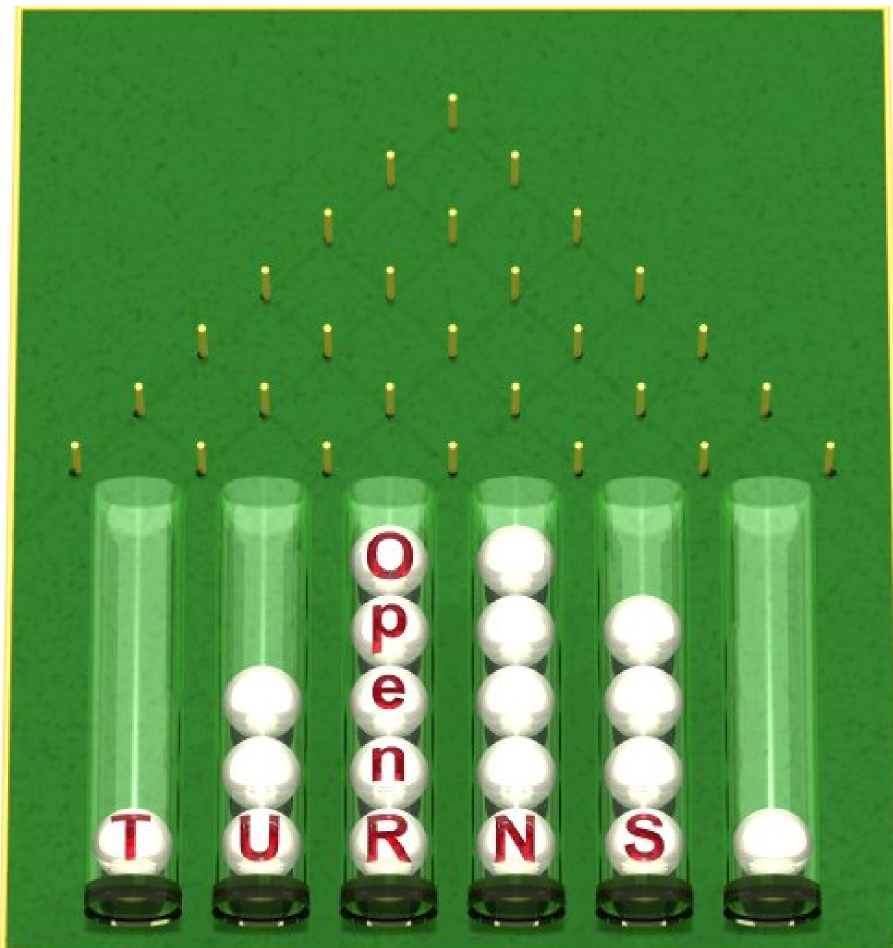
First Elements of the Architecture Guide

Open TURNS version 1.0

Documentation built from package openturns-doc-April2012

April 20, 2012



Open TURNS project: General architecture specifications

Abstract

The Open TURNS project is an open source development project. It aims at developing a computational platform designed to carry out industrial studies on uncertainty processing and risk analysis.
This platform is intended to be released as an Open Source contribution to a wide audience whose technical skills are very diverse. Another goal of the project is to make the community of users ultimately responsible for the platform and its evolution by contributing to its maintenance and developing new functions.
This architecture specifications document therefore serves two purposes:

- to provide the design principles that govern the platform, in order to guide the development teams in their development process;
- to inform external users about the platform's architecture and its design, in order to facilitate their first steps with the platform.

In the first section of this document, we will introduce the concepts that governed the construction of the platform. These concepts resulted from the requirements analysis carried out with the users and the developers, following the UML approach. The general functions of the platform allowed us to categorize the concepts by giving us a more synthetic and global view of its components.
The second section details the technical choices that were retained for the platform's development.

©2005-2012 EDF - EADS - PhiMeca

Contents

1 Introduction **13**
1.1 Document outline . . . 14
1.2 Bibliography . . . 14
1.2.1 Modelling . . . 14
1.2.2 C++ and STL . . . 14
1.2.3 Multithreading . . . 14
1.2.4 Python . . . 14
1.2.5 Qt . . . 14
1.2.6 Subversion . . . 14
1.2.7 Websites . . . 14

2 Analysis model and functional architecture **15**
2.1 Description of the analysis approach . . . 15
2.2 Requirements analysis and use cases . . . 16
2.2.1 Definition of the actors involved with the Open TURNS platform . . . 17
2.2.2 Modeling package . . . 17
Creating a numerical sample . . . 18
Creating a distribution . . . 18
Creating a random vector . . . 19
Creating a failure event . . . 19
Creating a numerical function . . . 19
2.2.3 Propagation Package . . . 19
Generating a numerical sample . . . 19
Statistical computations on a random vector . . . 20
Computing the moments using SRSS . . . 20
2.2.4 Prioritization Package . . . 20
Regression test case prioritization . . . 20
2.2.5 Contribution package . . . 21
Integrating a new external code . . . 22
Integrating a new distribution . . . 22
Integrating a new algorithm . . . 22
2.2.6 Configuration package . . . 22
External code configuration . . . 23
Computer configuration . . . 23
2.3 Description of the analysis concepts . . . 23
2.3.1 Notations used in the analysis model . . . 23
Concept . . . 23

Association 24
Unidirectional association 24
Generalization/Specialization 24
Aggregation 24
Composition 24
Multiplicity of associations 24
Role within an association 25
Association's name 25
2.3.2 Basic concepts 25
Integer 25
NumericalScalar 25
String 26
FileName 26
NumericalPoint 26
Description 26
NumericalSample 26
Matrix 26
2.3.3 Uncertainty concepts 27
RandomVector 27
FailureEvent 28
Distribution 28
UsualDistribution 28
AssemblyDistribution 28
WeightedDistribution 29
MixtureDistribution 29
FunctionalDistribution 29
DistributionFactory 30
Kernel 31
2.3.4 Function concepts 31
NumericalFunction 31
Copula 31
2.3.5 Algorithm concepts 32
SimulationAlgorithm 32
2.4 Synopsis of a functional architecture 32
2.4.1 Packages 32
2.4.2 Layers 33
2.4.3 General diagram 33

