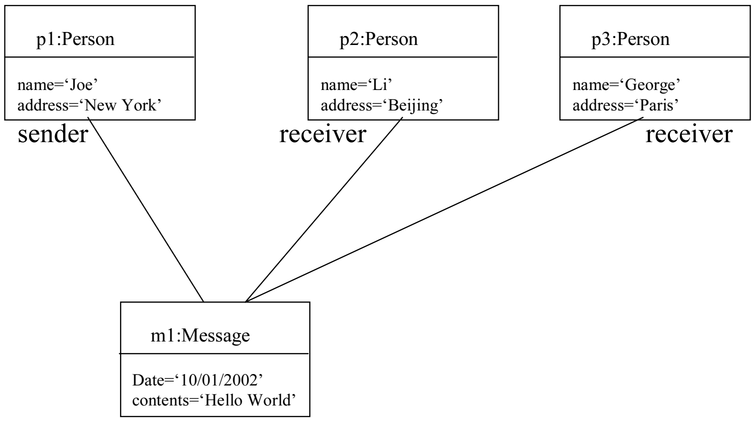
From this class diagram, and using the rules defined by the XMI standard, we can derive the following DTD defining the structure of a concrete message:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Extract from DTD Generated by: XMI
Application Framework 1.15 -->

<!--_________________________________________ -->
<!--                                          -->
<!-- XMI.reference may be used to refer to    -->
<!--data types not defined in the metamodel   -->
<!-- ________________________________________ -->

<!ELEMENT XMI.reference ANY >
<!ATTLIST XMI.reference
            %XMI.link.att;
>
<!-- ________________________________________ -->
<!--                                          -->
<!-- XMI.element.att defines the attributes   -->
```

```
<!-- that each XML element that corresponds to-->
<!-- a metamodel class must have to conform to-->
<!– the XMI specification.                    -->
<!--_________________________________________ -->

<!ENTITY % XMI.element.att
               'xmi.id ID #IMPLIED xmi.label
CDATA #IMPLIED xmi.uuid
                CDATA #IMPLIED ' >
<!--_________________________________________ -->
<!--                                          -->
<!-- XMI.link.att defines the attributes that -->
<!-- each XML element that to a metamodel     -->
<!-- class must have to enable it to function -->
<!-- as a simple XLink as well as refer to    -->
<!-- model constructs within the same XMI     -->
<!-- file.                                    -->
<!-- ________________________________________ -->
<!ENTITY % XMI.link.att
   'href CDATA #IMPLIED xmi.idref IDREF #IMPLIED
   xml:link CDATA #IMPLIED xlink:inline (true |
   false) #IMPLIED xlink:actuate (show | user)
   #IMPLIED xlink:content-role CDATA #IMPLIED
   xlink:title CDATA #IMPLIED xlink:show (embed |
   replace | new) #IMPLIED xlink:behavior CDATA
   #IMPLIED' >

<!--_________________________________________ -->
<!--                                          -->
<!-- CLASS: Correspondent                     -->
<!--_________________________________________ -->
<!ELEMENT Correspondent.name (#PCDATA |
    XMI.reference)* >
<!ELEMENT Correspondent.address (#PCDATA |
    XMI.reference)* >
<!ELEMENT Correspondent (Correspondent.name |
                         Correspondent.address |
                         XMI.extension)* >
<!ATTLIST Correspondent
            name CDATA #IMPLIED
            address CDATA #IMPLIED
            %XMI.element.att;
            %XMI.link.att;
>
<!--_________________________________________ -->
<!--                                          -->
<!-- CLASS: Message                           -->
<!--_________________________________________ -->
<!--                                          -->
<!ELEMENT Message.date (#PCDATA | XMI.reference)*
>
<!ELEMENT Message.contents (#PCDATA |
XMI.reference)* >
<!ELEMENT Message.sender (Correspondent)* >
<!ELEMENT Message.receiver (Correspondent)* >
<!ELEMENT Message (Message.date |
                   Message.contents |
                   Message.sender |
                   Message.receiver |
                   XMI.extension)* >
<!ATTLIST Message
            date CDATA #IMPLIED
            contents CDATA #IMPLIED
            %XMI.element.att;
            %XMI.link.att;
>
```

Example of a metamodel instance and its XML

A concrete message conforming to the message structure can be represented at the metamodel level by an object diagram. Inter-objet links match associations between the corresponding classes and class attributes are valuated by each instance.

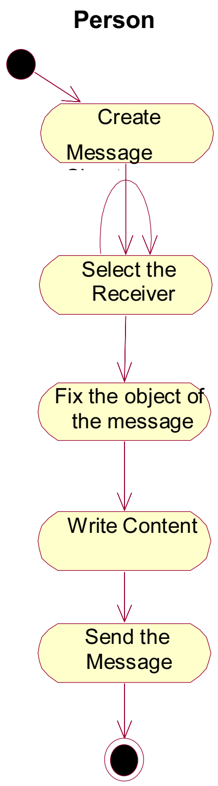


The XMI standard defines how to transform an object model into an instance of the DTD, so that XMI associates a DTD to a metamodel and an XML document conforming to the DTD to a model instance of the metamodel. Thus, models can be linearized and exchanged over a network.

```
<?xml version="1.0" encoding="UTF-8"?>
<XMI xmi.version="1.1" timestamp="Mon Apr 03 14:00:10
PDT 2000">
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>XMI Application Framework
        </XMI.exporter>
      <XMI.exporterVersion>1.05
        </XMI.exporterVersion>
    </XMI.documentation>
  </XMI.header>
  <XMI.content>
    <Message xmi.id="m1" Date="10/01/2002"
        contents="Hello World">
      <Message.sender>
  <Person xmi.id="p1" name="Joe"
             address="New York"/>
      </Message.sender>
      <Message.receiver>
        <Person xmi.id="p2" name="Li"
             address="Beijing"/>
        <Person xmi.id="p3" name="George"
             address="Paris"/>
      </Message.receiver>
    </Message>
  </XMI.content>
</XMI>
```

APPLICATION TO UML

Each UML CASE tool uses a specific format to store UML models, so that exchanging UML models between tools is only possible if the corresponding import functionality has been proposed. In the other way, the writing a tool to analyse an UML model is CASE tool dependent and supposes the existence and the knowledge of a CASE tool dependent language to navigate within the model. Consequently, using XMI to store and exchange models opens wide perspectives. In the following, we illustrate the use of XMI to store an UML model.



For each model, there are two levels of information which are taken into account: the modelled information and the constructions used to model the user's information, as for the considered diagram. In this example, we describe operations needed for a person to send a mail. To describe this process we have chosen to use an activity diagram, in order to show that not only UML class diagrams can be stored in XMI, but that XMI permits the description of any diagram. The actions performed by a person for the described process are represented as round rectangles. This diagram presents the succession of actions needed to perform the sending of messages since the starting point (initial Peudostate) until the end of the process (final state). The diagram indicates that each action needs to be completed before the following one starts and that no conditions are attached to each transition. Finally, we remark that the 'Select the Receiver' activity can be performed several times without limitation before triggering the 'Fix the object …' activity.

Each modelled information is rendered with its metadata which is defined in the framework of the UML metamodel. All common activities are described by an UML:ActionState metadata, whereas initial and final states are qualified in another way where attributes are sometimes used to distinguish a particular element. Note that each ActionState is described here with a name given by the modeller, incoming and outgoing transitions connected to the previous or following action, represented by arrows. The sequence of activities builds an UML: ActivityGraph which can be described in XMI as follows:

The sequence of activities builds an UML:ActivityGraph which can be described in XMI as follows:

```
<!—Whole UML:ActivityGraph -->
<UML:ActivityGraph xmi.id='S.282.1535.01.6'
   xmi.uuid='3DA5743C031D'
   name='contactSomeone' …
   context='S.282.1535.01.5' >
 <UML:StateMachine.top>
  <!-- Correspondance::contactSomeone::{top}
        [CompositeState] -->
   <UML:CompositeState xmi.id='XX.10.1535.2.3'
       name='{top}' … >
        <!—Set of states -->
   <UML:CompositeState.subvertex>
   <!--Correspondance::contactSomeone::{top}::
          [Pseudostate] -->
        <!—Initial state -->
     <UML:Pseudostate xmi.id='G.1' name='' …
        kind='initial' outgoing='G.2' />
     <!-- Correspondance::contactSomeone::{top}::
        [FinalState] -->
            <!—Final state -->
     <UML:FinalState xmi.id='G.3' name='' …
         incoming='G.14' />
     <!-- Correspondance::contactSomeone::{top}::
         Select the Receiver [ActionState] -->
            <!—Common Activities -->
     <UML:ActionState xmi.id='G.4'
        name='Select the Receiver' …
        outgoing='G.5 G.6' incoming='G.6 G.12'> …
</UML:ActionState>
     <!-- Correspondance::contactSomeone::{top}::
        Fix the object of the message
        [ActionState] -->
```

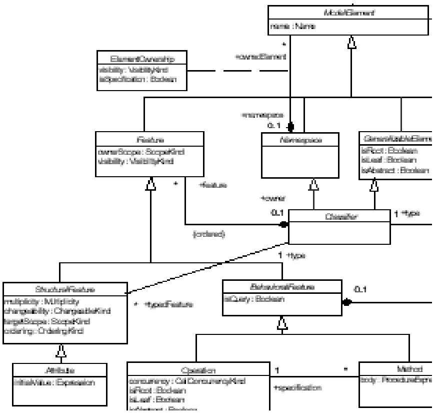
```
    <UML:ActionState xmi.id='G.7'
       name='Fix the object of the message' …
       outgoing='G.8' incoming='G.5' >…
    </UML:ActionState>
    <!-- Correspondance::contactSomeone::{top}::
       Write Content [ActionState] -->
    <UML:ActionState xmi.id='G.9'
       name='Write Content' … outgoing='G.10'
       incoming='G.8' >
          …
    </UML:ActionState>
    <!-- Correspondance::contactSomeone::{top}::
        Create Message Sheet [ActionState] -->
    <UML:ActionState xmi.id='G.11'
        name='Create Message Sheet' …
        outgoing='G.12' incoming='G.2' >
          …
    </UML:ActionState>
    <!-- Correspondance::contactSomeone::{top}::
        Send the Message[ActionState] -->
    <UML:ActionState xmi.id='G.13'
        name='Send the Message'
        outgoing='G.14' incoming='G.10' >
           …
    </UML:ActionState>
   </UML:CompositeState.subvertex>
  </UML:CompositeState>
 </UML:StateMachine.top>
       <!—Set of transitions connecting
            activities-->
 <UML:StateMachine.transitions>
   <UML:Transition xmi.id='G.2' name='' …
       source='G.1' target='G.11' />
   <UML:Transition xmi.id='G.5' name='' …
       source='G.4' target='G.7' />
   <UML:Transition xmi.id='G.6' name='' …
       source='G.4' target='G.4'/>
   <UML:Transition xmi.id='G.8' name='' …
       source='G.7' target='G.9' />
   <UML:Transition xmi.id='G.10' name='' …
       source='G.9' target='G.13' />
    <UML:Transition xmi.id='G.12' name='' …
        source='G.11' target='G.4' />
    <UML:Transition xmi.id='G.14' name='' …
   source='G.13' target='G.3' />
</UML:StateMachine.transitions>
<UML:ActivityGraph.partition>
<!--  Correspondance::contactSomeone::Person
      [Partition] -->
     <!— definition of responsible of
      activities -->
   <UML:Partition xmi.id='G.22'
      xmi.uuid='3DA5760700B0'
      name='Person' visibility='public'
      isSpecification='false'
      contents='G.1 G.4 G.7 G.9 G.11 G.13 G.3' />
   </UML:ActivityGraph.partition>
  </UML:ActivityGraph>
```

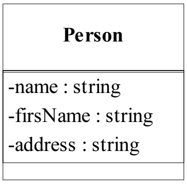
XMI PRODUCTION RULES

In any way, XML elements cannot be suppressed totally to benefits of compactness of attributes. Each XML feature has its own particular utility.

Production by object containment

"Most of metamodels are characterised by a composition hierarchy" [ref8]." XMI takes benefits of the major characteristic of XML: it is a hierarchical language. XML elements are principally used to enlighten composition depicted at metamodel level. For instance, a classifier which can be a Class, is a namespace which can be aggregated of modelElements like Attribute trough cascading inheritance via Features and structuralFeature. This example explains how attributes can be part of a class.





The composition relationship is well adapted to be related as containment or hierarchy of objects. This containment is expressed XML tag containment, each step representing a navigation step trough intermediate constructions like:

```
<!— 'Class' inherits from Classifier-->
<UML:Class xmi.id='c1' name='Person' >
     <!— Content of Classifier is a set of
         Feature-->
     <UML:Classifier.feature>
     <!— This content can be an Attribute as
         StructuralFeature child-->
        <UML:Attribute xmi.id='a1' name='name' …>
            …
        </UML:Attribute>
            …
     </UML:Classifier.feature>
</UML:Class>
```

This principle allows to declare each feature as part of its composite structure following the composition links given at the metamodel level by associations with filled-diamond at the origin point of the navigation. This allows too, to declare one time each feature which can be qualified by an unique identifier named 'xmi.id', that constitutes an unique reference of the declared element. More than given an extra representation of models, XMI gives an understandable version of implementation of the UML structure and syntax using any model an user can make.

This first principle – object containment for composition links – is after all insufficient to relate the full richness of UML. In the latest case attributes were part of their Classifier which was a Class, but relationships between modelled elements cannot be expressed fully – this principle is only usable for diamonds associations, but what about the others? To complete this mechanism XMI defines production by package extent.

Package extent production

Relating modelled information forces to describe each construction defined by the modeller. But there is no need to find in this description multiple declaration of these elements. To prevent from multiplying occurrences of the same element in the XML file, XMI defines a way to reference already or not yet declared structure. This facility is given by the use of the xmi.idref attribute which can both contain a reference towards a local element (contained in the same XMI file) or in an external document as an URL expression. Any xmi.idref attribute value must be an xmi.id existent value. For instance, any attribute of a class must have a data type we don't need to declare several times. DataTypes in the UML metamodel inherits features from Classifier and are linked to attribute via StructuralFeature inheritance. One can say, as regard to the UML metamodel a StructuralFeature see a Classifier as its type.