3 Design model and software architecture **37**
3.1 Notation and glossary 37
3.1.1 Class 37
3.1.2 Object 38
3.1.3 Attribute 38
3.1.4 Method 38
3.1.5 Visibility 38
3.1.6 Scope 38
3.1.7 Abstract class and method 38
3.1.8 Method call 39

©2005-2012 EDF - EADS - PhiMeca

3.2 Design patterns . . . 39
3.2.1 Introduction . . . 39
3.2.2 Singleton pattern . . . 39
3.2.3 Factory pattern . . . 40
3.2.4 Strategy pattern . . . 40
3.2.5 Composite pattern . . . 41
3.2.6 Iterator pattern . . . 41
3.2.7 Visitor pattern . . . 42
3.3 Package description . . . 43
3.3.1 Basis brick . . . 43
Data types . . . 44
Id . . . 44
NumericalScalar . . . 44
String . . . 44
FileName . . . 44
Bool . . . 44
UnsignedLong . . . 44
LibraryHandle . . . 44
LibrarySymbol . . . 44
Common classes . . . 45
Object . . . 45
PersistentObject . . . 45
IdFactory . . . 46
Exception . . . 46
FileNotFoundException . . . 46
InvalidArgumentException . . . 46
NoWrapperFileFoundException . . . 46
WrapperFileParsingException . . . 46
WrapperInternalException . . . 47
XMLException . . . 47
XMLParserException . . . 47
DynamicLibraryException . . . 47
Iterator . . . 47
Pointer . . . 47
BoostPointerImplementation . . . 48
Thread . . . 48
ThreadStatus . . . 48
Threadable . . . 49
Lockable / Lock . . . 49
Path . . . 49
StorageManager . . . 49
WrapperFile . . . 49
WrapperData . . . 49
XMLStringConverter . . . 50
XMLWrapperErrorHandler . . . 50
Base type . . . 50
Collection . . . 50
NumericalPoint . . . 50

©2005-2012 EDF - EADS - PhiMeca

ZeroPoint . . . 51
Description . . . 51
Matrix . . . 51
SquareMatrix . . . 52
SymmetricMatrix . . . 52
CorrelationMatrix . . . 52
FictiveCorrelationMatrix . . . 52
KendallCorrelationMatrix . . . 52
SpearmanCorrelationMatrix . . . 53
Tensor . . . 53
FamilyPolicy . . . 53
NormalFamily . . . 54
Basic numerical functions . . . 54
NumericalMathFunction . . . 54
AnalyticalNumericalMathFunction . . . 54
TabulatedNumericalMathFunction . . . 54
ComputedNumericalMathFunction . . . 54
ScriptedNumericalMathFunction . . . 55
CompositeNumericalMathFunction . . . 55
MathFunctionOperator . . . 55
MathFunctionAddition . . . 55
MathFunctionSubstraction . . . 56
MathFunctionMultiplication . . . 56
MathFunctionDivision . . . 56
MathFunctionComposition . . . 56
Library . . . 56
LibraryLoader . . . 56
NumericalWrapperFunction . . . 56
UsualNumericalWrapperFunction . . . 57
Statistical objects . . . 57
CovarianceMatrix . . . 57
IdentityMatrix . . . 57
NumericalSample . . . 57
ConcreteNumericalSample . . . 58
CompositeNumericalSample . . . 58
NumericalSampleFactory . . . 58
CSVNumericalSampleFactory . . . 58
3.3.2 Field-specific brick . . . 58
Data modeling . . . 59
Distribution . . . 59
UsualDistribution . . . 59
AssemblyDistribution . . . 60
Mixture . . . 60
FunctionalDistribution . . . 61
DistributionCollection . . . 61
DistributionFactory . . . 61
DistributionFactoryPolicy . . . 62
Kernel . . . 62

©2005-2012 EDF - EADS - PhiMeca

GenericRandomVector . . . 63
RandomVector . . . 63
ConstantRandomVector . . . 63
CompositeRandomVector . . . 63
Copula . . . 63
NormalCopula . . . 64
FailureEvent . . . 64
ComparisonOperator . . . 65
Less . . . 65
LessOrEqual . . . 65
Equal . . . 65
GreaterOrEqual . . . 65
Greater . . . 65
MaximumLikelihood . . . 66
MomentMethod . . . 66
RandomVectorIterator . . . 66
OptimalWalkIterator . . . 67
RandomVectorVisitor . . . 67
OptimalWalkVisitor . . . 67
PropagationVisitor . . . 67
Distributions . . . 67
XDistribution . . . 68
XDistributionFactory . . . 68
XDistributionKernel . . . 68
Simulation algorithms . . . 68
UncertaintyAlgorithm . . . 68
Result . . . 69
MonteCarlo . . . 69
FORM . . . 70
SORM_Breitung . . . 70

4 Technical architecture **71**
4.1 Target platforms . . . 71
4.2 Namespace . . . 71
4.3 Internationalization . . . 72
4.4 Accessibility . . . 72
4.5 Tools . . . 72
4.5.1 Tool evolution policy . . . 72
4.5.2 Programming conventions . . . 73
4.5.3 Version control . . . 73

©2005-2012 EDF - EADS - PhiMeca

©2005-2012 EDF - EADS - PhiMeca

List of Figures

2.1 Example of a use case triggered by a system user. 16
2.2 Modeling use cases. 18
2.3 Propagation use cases. 20
2.4 Prioritization use cases. 21
2.5 Contribution use cases. 21
2.6 Configuration use case. 22
2.7 Graphical representation of concepts from an analysis model and their relationships. 23
2.8 Basic concepts. 25
2.9 Matrix concepts. 26
2.10 The concept of random vector. 27
2.11 The concept of failure event. 28
2.12 The concept of uncertain distribution. 29
2.13 The concept of distribution family. 30
2.14 The concept of numerical function. 31
2.15 The concept of simulation algorithm. 32
2.16 General diagram. 35
2.17 Functional diagram. 36