```
<UML:Class xmi.id='S.244.1337.52.1'
      name='Person' … >
    <UML:Classifier.feature>
       <UML:Attribute xmi.id='S.244.1337.52.4'
           … type='G.7' >
       …
       </UML:Attribute>
    </UML:Classifier.feature>
  </UML:Class>
  <!-- ======= String    [DataType] ======= -->
  <UML:DataType xmi.id='G.7'
    name='String' … />
```

The reference is inserted as content of the association's role name that allows to access the datatype feature. This mechanism implies that the knowledge of the whole XMI file must be effective when trying to navigate over those types of links. This implies too, that verifications of destination links must be done to retrieve distant information. Both of these considerations are sufficient to force this mechanism to be active only if the file is valid as regard of its DTD.

Major’s change between XMI 1.0 and XMI 1.1

As we have seen in the preceding example, the expressions of the two XMI versions are rather different. The same information is present but attributes are more used in the latest form of XMI than attributes were in the past one. This major change allows to reduces significantly the length of the XMI files and bring a notice-able gain in any transformation process with XSL. In fact, this change is due to philosophy change in representing information. We can understand this change taking a look to the UML Metmodel (UML Metamodel for Attribute).

In the fist XMI version (1.0) each information is declared in the context of the belonging abstract or concrete metamodelElement. As for example the declaration of the attribute name ‘fisrtName’ obliges to enclose this declaration in the ‘modelElement’ class where it has been specified. This mechanism forces to have similar redundant information for each concrete information the user has modelled. The novelty of the XMI 1.1 version is to consider that if a concrete class inherit features from ancestors, these features must be considered as local and can be represented as attributes of the class they are propagated in, without the original class of this feature have to be precised. This view allows a better readability of the XML documents produced

OTHER

The main idea which leads to elaborate a new language such XML was to offer a language everybody can extend, read, control, use to mix/extract information, use to present information in different ways for different supports, use to interchange data.

The reduced complexity compared to SGML (Standard Generalized Markup Language) and its connection with HTML (HyperText Markup language) has lead to the quick emerging of XML as a standard for data storage and exchange.

XML is now used to exchange information, because this information is qualified by its means and so easily identifiable, to model information because of its capability of structuring this information and for manipulation of the information using tools. Today lot of tools implement an XML export of proprietary storage formats or store all their data in XML.

Within Neptune project we have used XML format to be independent of proprietary storage formats. XML is both used to feed UML models into Neptune in order to check or document those models. XML is also the storage format for all the transformations graphically designed by the user.

In this part we focuses on data representation with XML and optional XML declarations are out of our scope.

XML document

"An XML is a text document composed of declarations, elements, comments, character references, and processing instructions, all of which are indicated in the document by explicit markup." This document must have at least one XML element as entry-point called the 'root' and must be well-formed, that means each element, within the document, opened with its start-tag must be closed by its respective end-tag or be an empty XML element – included the root element.

An XML document must have an XML element as content. This element can be split in several elements or can be an empty element with no content. The two examples following are well-formed XML documents:

```
Ex 1: <Book>
          <Title>My Title</Title>
          <Summary> Nowadays … >/Summary>
    </Book>
Ex 2:
    <Book/>
```

but the following example doesn't satisfy the well-formed constraint, that means this document is not an XML document:

```
    <Book>
          <Title>My Title
          <Summary> Nowadays …</Summary>
    </Book>
```

We can formalise this definition by giving a definition of what is an XML document:

```
      XMLDocument::= XMLElement | XMLEmptyElement
```

DTD and valid documents

As already stated an XML document must be contained in one element called "the root" and be well-formed to be considered as an XML document. This basic principle constrains to close any opened tag, and that's all. But, if a document can be so easily written it must be also easily understood. Due to the fact that all is allowed if the well-formed constraints are satisfied, no one could say that a document has similar meaning of another if its content has changed or if the tags to express information have changed. This means that if a community of users wants to use XML to exchange XML coded documents this community has to decide what kind of information she needs and so fixes the vocabulary used to encapsulate information. This process implies that together the community users defined a particular grammar of their language in a non XML-based particular document called DTD (Document Type Declaration).

The benefits of this DTD are multiple:

- First, everyone has the insurance that if an XML document conforms to its DTD then the document uses only the vocabulary fixed in the grammar,
- Second, everyone can be sure that required information is present,
- Third, everyone can be sure that the expressed information respect the containment hierarchy defined by the DTD authors,
- Fourth, everyone can be sure internal references expressed with IDREF(S) are resolved.

To describe the content of a document class in a DTD, the writer describes all the elements and attributes used in the XML documents. No particular order is mandatory, but for a better understanding it is logically admitted to begin with the description of the root element.

For each element the writer must specify the name of the element and its content. The content of the element can be given by a list of sub-elements or several sub-lists of elements, enclosed within parenthesis. The list of elements may be affected with a multiplicity (?: 0 or 1, *: 0 or several, +: 1 or several) to specify if the whole content can be optional, mandatory one or several times. More, in the list each element may also be affected with a multiplicity to constrain its presence. Connectors between sub-element may be:

|: indicates only one of the elements or list connected with this operator may appear,

,: indicates elements must appear in the given order,

(): delimits elements or lists of elements connected with "|" or ",".

The content of an element can also be PCDATA (String) or a mix with PCDATA and lists or elements declarations.

The declaration of each attribute must include the name of the attribute, its type and the clause specifying if the attribute is IMPLIED, REQUIRED or FIXED .

Example: DTD for last Book document

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!ELEMENT Book (Chapter*, Toc) >
<!ELEMENT Chapter (Section)* >
<!ATTLIST Chapter
      idElt ID #REQUIRED
      titleChap CDATA #REQUIRED
      startpage CDATA #REQUIRED >
<!ELEMENT Section EMPTY >
<!ATTLIST Section
      idElt ID #REQUIRED >
<!ELEMENT Toc (TocEntry)* >
<!ELEMENT TocEntry EMPTY >
<!ATTLIST TocEntry
      chap.idref IDREF #IMPLIED >
```

To complete the process and to use a such DTD the writer must specify an additional declaration <!DOCTYPE Book SYSTEM 'bookDTD.dtd'> specifying that the XML document must be analysed in conformance with its external DTD. This declaration for an external DTD must obey the following syntax:

```
    <!DOCTYPE NameOfRootXMLElement SYSTEM
           'DTDPathFile'>
```

APPENDIX D

Overview of Model transformation and XSL

Software development consists in writing and transforming models, i.e. more or less precise representations of a physical or conceptual world. Models can describe interactions between a system and its environment, the internals of a system with architecture, data or behavioural considerations. Several formalisms can be used to represent these models. Graphical high level descriptions are intended to be read by non specialists; executable (binary) models will be executed on the target architecture. For the sake of reusability, intermediate models will be made independent of the architecture. They come from high level programming language sources to software architecture description models that can be mapped to various physical architectures. Model transformation is needed here and generalises compilation of a code written in a programming language into binary code. Consequently, model transformation is an important concept which has been recently emphasised by the OMG through the MDA (Model Driven Architecture) proposal. The starting point of this proposal is the UML notation which supports the expression of various models at very different levels of detail. For model transformation to be automated, it is necessary to define the precisely the syntax of the modelling language. In fact, two notions of syntax must be distinguished. The concrete syntax of the formalism defines how specifications are displayed in terms of correct sequences of characters or in terms of graphical elements. The abstract syntax is not concerned by such visual features, but defines at a more abstract way what concepts are needed and what are the relations between these concepts. The abstract syntax representation of a document captures its essence and ignores visual aspects, which can be retrieved through the use of pretty printers. Meta modelling consists in specifying these concepts and relations. A meta modelling language can be used to define the abstract syntax of several languages and thus can be a support to write transformations between these languages.

In the framework of UML, the concrete syntax is graphic and informally specified. It is based on rectangles, lines, arrows, etc. However, the abstract syntax of all UML diagrams and thus of all the formalisms defined by the UML notation can be described formally using the MOF, which is mainly a subset of UML restricted to class diagrams. It is thus graph-based as opposite to textual languages which are tree-based. Then, transforming a model is to rewrite a graph. Consequently, it is possible to tackle models transformations using graph grammars. This way was notably experienced by a German team which has produced the system PROGRESS. A special version of PROGRESS is dedicated to UML models.

PROGRESS appears as a tool devoted to process graphs ; it is composed by :

- a graphical and textual editor,
- an incremental analyser,
- a browser to travel through the graph to be processed.

According to the authors, PROGRESS mainly addresses the following areas :

- huge graphs where little ones are to be recognised,
- environments including customised indexation system,
- graph grammars with a low number of alternatives to be considered at each rewriting step.

Moreover, according to the background of graphs rewriters, in general, graphs grammars are ambiguous thus designing transformations systems resting on this approach will lead to inefficient results. Another way to deal with models transformations is to rest on meta models and related tools. According to this approach, the semantics of modelling languages are described in the same formalism, Meta Object Facility (MOF) for instance, transformations are then considered as translations from a source language to a target language, the semantics of both of the languages are depicted using the same meta model. This solution was adopted by the Distributed System Technology Center through a specific software called « dMOF». A similar approach was taken by France Telecom which has produced Univers@lys devoted to UML. This system is used to store the models, the transforming processes are implemented by the mean of API automatically generated by the software according to the transformation requirements.

Among the models transformations process, the ones resting on the language XSLT are very popular. In the following, we are going to describe this approach.

Currently the norm to format digital texts is the eXtended Mark up Language (abbreviated XML). XML is a revisited version of the former standard SGML avoiding some drawbacks at syntax and semantics levels. Dealing with text encoded in the XML format, supposes the existence of a software processor to transform the XML encoded texts according to an output format, HTML for instance, such an output protocol is called a « style sheet ». Thus XML was provided with a specific language devoted to process style sheets; this language is called « XML Stylesheet Language » (abbreviated XSLT). The aim of XSLT is to transform any text encoded in the XML format into another text the structure of which is expressed by a stylesheet. In the field of models and meta models for software engineering, graphical formalisms are very commonly used but, in order to process or exchange them, textual standards are necessary involved. As a model in a textual form is but a text, XML was chosen as a standard to encode textual representation of graphical models. The most popular example is the standard XMI (XML Models Interchange) devoted to the exchanges of UML models depicted in textual format. Consequently, it appears that transforming and processing ? models necessitates a language able to implement transformations of XML texts, of course XSLT was the natural candidate to do such a job.

XSLT is a declarative language; a program is set of unordered rules. Each rule called a »template » aims at matching a sub-tree of a syntactic tree describing an XML text in order to produce any relevant information corresponding to the sub-tree. Syntactically speaking, the general form of rule is:

```
<xsl :template match = « sub-tree attribute » >
    produced information
</xsl :template>
```

« sub-tree attribute » qualifies the root of a sub-tree, it can be the name of a node, the children of a node, a line descent; a node can also be depicted by its XML attributes. In order to facilitate the traversal of a tree, XSLT provides the programmer with conditional, loop and procedure call statements. The expressive power of the XSLT language is difficult to state: such a point is not clearly tackled in the reference documents provided by W3C. Some Finnish works about this topics conclude that this language possesses the power of Turing's machine, but this conclusion is not well established in the material we could access to; according to our experience, it seems that XSLT is a kind of attribute grammar allowing to compute semantic attributes in one traversal of the syntactic tree, so multi-visits of a tree requires as many XSLT processors calls as necessitated passes on the tree.

XSL

As opposite to HTML which mixes information and presentation, XML documents only encode information. XSL has been introduced by the W3 consortium to define stylesheets applying to XML documents.

These stylesheets define how to visualise information stored in XML documents compatible with a given DTD. XSL is composed of three parts: XPATH, XSLT and XSL-FO. Due to the extensible structure of XML documents, which is tree-based, the W3 consortium was driven to define first a language based on the core features of a tree: its hierarchical node structure. This has been made through the XPATH specification that provides functionalities to navigate within the tree structure. Then, the XSLT specification has been published, including XPATH facilities, in order to allow any transformation of an XML source document through the attachment of transformation rules to the nodes of an XML tree. Finally, the XSL environment has been completed with the evolved formatting capabilities of XSL-FO. Implementations of this set of norms are available today and allow any transformation of an XML document through a given set of transformation rules defined by an XSL stylesheet.

XPATH

XPATH is not an XML-based language but a functional language over path expressions, strings numbers and booleans. Path expressions provide a way to designate a set of nodes of an XML document, either using relative or absolute navigation expressions. XPATH essential contribution is the definition of a syntax that allows addressing any part of an XML document using meta-data spelling. Regular XPATH expressions follow the model "Axis::FilterNode[ListOfPredicate]" where:

- Axis: may be a forward axis as Child or Descendant or a reverse axis as ancestor, preceding-sibling, and others. Some shortcuts allow more compact expressions,
- FilterNode: may be a list of node given in expanded form specifying all the nodes to access data, or shortcut expressions,
- ListOfPredicate: allows conditional selection of a subset of the current set of nodes.

Using axes the writing of navigation gains on compactness and allows a good readability of XSL stylesheets. Using child or descendant axes do not penalise the performances of an XSLT processor, but using the ancestor axis is very awful. Such a traversal is not the natural traversal of an XML document using XSLT, that naturally performs top-down traversals.

Sometimes, navigation needs to reach distant ancestors or distant elements, for example in order to retrieve a context or to point to a type declaration. XPATH provide a mechanism to prevent from long search through ancestors: it is possible to declare pointers to designate nodes and to follow them. This mechanism is based on the XPATH functions *key* and *id*. The id function is of higher level and acts as follows: it takes as argument an XPATH expression of which value must be the one of an attribute of a node of the XML tree. This attribute must be declared of type ID in the DTD. Nodes containing this attribute must be uniquely determined by it. Then, the *id* function returns this unique node. Consequently, such a navigation is only possible within a valid document with respect to its DTD. The use of the *id*() function speeds a lot any navigation using the ancestor axis if the child element specifies a reverse link towards its ancestors using an IDREF typed attribute.

*For example, c*onsider the following XML document describing the possessions of a person. This document contains internal links between a car and its owner, and between a parking and the car it is assigned to.

```
<doc>
   <Person id=″p1″ name=″Dupond″>
     <Owns>
       <Car num=″c1″ mark=″renault″
            parking=″pk1″/>
       <Car num=″c2″ mark=″peugeot″/>
     </Owns>
   </Person>
   <Person id=″p2″ name=″Durand″>
     <Owns>
        <Parking place=″pk1″ state=″rented″
                 tenant=″p1″/>
     </Owns>
   </Person>
</doc>
```

- From the root of the document, the `descendant::Car` path specifies the set of cars present in the document,
- This set can be restricted to cars of a given mark: a predicate specifies a particular value for the mark attribute :
- descendant ::Car[@mark=”renault”],
- From a car, we can select all declared cars of the same mark:
- /doc::descendant::Car[@mark=current()/@mark],
- From a car, we can follow the links and get the name of the parking renter :
- id(@parking)/ascendant::Person/@name

- or from a parking we can identify the car parked :
- id(@tenant)/descendant::Car[@parking=
- current()/@place] .

XSLT

The goal of XSLT is to offer capabilities to specify transformations. An XSLT transformation is specified by a couple containing a pattern for the selection of XML nodes and a term defining the replacement for each selected node. Applying transformations builds a document that can be in XML format from an XML document.

The particularity of XSLT is defined both by the XML nature of this language and by the implied recursive process used to reach any node of the source XML document.