3.1 Representing classes. 37
3.2 Collaboration diagram. 39
3.3 Singleton structure. 40
3.4 Factory structure. 40
3.5 Strategy structure. 41
3.6 Composite structure. 41
3.7 Example of tree modeled by the Composite. 42
3.8 Iterator structure. 42
3.9 Visitor structure. 43
3.10 Object, PersistentObject and IdFactory classes. 45
3.11 Exception class. 46
3.12 Pointer class. 47
3.13 Thread, Threadable and ThreadStatus classes. 48
3.14 Collection and DistributionCollection classes. 50
3.15 Collection, NumericalPoint, ZeroPoint and Description classes. 51
3.16 Collection and Description classes. 51
3.17 Collection, Matrix, SquareMatrix and SymmetricalMatrix classes. 52
3.18 SymmetricMatrix, CovarianceMatrix, IdentityMatrix, CorrelationMatrix, and related classes. . . 53
3.19 Collection and Tensor classes. 53
3.20 FamilyPolicy and NormalFamily classes. 54

3.21 NumericalMathFunction and related classes, MathFunctionOperator and related classes. 55
3.22 Collection, NumericalSample, and related classes. 57
3.23 NumericalSampleFactory, CSVNumericalSampleFactory, and NumericalSample classes. 58
3.24 Distribution, UsualDistribution, AssemblyDistribution, Mixture, and FunctionalDistribution classes. 59
3.25 UsualDistribution and related classes. 60
3.26 AssemblyDistribution, DistributionCollection and Copula classes. 61
3.27 Mixture and DistributionCollection classes. 61
3.28 FunctionalDistribution and NumericalMathFunction classes. 61
3.29 DistributionFactory and related classes. 62
3.30 DistributionFactoryPolicy, MomentMethod, and MaximumLikelihood classes. 63
3.31 Kernel and related classes. 64
3.32 GenericRandomVector and related classes. 65
3.33 Copula and NormalCopula classes. 65
3.34 FailureEvent class. 66
3.35 Iterator, RandomVectorIterator and GenericRandomVector classes. 66
3.36 RandomVectorVisitor, PropagationVisitor and OptimalWalkVisitor classes. 67
3.37 UncertaintyAlgorithm, Result, Threadable and related classes. 69
3.38 UncertaintyAlgorithm and MonteCarlo classes. 69
3.39 UncertaintyAlgorithm and FORM classes. 70
3.40 UncertaintyAlgorithm and SORM_Breitung classes. 70

©2005-2012 EDF - EADS - PhiMeca

List of Tables

3.1 Mapping between distributions and classes 68

4.1 Examples of Linux distributions supported by the project's partners 71
4.2 Software development tools 72

©2005-2012 EDF - EADS - PhiMeca

Chapter 1

Introduction

This document makes up the general specifications for the architecture of the Open TURNS platform. The architecture described here adresses the following goals:

- *building an open, upgradable and generic platform for the treatment of uncertainties*, relying on recognized and valid mathematical methods as well as on a methodological approach that was put forward and supported by the partners.

- *interfacing with any field-specific code.*

To address these questions, the Open TURNS platform needs to be:

- *portable*: the ability to build, execute and validate the application in different environments (operating system, hardware platform as well as software environment) based on a single set of source code files.

- *extensible*: the possibility to add new functions to the application with a minimal impact on the existing code.

- *upgradable*: the ability to control the impact of a replacement or a change on the technical architecture, following an updgrade of the technical infrastructure (such as the replacement of one tool by another or the use of a new storage format).

- *durable*: the technical choices must have a lifespan comparable to the application's while relying on standard and/or Open Source solutions.

The computing architecture will be detailed in this document according to 3 different abstraction levels:

- *functional architecture*: describes the application's functions and the distribution of its components without addressing any actual programming issues.

- *software architecture*: defines the programming design of the modules identified in the functional architecture.

- *technical architecture*: lists the technical choices that were made for the development of each module. These technical choices are justified both by environmental constraints and by the requirements expressed in other archectural levels.

1.1 Document outline

The document is organized as follows:

- *Chapter 2* deals with the functional architecture of the Open TURNS platform (module distribution and description of each brick's function).
- *Chapter 3* defines the technical architecture of the Open TURNS platform (technical choices).
- The *bibliography* gives an alphabetical list of all references used in this document.

1.2 Bibliography

The following subsections organize the bibliographical references used in this document according to the main fields at stake. For more details about each reference, please refer to the bibliography at the end of this document.

1.2.1 Modelling

[Mul] [GHJV]

1.2.2 C++ and STL

[ES] [Del] [Meya] [Meyc] [Ale] [Aus] [Meyb]

1.2.3 Multithreading

[LB]

1.2.4 Python

[LA] [Lut] [PY]

1.2.5 Qt

[Dal]

1.2.6 Subversion

[CSFP] [SVN]

1.2.7 Websites

[SWI] [BOO] [OT] [OTD] [UNI] [R]

©2005-2012 EDF - EADS - PhiMeca

Chapter 2

Analysis model and functional architecture

This section deals with the analysis of the requirements that guided the general implementation choices for the Open TURNS platform. The system is here to be seen from a rather generic, functional point of view; it mainly consists in blocks handling the main functions of the platform. We shall detail the fundamental concepts that arise from the requirements and their role within the system.
At this stage of modeling, technical considerations shall not (except in specific cases) be addressed yet. The platform is seen as a system made up of components interacting with each other, providing designated functions. In this section, we shall answer the question "What does the system actually perform ?"