XSLT and XML

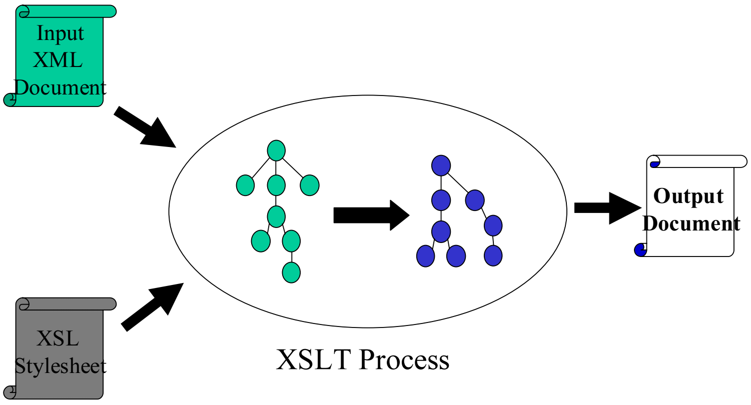
First of all, XSLT instructions are XML expressions. For example, the detail of any transformation is part of the main `<xsl:stylesheet>` element, root of the transformation document terminated by its respective closing tag `</xsl:stylesheet>`. Another example is given by the current use of the `<xsl:apply-templates/>` instruction to process descendants of the context node using the empty XML element form. Intensive use of attributes is also made in XSLT to define the kind of XML information searched. For example, the definition context of transformation rules can be defined with the use of the match attribute of the `xsl:template` used to filter a particular XML element or the `select` attribute of the `xsl:value-of` element used to extract the value of the pointed attribute.

```
<xsl:stylesheet>
  <xsl:template match="Person">
        <xsl:value-of select="@name"/>
        <xsl:apply-templates/>
  </xsl:template>
   …
</xsl:stylesheet>
```

In the same manner XSLT defines all the features of a classic language like conditional, iterative structure and some evolved capabilities like numbering or sorting instructions.

XSLT processing

The processing of XSLT stylesheets performs tree rewriting. That means that any tree represented by an XML document can be transformed into another tree. The XSL transformation applies on a current node, root of its sub-tree, and processes the node list including the current node and its descendants. From the input tree, XSLT processing may filter, reorder or add new nodes to the result tree through the definition of template rules.



Transformations depend on implied mechanisms and explicit definition of template rules.

Designing transformation

Designing a transformation of a source XML document to any destination format requires to deal with inherent process of XSL and to define concrete rules in charge of the extraction/reorganisation of the information.

XLST implied mechanisms

The first thing an XSLT writer must consider is that in all XSLT processes some rules are implied. The first of these rules examines all the elements of an XML document from the root to all the leaves, and the second rule outputs any text information contained in a node.

What must be very well understood, referring to the first mechanism, is that the descending traversal is processed, by default, until a node is matched by a template rule. The template rule is supposed to process the sub-tree having this node as root. In other terms, by default, the descendant traversal is stopped at the highest level of the tree for which a template rule exists. For example, the following set of rules will not process the Parking sub-tree because this information is intended to be processed in the context of the “Person” node, which appears first in the top-down traversal of the XML document:

```
<xsl:template match=”Person”>
 <xsl:value-of select=”@name”/>
</xsl:template>

<xsl:template match='Parking' priority = '5'>
 <xsl:value-of select="@place"/>
</xsl:template>
```

However, it is possible to continue the descendant traversal from a pointed context node. This ability is given by specifying that some/all written templates must be applied over descendants of the context node, allowing their transformation. This recursive call is launched by the <xsl:apply-templates/> instruction:

```
<xsl:template match=”Person”>
  <xsl:value-of select=”@name”/>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match='Parking' priority = '5'>
  <xsl:value-of select="@place"/>
</xsl:template>
```

In this case, all the descendants of Person will be considered and *Parking* information will be output, namely the value of the attribute *place*.

Whereas the default top-down traversal rule is necessary, the implied text output rule can be awkward, because the goal of a transformation is not necessarily to output text for each node: we can imagine XML transformations for versioning support, production of HTML pages or documents of any format. In most cases, the goal of a transformation is to extract piece of information from the source document and to produce a new document with a different structure, so that automatic text production rules cannot be applied. To meet this requirement, the default rule must be occulted using an XML empty element specifying no action when a text node is encountered. The following declaration overrides the default rule and allows satisfying this goal:

```
<xsl:template match=″text()″/>.
```

In this case, only text extracted in the context of the matched element using `<xsl:value-of select=″…″/>` instruction will be part of the result document.

Defining transformation rules

All XSL transformation processes are based on the definition of rules written in order to specify how the XML source information is transformed. Two kinds of rules exist: matching rules apply to nodes matching the given template while named rules are explicitly launched by a caller rule.

- Matching transformation rules

  Defining a transformation rule is generally made with a match clause A transformation rule generally contains a match clause which identifies a set of nodes. Using this capability offers the possibility to consider that all the nodes corresponding to the specified description are processed in the same way. It means that defined transformations is based on meta-data identification.

  The following example performs the identity transformation for all the nodes of the input document except for the *Person* node for which the transformation is to include an additional attribute called *Country* whose value is fixed to *France* :

```
<xsl:template match='Person'>
    <xsl:copy>
       <xsl:apply-templates select='@*'/>
           <xsl:attribute name="Country"
           namespace="">
               France</xsl:attribute>
       <xsl:apply-templates/>
    </xsl:copy>
</xsl:template>
<xsl:template match='@*|node()'>
    <xsl:copy>
       <xsl:apply-templates select='@*|node()'/>
    </xsl:copy>
</xsl:template>
```

This example shows how to define a transformation applying to all persons. It is also possible to write more precise filters using a combination of meta-data and data values to address specific elements:

```
<xsl:template match='Person/[@name='Joe']'>
  <xsl:copy>
    <xsl:apply-templates select='@*'/>
          <xsl:attribute name="Country"
                namespace="">     USA
          </xsl:attribute>
    <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>
```

The two rules applying to non-disjoint sets of nodes, ambiguities may occur. In fact, a conflict resolution principle is defined so that the more specific rules are privileged. An explicit priority can also be assigned to a rule.

- Named transformation rules

As every programming language, the writing of XSLT rules can give only a single file with a unique block of instructions or can be organised with the definition of libraries or functions.

All transformations are made in the context of the identified elements. However, the number of lines in a rule is not limited and for a possibly long rule several information must be extracted in the context of a particular container element. XSLT specification allows defining named rules without using a match clause to split the whole transformation into more readable pieces. The different named rules shall be called from the rule that defines the context node. Using such method does not change the context node, which remains the node in which the call has been done.

For example the extraction of the name of a “Person” can be made calling a specific named rule like :

```
<xsl:template match=”Person”>
    <xsl:call-template name=”PersonName”/>
</xsl:template>
```

```
<xsl:template name=”PersonName”>
    <xsl:value-of select=”@name”/>
</xsl:template>
```

- Using Namespaces

  Namespaces allow XSL stylesheets to intersperse different XML languages in order to address different needs. For example the same stylesheet can both contain XSLT statements to address tree transformations, XSL-FO or/and SVG instructions to define the structure and presentation of a document. Obviously it is not spring of a pure XSLT processor to treat HTML, FO or SVG features included in the stylesheet. In such cases three different processors are necessary to produce the whole transformation. Namespace allow processors involved in a transformation to identify statements they are concerned about.

  Namespaces are identified by an URI to which a label is associated. Because label is user defined processors are intended to know the correct URI of the language. Namespaces of a stylesheet embedding XSLT and FO statements is given for example by the following expression :

```
<xsl:stylesheetversion="1.0"
 xmlns:xsl=
    "http://www.w3.org/1999/XSL/Transform"
 xmlns:fo="http://www.w3.org/1999/XSL/Format">
```

  and can be used to defined a structure of a page like:

```
<xsl:template match="/">
  <fo:root>
    <fo:layout-master-set>
      <!-- layout for the first page -->
      <fo:simple-page-master
          master-name="first"
          page-height="29.7cm"
          page-width="21cm"
          margin-top="1cm"
          margin-bottom="1.5cm"
          margin-left="2.5cm"
          margin-right="2.5cm">
           …
      </fo:simple-page-master>
    </fo:layout-master-set>
    <!-- end: defines page layout -->
</fo:root>
</xsl:template>
```

This example defines an XSLT transformation that processes an XML document from its root and produces the structure of the page presentation using FO capabilities.

Avoiding rules conflicts

Designing transformations implies the writing of many rules, each rule applying to a category or a specific source element. However, transformations may be context dependent. For example a name may be treated differently if it belongs to a car description or a name description. Here we study the different principles helping to write rules applying to specific contexts.

Using depth level of information

If a specific transformation applies when a node is has a particular father, a relative path to the node can be mentioned, as illustrated by the following example:

```
<xsl:template match="doc/Parent">
```

The corresponding transformation applies to *Parent* nodes having a *doc* node as father. Furthermore, it has a greater priority than the rule matching *Parent,* so that it is possible to declare a general transformation together with a specification transformation for such *Parent* nodes.

Distinguish a particular instance

A specific transformation can also be attached to unique instances or to instances having specific valued attributes through the use of a predicate testing the value of some attributes:

```
<xsl:template match=" Parent[@name='Joe']">
```

However, this rule can enter in conflict with a general rule matching *Parent*. Actually, the presence of a predicate does not confer to the rule a higher priority. The XSLT processor signals the ambiguity, which can be solved using explicit priority declarations.

Using priority

Ambiguity can be left by using a priority clause in the template rule expression. This priority forces the execution of the affected rule if the priority is greater than 0.5, which is the default priority for a rule, or greater than the priority of another conflict rule. The XSLT norm gives the "*modus operendi*" to calculate the priority of a rule - refer to the XSLT norm for detailed information.

```
<xsl:template match=" Parent[@name='Joe']"
   priority='2'>
```

This example solves the conflict in the previous situation.

Using mode attribute in template rules

Specific transformations can be identified by a *mode*. For this purpose, a *mode* attribute can be attached to a template rule. Rules with a given mode are taken into consideration during the recursive traversal of the sub-tree of a node, if the specific mode is specified in the *apply-templates* call. For example, a mode can be used to separate rules extracting all information concerning a person and rules extracting only the name of the person:

```
<xsl:template match = "Person"
              mode = "PersonIdentification">
      <xsl:value-of select="@name"/>
</xsl:template>
```

This template rule can be called using:

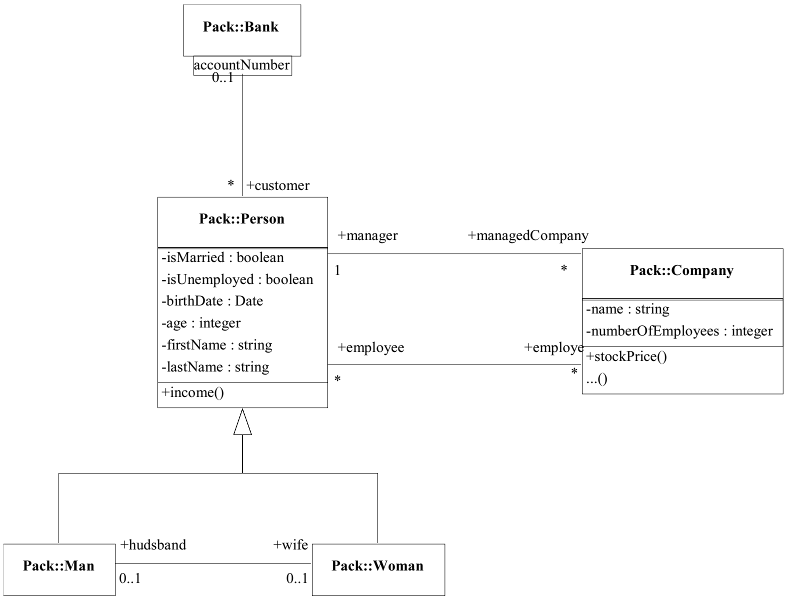
```
<xsl:apply-templates
   mode="PersonIdentification"/>
```

Complete example of UML Model Transformation

The following example presents an UML model made of a class diagram and the result of two transformations: the first one generates Java code and the second one a PDF document containing the same Java code with visual improvements. The source code of the XSL transformation used to produce the PDF document is given.

UML Model

XSL transformations apply to the following UML model which covers several features of UML class diagrams: classes with public or private properties, inheritance links, and qualified associations.



Java code generation

An XSL transformation applying on the XMI representation of the model can be used to produce the following Java code. XSL rules are not given as they are in fact similar to the ones used to generate the PDF document, except the statements related to the XSL-FO generation. Java code generation is in fact straightforward and follows the following principles:

- Inheritance links are translated without transformation, which means that multiple inheritance must be forbidden.
- Attributes and operations are found within the class declaration in the XMI file and are output when they are traversed.
- Associations are described outside classes. Thus association ends referencing the current class must be searched for, so that the name of the opposite role can be added to the class attributes. Multiplicities greater than 1 are expressed using arrays.
- Qualified associations are encoded by operations taking the qualifier as parameter.

It has to be noted that improving code generation with the addition of get and set methods for attributes, the management of association classes, etc. would be easy.

```
Package Pack;

  public class Bank
  {
    // Associations
    public Person customer (
        Integer accountNumber);
  }

  public class Company
  {
    // Attributes
    private String name ;
    private Integer numberOfEmployees ;

    // Operations
    public  stockPrice();
    public Company();
    public  employ(Person p, Person q);
    public  demising(Person p);

    // Associations
    public Person manager;
    public Person employee[];
  }

  public class Person
```

```
{
  // Attributes
  private Boolean isMarried ;
  private Boolean isUnemployed ;
  private java.util.Date birthDate ;
  private Integer age ;
  private String firstName ;
  private String lastName ;

  // Operations
  public  income();

  // Associations
  public Company managedCompany[];
  public Company employer[];
}

public class Woman extends Person
{
  // Associations
  public Man husband;
}

public class Man extends Person
{
  // Associations
  public Woman wife;
}
```

Documentation generation using FO rendering

This paragraph presents the complete source code of the XSL transformation producing a Java source code with FO rendering from an XMI document. The Java code is the same as the one presented before, but each kind of information (package, class, attributes, operations) is output with a specific rendering. In fact, the PDF document is obtained after a two passes process:

- First, the XSL processor is run on the XMI model. The output document is an XML document with XSL-FO tags.
- Second, the fop tool is run and outputs the PDF document.

Document generation is thus highly dependent on the use of XML namespaces which identify XSL instructions used by the first path and FO instructions generated by the first path and used by the second path. Hence, the XSL stylesheet contains both kind of instructions.

XSL Stylesheet

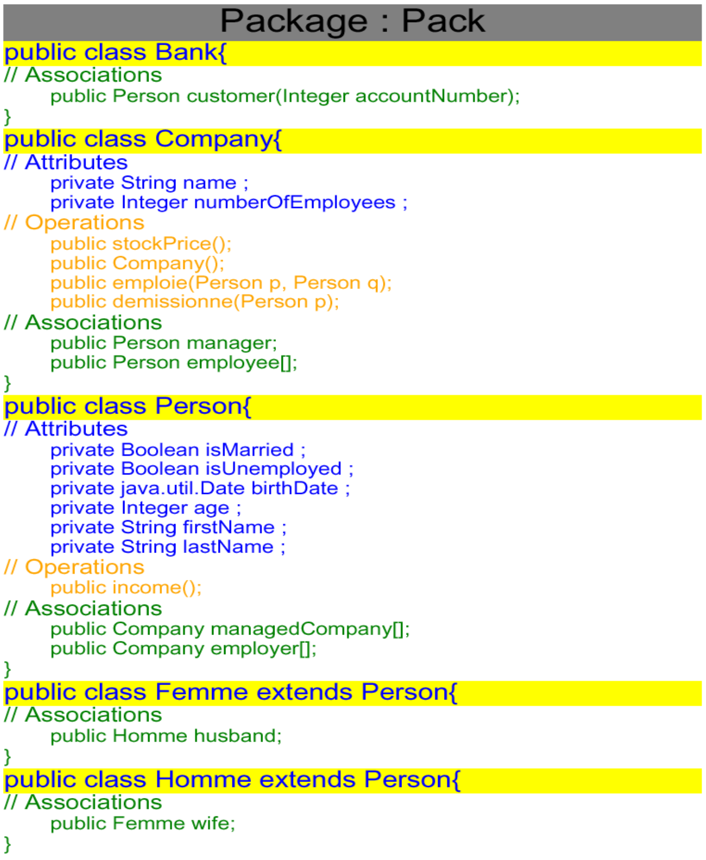
The main characteristic of this stylesheet is that it uses three families of XML tags identified by `xsl,fo` and `UML` namespaces. The `xsl` namespace identifies instructions used to produce the `fo` document from the `UML` document. Thus, `UML` tags are used in match instructions, while `fo` tags are output.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:UML="http://org.omg/UML/1.3"
  xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <xsl:output method="xml" encoding="utf-8"
      indent="yes"/>