2.1 Description of the analysis approach

The Open TURNS project aims at producing a risk analysis tool. When the project started, there was no equivalent tool available on the market, whether it be proprietary or Open Source software. Therefore, a choice was made to design an entirely new tool "from scratch", beginning with the requirements analysis, using the most effective methods and the most up-to-date scientific concepts, relying on tools widely recognized by the scientific community.
Considering the recent progress in computer science and software engineering, the platform will be implemented using an object-oriented language, which naturally leads us to use an object modeling approach to analyze and design the elements that will make up the code. The UML approach was therefore chosen for this preliminary stage of the project.
Readers interested in UML will find more detailed information in the guide written by Pierre-Alain Muller [Mul]. However, we will sum up the approach to present all the notions relevant to this document.
UML relies on the analysis of the requirements expressed by the future platform users, using a standard graphical formalism. Everything starts with the definition of use-cases that describe how the users will interact with the system. There can be several users (sometimes several thousands) but only their role regarding the system is taken into consideration. These users are supposed to describe the actions they want to perform and the responses they expect from the system. Therefore, at this stage of modeling, the actual realization ("how") of actions is not taken into account; only the functional need of the user ("what") is being considered.
First, these use cases are graphically modeled with ellipses, the users being represented as human figures (as shown in Figure 2.1).
All of the use cases triggered by the same user allow us to have an overview of the actions this user can carry out on the system.
Conversely, requiring all the users to describe the same use case allows us to study the case from multiple angles and to determine all the interactions that need to be taken into account between users.
Naturally, the definition of a use case is not limited to putting a designation within an ellipse. This description

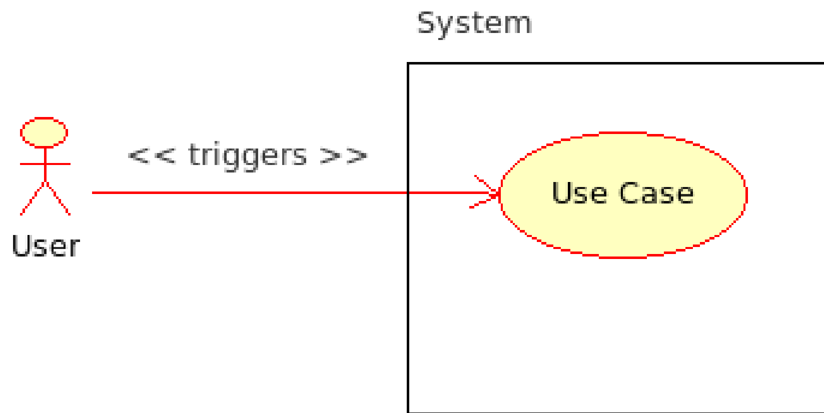


Figure 2.1: Example of a use case triggered by a system user.

is only used to provide an overview of the use case. The use case must be described in detail: most often, this stage uses a natural language (i.e. free format) description of the user's actions and the system's responses. If this description requires highly technical content (which is the case for the Open TURNS platform), it may be wiser to use a language close to either the user's requirements or the target system. The appendix provides an extensive description of the use cases (in algorithmic form) written for the Open TURNS platform.
Based on these detailed descriptions, we can now extract concepts that are fundamental for the system and link them so as to view their interactions, their functional proximity and the abstract ideas that can be derived from them. We refer to this set of concepts and links as an *analysis model*.
It is then possible, based on these concepts, to create bricks of variable size encompassing related notions, and thus to show the logical and functional view of the platform. This global view of the platform is essential to have a general understanding of the system as well as to consider its evolution capabilities.
The present chapter describes the results of this analysis, which was carried out with and for the platform users. In the next chapter, we will rely on this analysis to introduce the design model and the software architecture.

2.2 Requirements analysis and use cases

Interviews with the future users of the system have uncovered the following general requirements:

- the platform must allow the users to carry out uncertainty and risk analysis studies as well as statistically process data provided both internally and externally;
- the platform must be of ergonomic and easy use for novice users, as well as complete and precise for expert users. It must therefore follow the Methodological Reference [Dut] provided by EDF R&D;
- the platform must provide a graphical user interface (GUI) as well as a text user interface (TUI); it must also provide a means of external control by another code (*scripting* and *batch*);
- the platform must be efficient in order to handle several millions (or more) of computations, and must be able to use the distributed computing capabilities of an heterogenous network or a remote computer;
- the platform must run in a Unix-like environment, but future ports to other environments (Windows, etc.) will be considered;
- the platform is generic and must be able to interface with (almost) any field specific code;

©2005-2012 EDF - EADS - PhiMeca

- the platform is developed using Open Source software and will be distributed under the terms of an Open Source license;
- the platform includes efficient scientific methods that can be expanded or supplemented;
- the platform uses Open Source tools that are considered as references in the field of statistics and optimization;
- the platform will be primarily developed using object-oriented technologies;
- the platform must be able to run either as a *standalone* application or as an extension to another application;
- the platform must interface with an indexed storage system that allows the preservation of results coming from previous computations, so as to avoid having to compute them again;
- the platform must include an extensive online documentation.

These general requirements are described with more details in the use cases, which are grouped in packages according to the recommendations given in the Methodological Reference.

2.2.1 Definition of the actors involved with the Open TURNS platform

The analysis of the actors involved with the Open TURNS platform brings us to the following list:

- *User*: this actor carries out the uncertainty treatment studies with the platform. From a practical point of view, it is the main actor which we will focus on, since the platform is being developed with that role in mind. It is in most cases a human being that will be interacting with the system, but other systems can also play that role in automated running modes (*scripting* and *batch*).
- *Developer*: this actor adds new functions to the Open TURNS platform. Although the platform does not focus on this actor, it is implemented so as to make the development and integration tasks as easy as possible for them. It is always a human being.
- *Administrator*: this actor sets up the Open TURNS platform on the target network. Their role is to configure the platform so that it provides the expected services. It is always a human being.
- *External code*: this actor is the field specific code that will be called upon in an uncertainty study. From the system's point of view, it is a passive actor.
- *Storage system*: this actor is the system that allows the preservation of results coming from previous studies. From the system's point of view, it is also a passive actor.

2.2.2 Modeling package

This package gathers all the use cases involved in defining an uncertainty treatment study before any computation is actually carried out by the system. Figure 2.2 details these use cases.
The only actor involved in this package is the *user*.
For the detailed use cases, please refer to the appendix.

©2005-2012 EDF - EADS - PhiMeca

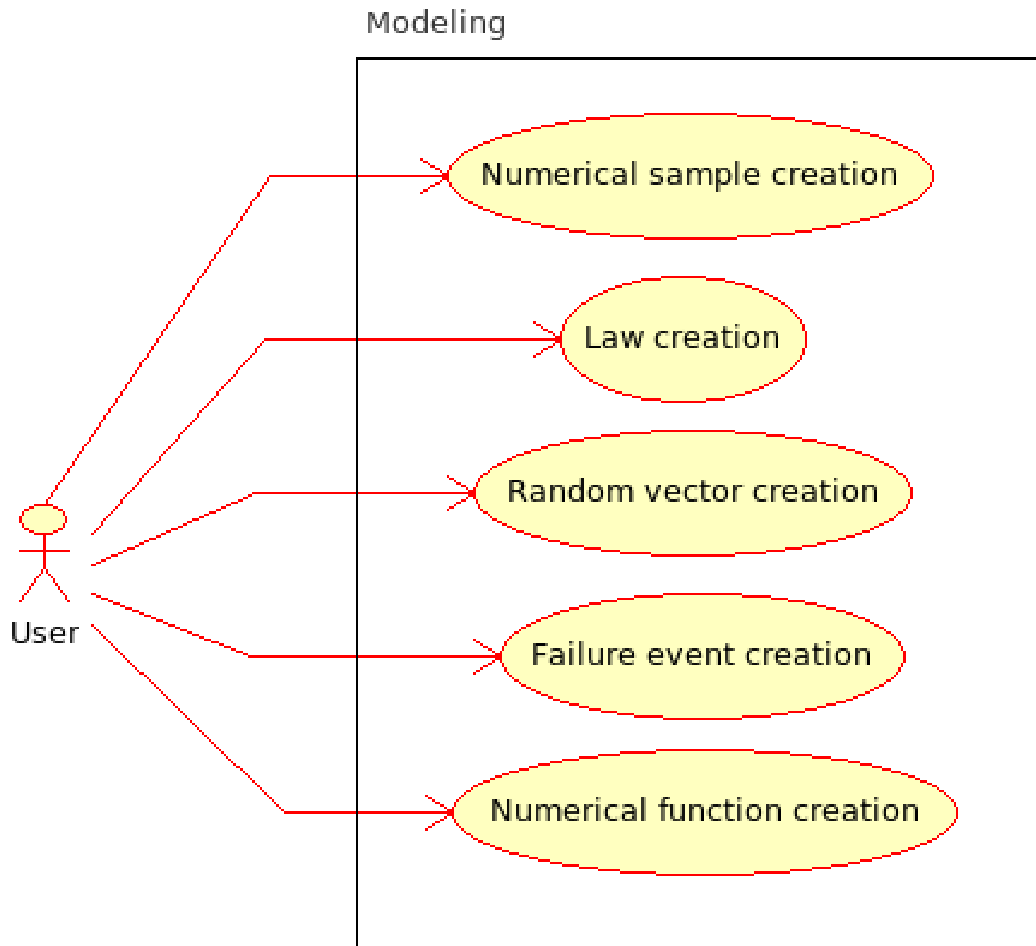


Figure 2.2: Modeling use cases.

Creating a numerical sample

The numerical sample is a key concept for the Open TURNS platform, particularly with regards to its statistical capabilities. The user can create a sample in the system in different ways:

- by importing a file containing the description of the sample;
- by manually setting the values of the sample through the interface;
- by using a random variable to produce the sample;
- by extracting a subsample from a given sample or expanding[1] a given sample into a larger one.

Creating a distribution

The distribution is probably the core concept of the Open TURNS platform. The uncertainty treatment model is, for the most part, based on the notion of distribution. A distribution can be created in different ways:

- by setting its parametric type and parameters through the interface;

[1] In the case of the non parametric bootstrap, elements of a sample are randomly drawn to create a larger sized sample.

©2005-2012 EDF - EADS - PhiMeca

- by automatically establishing its parameters based on a numerical sample and a hypothesis on its type (distribution inference);
- by combining other distributions (mixture distribution, assembly distribution);
- by extracting a sub-section of a given distribution;
- by creating the distribution from other distributions and numerical functions.

Creating a random vector

The random vector represents the concept of random variable within the Open TURNS platform. It is the concept most easily manipulated by users in their studies. It can be created as follows:

- by setting its joint distribution;
- by combining several other random vectors;
- by extracting a sub-vector from a given random vector, or expanding a given random vector into a larger one;
- by creating the vector from other random vectors and numerical functions.

Creating a failure event

A failure event is used to identify the limit state of a numerical function. It can be created very easily with a random vector, a threshold (in the form of a point) and a comparison operator that will compare the value of the vector with the one of the threshold.

Creating a numerical function

The numerical function is, along with the distribution, another core concept of the Open TURNS platform. The numerical function represents the physical or theoretical model that is to be studied and through which the uncertainties on the input random vector will be propagated. This numerical function can be given in an analytical form or in a coded form (in which case it is defined within a external code).

2.2.3 Propagation Package

This package encompasses all the use cases that appear after modeling, when the user wants to propagate uncertainty through the external code. Figure 2.3 details these use cases.
In all use cases from this Propagation package, two new actors appear beside the user: the external code and the storage system. From the system's point of view, the external code behaves as a numerical function on which an uncertainty study is to be carried out. The storage system, which is important but optional, acts as a replacement for the external code during the (costly) evaluation of the numerical function.
The use cases are detailed in the appendix.