 <xsl:template match="/">
 <fo:root>
   <fo:layout-master-set>
     <!-- layout for the first page -->
     <fo:simple-page-master master-name="first"
         page-height="29.7cm"
         page-width="21cm" margin-top="1cm"
         margin-bottom="1.5cm"
         margin-left="2.5cm"
         margin-right="2.5cm">
       <fo:region-body margin-top="1.2cm"
          margin-bottom="1.5cm"/>
       <fo:region-before extent="1cm"/>
       <fo:region-after extent="1cm"/>
     </fo:simple-page-master>
    </fo:layout-master-set>
    <!-- end: defines page layout -->
    <xsl:apply-templates/>
  </fo:root>
</xsl:template>

<xsl:template name="TableEditor"
     match="UML:Package">
   <fo:page-sequence master-reference="first">
      <fo:flow flow-name="xsl-region-body">
```

```
        <fo:block line-height="30pt"
           font-size="25pt"
           text-align="center"
           background-color="grey">
        Package: <xsl:value-of select="@name"/>
        </fo:block>
        <xsl:apply-templates/>
       </fo:flow>
    </fo:page-sequence>
</xsl:template>

<xsl:template match="UML:Class">
   <fo:block line-height="20pt" font-size="18pt"
      text-align="left" background-color="yellow"
      color="blue">
     <xsl:value-of select="@visibility"/>
      class
     <xsl:value-of select="@name"/>
     <xsl:for-each select="id(@generalization)">
       <xsl:if test="position()=1"> extends
       </xsl:if>
       <xsl:value-of
           select="id(@parent)/@name"/>
       <xsl:if test="not(position()=last())">,
       </xsl:if>
      </xsl:for-each>
      <xsl:text>{&#10;&#9;</xsl:text>
    </fo:block>
 <xsl:apply-templates/>
 <fo:block line-height="18pt" font-size="16pt"
    text-align="left" color="green">
   // Associations
 </fo:block>
 <fo:block line-height="18pt" font-size="14pt"
    text-align="left" color="green">
 <xsl:for-each
    select="/descendant::UML:AssociationEnd
    [@participant=current()/@xmi.id]">
   <xsl:if test="preceding-sibling::
        UML:AssociationEnd !=''">
      <xsl:apply-templates
          select="preceding-sibling::
          UML:AssociationEnd" mode="context"/>
    </xsl:if>
    <xsl:if test="following-sibling::
         UML:AssociationEnd !=''">
      <xsl:apply-templates
          select="following-sibling::
          UML:AssociationEnd" mode="context"/>
    </xsl:if>
   </xsl:for-each>
   <xsl:text>}&#10;&#10;</xsl:text>
 </fo:block>
</xsl:template>

<xsl:template name="MultiplicityRange">
  <xsl:if test="descendant::
      UML:MultiplicityRange/@upper != '1'">
    <xsl:text>[]</xsl:text>
  </xsl:if>
</xsl:template>

<xsl:template match="UML:AssociationEnd"
    mode="context">
  <fo:block line-height="16pt" font-size="14pt"
     text-align="left" color="green"
     start-indent="30pt">
   <xsl:if test="@isNavigable != 'false' and
       @name!=''">
      <xsl:value-of select="@visibility"/>
      <xsl:text>&#x20;</xsl:text>
      <xsl:value-of
          select="id(@participant)/@name"/>
      <xsl:text>&#x20;</xsl:text>
      <xsl:value-of select="@name"/>
      <xsl:call-template
          name="MultiplicityRange"/>
      <xsl:call-template name="AssociationKey"/>;
     </xsl:if>
   </fo:block>
  </xsl:template>

  <xsl:template match="UML:Class/UML:
      Classifier.feature">
    <fo:block line-height="18pt" font-size="16pt"
       text-align="left" color="blue">
      // Attributes
    </fo:block>
    <xsl:apply-templates select="UML:Attribute"
      mode="attribute"/>
  <fo:block line-height="18pt" font-size="16pt"
     text-align="left" color="orange">
  // Operations
  </fo:block>
  <xsl:apply-templates select="UML:Operation"
      mode="operation"/>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="UML:Attribute"
    mode="attribute">
  <fo:block line-height="16pt" font-size="14pt"
     text-align="left" color="blue"
     start-indent="30pt">
    <xsl:value-of select="@visibility"/>
    <xsl:text>&#x20;</xsl:text>
    <xsl:value-of select="id(@type)/@name"/>
    <xsl:text>&#x20;</xsl:text>
    <xsl:value-of select="@name"/> ;
  </fo:block>
 <xsl:apply-templates/>
</xsl:template>

<xsl:template name="AssociationKey">
  <xsl:if test="descendant::UML:Attribute
      !=''">
    <xsl:text>(</xsl:text>
    <xsl:for-each select="descendant::
        UML:Attribute">
      <xsl:value-of select="id(@type)/@name"/>
      <xsl:text>&#x20;</xsl:text>
      <xsl:value-of select="@name"/>
      <xsl:if test="not(position()=last())">,
      </xsl:if>
    </xsl:for-each>
    <xsl:text>)</xsl:text>
  </xsl:if>
</xsl:template>

<xsl:template match="UML:Operation"
    mode="operation">
  <fo:block line-height="16pt" font-size="14pt"
     text-align="left" color="orange" start-
     indent="30pt">
    <xsl:if test="@name
```

```
          != current()/../../@name">
        <xsl:value-of select="@visibility"/>
        <xsl:text>&#x20;</xsl:text>
        <xsl:apply-templates select="descendant::
            UML:Parameter[@kind='return']"/>
        <xsl:text>&#x20;</xsl:text>
        <xsl:value-of select="@name"/>(
          <xsl:apply-templates
              select="descendant::
              UML:Parameter[@kind='in']|
              descendant::
              UML:Parameter[@kind='inout']"
              mode="parameter"/>);
      </xsl:if>
      <xsl:if
          test="@name = current()/../../@name">
        <xsl:value-of select="@visibility"/>
        <xsl:text>&#x20;</xsl:text>
        <xsl:value-of select="@name"/>(
        <xsl:apply-templates select="descendant::
            UML:Parameter[@kind='in']|
            descendant::UML:Parameter
            [@kind='inout']" mode="parameter"/>);
      </xsl:if>
    </fo:block>
  </xsl:template>

  <xsl:template match="UML:Parameter[@kind='in']
      |UML:Parameter[@kind='inout']"
      mode="parameter">
    <xsl:value-of select="id(@type)/@name"/>
    <xsl:text>&#x20;</xsl:text>
    <xsl:value-of select="@name"/>
    <xsl:if test="not (position()=last())">,
    </xsl:if>
  </xsl:template>

  <xsl:template
      match="UML:Parameter[@kind='return']"
      mode="parameter">
    <xsl:value-of select="id(@type)/@name"/>
  </xsl:template>