Generating a numerical sample

Generating a numerical sample means applying the numerical function on the input numerical sample. The generated sample is the numerical sample produced as an output of the numerical function. The samples can reach very large sizes (several million points) and therefore, the evaluation of the numerical function needs optimization. This can be achieved either by computing only the points that have not been computed yet,

©2005-2012 EDF - EADS - PhiMeca

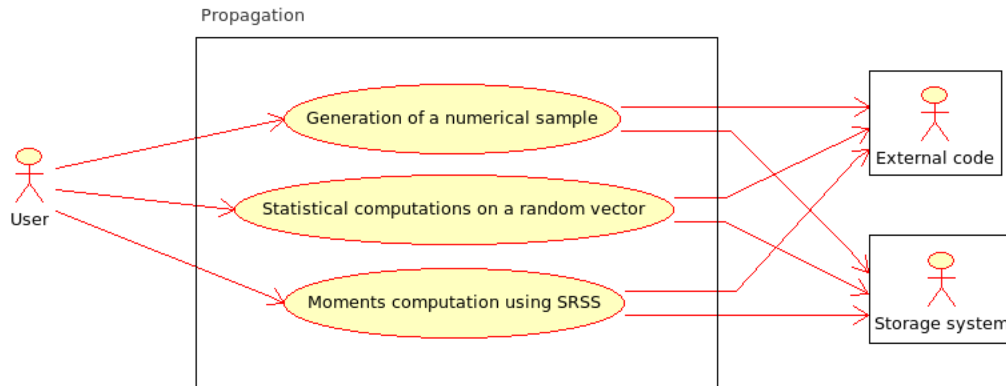


Figure 2.3: Propagation use cases.

which makes use of the storage system; or by distributing the external code's execution on a computer network. The propagation of uncertainty based on Monte-Carlo methods belongs to this application field.
The distributed computing system being integrated in the Open TURNS platform, it was decided not to represent it as an external actor.

Statistical computations on a random vector

The goal of an uncertainty processing study is to evaluate a given number of statistics on a random vector, which is generally defined as the output of a numerical function. The nature of these statistics may vary: mean value, standard deviation, threshold-crossing probability, and so on.
Depending on the statistics chosen, different algorithms can be implemented, which influences the number of calls on the numerical function and its derivative functions. As a direct corollary, the external code can be called a very large number of times, which results in an important computational overload that needs to be distributed throughout the network in order to reduce the waiting time for the user: an example of this would be the computation of the numerical function's gradient using the finite-difference method.
A storage system that is external to the platform and works as a cache can help reduce the number of computations by preserving the already computed results.

Computing the moments using SRSS

Computing moments using SRSS involves evaluating the numerical function, its gradient and its higher order derivatives. As for the computation of a threshold-crossing probability, it is sometimes necessary to evaluate the numerical function's derivatives using a finite difference algorithm, which requires the parallelization of the computations and the use of a storage system to preserve previous results.

2.2.4 Prioritization Package

The prioritization is the last stage in the uncertainty processing procedure, which evaluates the sensitivity of the output with respect to the input parameters. It comes after the propagation stage (see the previous section).
The use cases are detailed in the appendix.

Regression test case prioritization

Prioritization uses a linear regression computation between the input and output parameters of the numerical function evaluated during the propagation stage, so as to assess the sensitivity of the output parameters re-

©2005-2012 EDF - EADS - PhiMeca

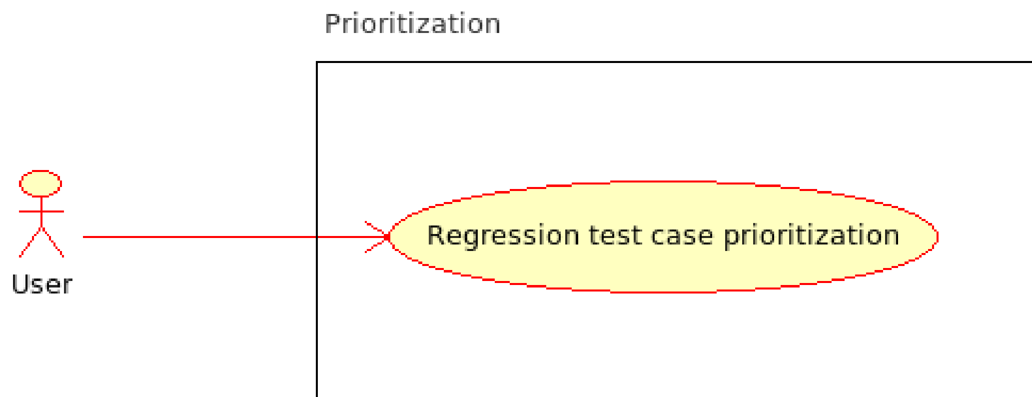


Figure 2.4: Prioritization use cases.

garding the input. This stage usually makes use of the platform's graphical abilities in order to visualize the results.

2.2.5 Contribution package

This package encompasses the use cases dealing with the development and integration of new functionalities into the Open TURNS platform.

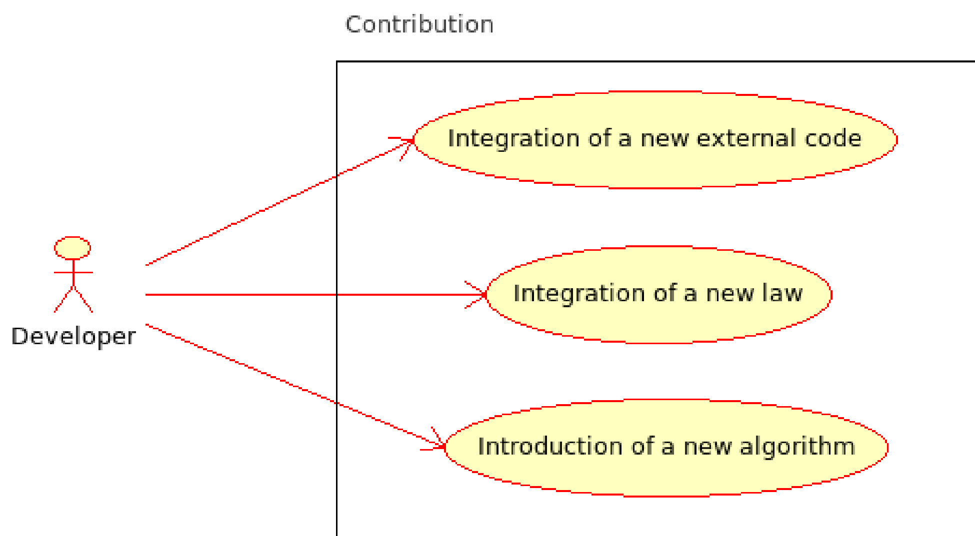


Figure 2.5: Contribution use cases.

As shown in Figure 2.5, a new actor, the developer, is involved and can be competent in one or several of the following fields:

- Computer science
- Numerical analysis
- Statistics/probabilities

©2005-2012 EDF - EADS - PhiMeca

Several persons may be needed to combine all of the above skills. Integrating new functions into the Open TURNS platform may require the collaboration of several people.
These use cases are not detailed in this document; they will however be introduced as documented examples provided with the platform distribution.

Integrating a new external code

The external code is the passive actor that supports the notion of numerical function; integrating new functions into the platform therefore requires interfacing with a field-specific code, using an API that the code must follow. If it is not natively the case, an interface layer between the API and the code needs to be developed. This layer is not the responsibility of the Open TURNS platform. The platform only defines the API that enables communication with the external code.

Integrating a new distribution

Integrating new distributions in the platform enhances its modeling capabilities. The standard distributions included in the platform are described in the project's Methodological Reference [OTmeth].

Integrating a new algorithm

Integrating new algorithms in the platform enhances its modeling and computing capabilities. The standard algorithms included in the platform are described in the project's Methodological Reference [OTmeth].

2.2.6 Configuration package

This package encompasses the use cases related to the platform's setup on a computer park for a field-specific use.

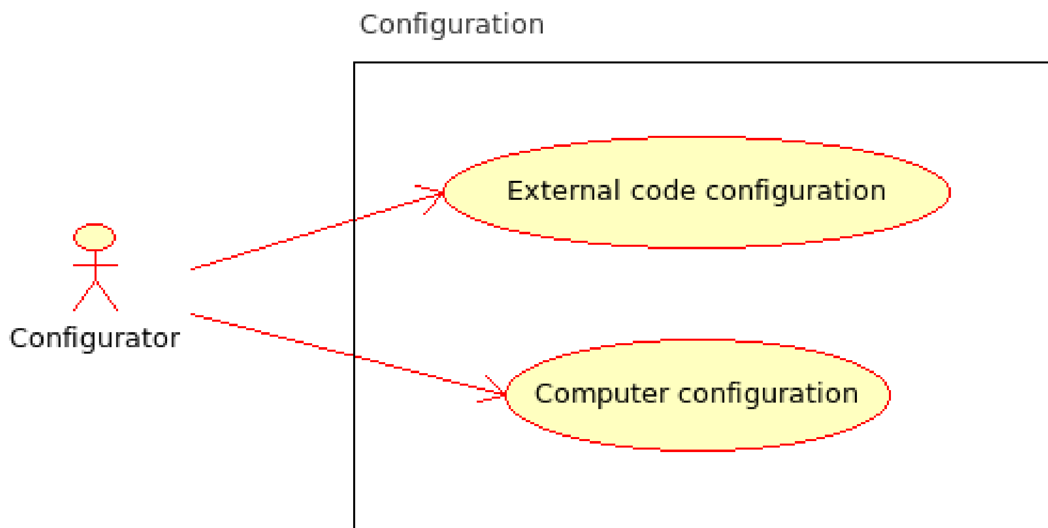


Figure 2.6: Configuration use case.

The configurator (shown in Figure 2.6) acts as the person responsible for the installation and administration of the platform. They set up the connection between the external codes and Open TURNS, and configure the platform's networking parameters. It is a static configuration: there is no automatic exploration of the computing environment.

©2005-2012 EDF - EADS - PhiMeca

These use cases are not detailed in this document; they will however be introduced as documented examples provided with the platform distribution.

External code configuration

Configuring external codes requires the configurator to define which data sets of the external code are related to the current uncertainty study, to state in the platform which external code is to be used and which of its parameters are considered uncertain.

Computer configuration

The computers' configuration is carried out by the configurator; a configuration states which computing resources are available for the execution of the platform and of its dependencies (external code, storage system, and so on), as well as the protocols, identifiers, priorities, et. related to these resources.

2.3 Description of the analysis concepts

Writing the use cases in a detailed form allows us to shed light on the concepts the user wishes to manipulate through the platform. These concepts are linked with one another by relationships such as "depends", "uses", "combines" ("aggregates", "composes"), "generalizes", "specializes", and so on. The following section shows the UML notational system (for more information, please refer to [Mul]) that was used, in order to make the graphs easier to understand.
In the analysis model, these concepts are abstract entities that do not necessarily have a direct projection as a programming entity (object, class,...). Their role is to clear the vision one can have of the system and of its internal mechanism. No technical considerations should normally appear at this stage of modeling. However, a technical consideration may exceptionally be taken into account at the analysis level if it should have nefarious consequences on the model.

2.3.1 Notations used in the analysis model

This section briefly describes the visual representation of the concepts and of the relationships that link them.

Concept

The concept is an entity manipulated by the system's user, who has an intellectual representation of the concept. Its symbol is a rectangle containing the name of the concept. Figure 2.7 shows six concepts A, B, C, D, E, and F, linked with one another.

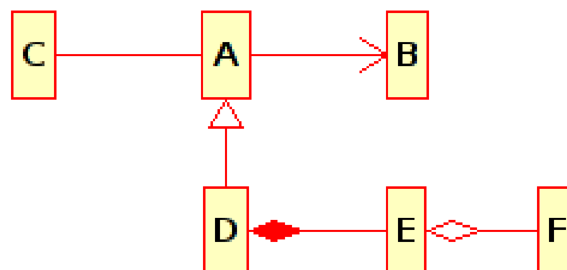


Figure 2.7: Graphical representation of concepts from an analysis model and their relationships.

©2005-2012 EDF - EADS - PhiMeca

Association

Association defines a relationship of reciprocal use between two concepts. In the example given in Figure 2.7, the concepts A and C are linked by an association. Associations are represented by full lines linking the concept boxes. Association is bidirectional: A knows C and C knows A.
Example: consider the concepts Vector and Distribution; we can say that a Vector may be associated to a Distribution, and vice versa.