  <xsl:template match="text()"/>
</xsl:stylesheet>
```

The PDF document after FO to PDF translation



```
                    Package : Pack
public class Bank{
// Associations
        public Person customer(Integer accountNumber);
}
public class Company{
// Attributes
        private String name ;
        private Integer numberOfEmployees ;
// Operations
        public stockPrice();
        public Company();
        public emploie(Person p, Person q);
        public demissionne(Person p);
// Associations
        public Person manager;
        public Person employee[];
}
public class Person{
// Attributes
        private Boolean isMarried ;
        private Boolean isUnemployed ;
        private java.util.Date birthDate ;
        private Integer age ;
        private String firstName ;
        private String lastName ;
// Operations
        public income();
// Associations
        public Company managedCompany[];
        public Company employer[];
}
public class Femme extends Person{
// Associations
        public Homme husband;
}
public class Homme extends Person{
// Associations
        public Femme wife;
}
```

AUTHORS

Javier Ballesteros Rodríguez

Javier Ballesteros Rodríguez is a Telecommunication Engineer. He graduated from the Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona (Universitat Politècnica de Catalunya). He started his professional career as a software developer, within the area of management software, before joining GTD in January 2001. At GTD, he worked on several projects in the area of Information Systems, Defence, Space and Secure Communication Systems.

Pierre Bazex

Pierre Bazex is professor at the Paul Sabatier University. Pierre Bazex manages research team on data base modelling and programming methods for software development. Pierre Bazex and his research team are working in collaboration with Toulouse societies (Aérospatiale/EADS, MATRA/EADS, CNES/ESA, and software maintenance societies like CS-SI and Telelogic).

Jean-Paul Bodeveix

Jean-Paul Bodeveix is assistant professor of computer science at University Paul Sabatier - Toulouse. His main research interests are linked to formal specification languages, formal development methods and verification techniques. His teaching domains cover these aspects as well as object oriented design and languages.

Agusti Canals

Agusti Canals is a software engineer (Université Paul SABATIER, Toulouse) and has been working at CS since 1981. Now project manager and senior software engineering consultant, he has already presented papers on HOOD, Ada, UML and object business patterns. He also currently teaches software engineering in different training structures, like Ecole Centrale Paris or Université Paul Sabatier (Toulouse).

Juan Carlos Cruellas

Juan Carlos Cruellas received his Ph.D. from Universitat Politecnica de Catalunya in 1990. He is professor at this University in the Computer Engineering and Electronic Engineering Schools, where he teaches software engineering. His main research topics are computers, networks security and process modelling.

Louis Féraud

Louis Féraud is professor of computer science at University Paul Sabatier - Toulouse since 1991. For nine years, he has been in charge of the master degree of computer science of that university. His domains of interests are covering formal semantics, compilation and object oriented programming as well. Louis Féraud is currently embedded in several European and national research projects aiming at applying compilation theories and methods to object oriented modelling problems.

Christophe Le Camus

Christophe Le Camus is a Conservatoire National Arts & Métiers (CNAM) computer engineer. He has worked in software editor industry like software trainer for administrative client in France (council up to 10000 citizen and Centre de Gestion) during seven years. He participates in analysis and programming of an accounting software based on administrative rules distributed to 2000 clients. Activities on Neptune project concerned about OMG activities (modeling with UML, MOF and XMI) and W3C (XML, XSL).

Josep Maria Llovet Pérez

Josep Maria Llovet Pérez has a Degree in Mathematics. He is a Mathematics High School Senior Professor (Catedràtic). He has also been a senior consultant in Information Systems for more than twenty years.

Thierry Millan

Thierry Millan received his Ph.D. from the Paul Sabatier University of Toulouse in 1995. Thierry Millan is a teacher of UML at the Paul Sabatier University and a researcher at the IRIT institute. His main research topics are *design methodology*, *persistence* and *object-oriented languages* (Java, C++, and Ada 95.

Chritian Percebois

Christian Percebois is Professor of computer science at the University of Toulouse III, France since 1992. He worked on Lisp and Prolog interpreters, garbage collecting for symbolic computations, asynchronous backtrackable communications in distributed logic languages, abstract machine construction through operational semantics refinements and typing in object-oriented programming. Since 1996, he is mainlly interested by multiset rewriting techniques in order to coordinate concurrent objects.

Laurent Pomiès

Laurent Pomiès is a software engineer. He graduated from the Ecole Nationale Supérieure de Physique de Marseille (1997). He spent three years in automotive industry (VDO CC, Siemens automotive), as an embedded software developer and architect before joining CS in august 2001.

REFERENCES

[ref1]
UN/CEFACT Techniques and Methodologies Group (TMG):
"UN/CEFACT Modelling Methodology". November 2001.

[ref2]
Eriksson, Hans-Erik and Penker, Magnus: "Business
Modeling with UML, Business Patterns at Work".
USA, Wiley Computer Publishing, John Wiley & Sons, Inc, 2000

[ref3]
Darnton, Geoffrey and Moksha Darnton: "Business Process Analysis".
Cambridge, U.K.: Thomson
Business Press, 1997

[ref4]
Emerson, E. A. Temporal and modal logic. In: J. van Leeuwen (Ed.), "Handbook of Theor. Comput. Sci.", Elsevier Sci. Publishers, 1990.

[ref5]
Jos Warmer
Document -- ad/02-05-09 (UML 2.0 OCL RFP revised submission)
http://www.omg.org/cgi-bin/doc?ad/02-05-09

[ref6]
"Guide Business Rules Project: Final Report" Hay, D., Allan Kolber and Keri Anderson Healy, 1995

[ref7]
Complete UML 1.4 specification
http://www.omg.org/cgi-bin/doc?formal/01-09-67
Document -- formal/01-09-67 (Unified Modeling Language, v1.4)
Contact: Ms. Linda Heaton

[ref8]
http://www.omg.org/cgi-bin/doc?formal/00-11-02
Document -- formal/00-11-02 (XML Metadata Interchange (XMI) version 1.1)
Contact: Ms. Linda Heaton

[ref9]
"Design schemes in space application"
A.Canals (Cisi), Ja.Veron et Jc.Lloret (CNES); GL'98, Paris

[ref10]
« Use of UML/CS SI development process »
A. Canals (CS SI) ; DASIA'99 Lisbon, ICSEA'99 Paris and JOOP (April'2001)

[ref11]
« How you could use NEPTUNE technology in the modelling process »
Agusti Canals, Yannick Cassaing, Antoine Jammes, Laurent Pomiès, Etienne Roblet ; DASIA’02 Dublin, JOT january2003

[ref12]
« The UML Activity Diagrams: Using examples through the method used by CS » by Agusti Canals, Yannick Cassaing, Antoine Jammes ICSSEA01 Paris

[ref13]
« UML and Architectures: Using the MERCURE and NEPTUNE results »
Agusti Canals and Yannick Cassaing ICSSEA02 Paris

[ref14]
Business Component-Based Software Engineering, chapter 12
Edited by F Barbier (KLUWER), 2003

[ref15]
HOOD, An industrial approach for software design, JP Rosen Edited by HOOD User’s group, 1997

[ref16]
Modélisation objet avec UML
PA Muller and N Gaertner, edited by Eyrolles 2000

[ref17]
Design Patterns
E Gamma, R Helm, R Johnson, J Vlissides, Edited by Thomson publishing, 1996

[ref18]
The unified software development process
I Jacobson, G Booch, J Rumbaugh, edited by Addison-Wesley, 1999

[ref19]
The unified modeling language
I Jacobson, G Booch, J Rumbaugh, edited by Addison-Wesley, 1998

May 2003

MESSAGES SA – TOULOUSE
111, rue Nicolas-Vauquelin
Tél. : 05 61 41 24 14