Unidirectional association

This association type is a restriction of the more general association, which allows flow in only one direction. This is the case for concepts A and B of our example: A knows B but B does not know A.
Example: consider the concepts Vector and Sample; we can say that a Vector may be associated to a Sample (a Vector can produce a Sample) but a Sample cannot be associated to a Vector.

Generalization/Specialization

This relationship links two concepts, one of them (A) being more general than the other (D). We can also revert the reading of the link and say that one of them (D) is more specialized than the other (A). Its graphical representation is a hollow arrow pointing to the more general concept.
Example: a Distribution is more general than a UsualDistribution, whereas a UsualDistribution is more specialized than a Distribution.

Aggregation

Aggregation is the relationship that links a container concept to a contained concept. Its graphical representation is a clear diamond shape on the containing concept end of the relationship.
Example: a Sample is a collection or aggregation of Points.

Composition

Composition is a restriction of Aggregation which introduces a life cycle dependency: contained concepts exist only for the container and within the container's lifespan; conversely, the container exists only if the containees also exist. Its graphical representation is a black diamond shape on the container end of the relationship.
Example: a MixtureDistribution is a composition of WeightedDistributions, each characterized by its scalar weight.

Multiplicity of associations

Each end of the association may have a multiplicity indicating the number of instances of the concept (that is, the number of real objects belonging to this concept) simultaneously existing for each instance of the facing concept. This multiplicity can have one of the following values:

- *1*: one instance only;
- *n*: n instances (n positive integer);
- *m..n*: between m and n instances (m and n positive integers);
- *: any number of instances (0 included);
- etc.

Example: A Sample is a collection made up of any number of Points (multiplicity * regarding the Point concept).

©2005-2012 EDF - EADS - PhiMeca

Role within an association

Along with the multiplicity, each end of the association can carry a name that designates the role played by the concept regarding the concept at the other end of the association.
Example: a Vector is associated to a Distribution. This Distribution (on its end of the association) is assigned the role of "joint distribution". Therefore the Distribution is a "joint distribution" for the Vector.

Association's name

Each association can have a name, generally taking the form of an action verb describing the association and the way it should be interpreted. If any disambiguation is required, it can be completed with an arrow indicating the direction for which the name makes sense.
Example: a Vector "creates" a Sample, therefore the association carries the name "creation".

2.3.2 Basic concepts

These concepts are the foundation bricks on which more advanced concepts will rely.

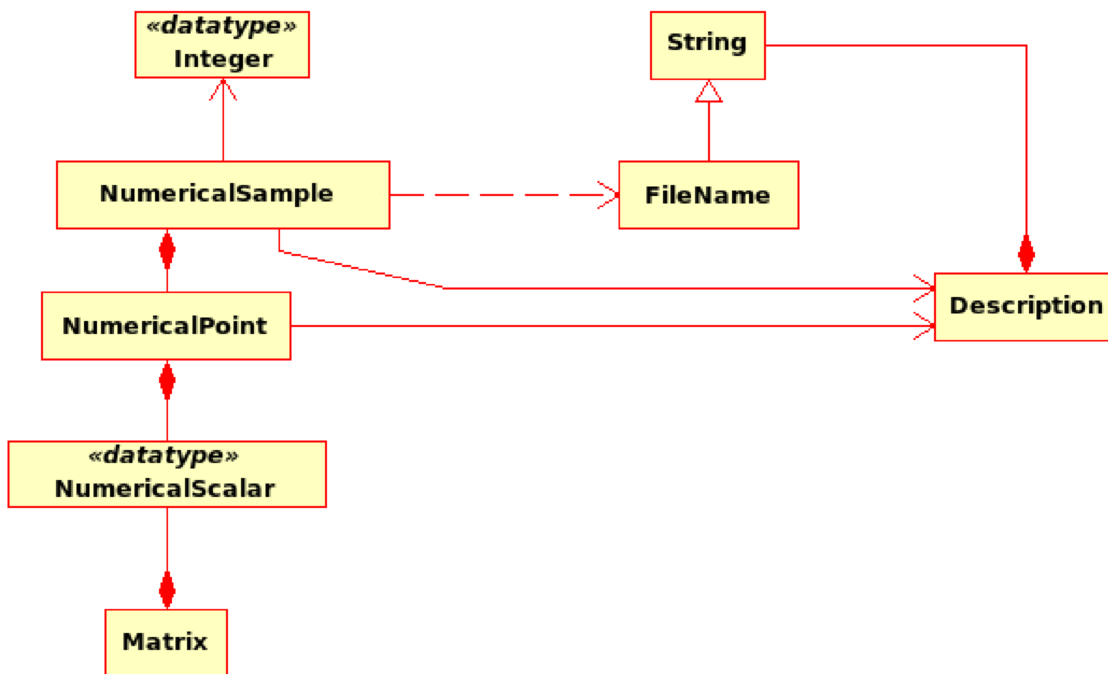


Figure 2.8: Basic concepts.

Integer

The integer is a basic type for the Open TURNS platform. It stores a whole positive or zero numerical data. Mathematically, it is a natural number. It may be used to describe the size of a sample, the dimension of a vector, the rank of a matrix, etc.

NumericalScalar

The NumericalScalar is another base type. It contains a numerical data that can be used in an uncertainty computation. Mathematically, it is a real number.

©2005-2012 EDF - EADS - PhiMeca

String

The String is a basic type that stores a text data of any length.

FileName

The FileName is a type derived from String specialized in the storage of filenames. Its syntax must correspond to a valid expression of a relative or absolute path to a file or directory on the disk.

NumericalPoint

The NumericalPoint is an elementary type made up of NumericalScalars. It covers the mathematical notion of a point in a multi-dimensional space, and once it is instantiated, its value remains constant. It can be used as a realization of a vector, mean value of a sample, etc. Its behavior is strictly deterministic.
The NumericalPoint gives access to its components. The NumericalPoint can be linked to a Description that will represent the names of its components.

Description

The Description is an elementary type made up of Strings. Its role is to provide a name or a description for an ordered set the same size as the Description. It can be associated with a NumericalPoint or a RandomVector to describe its components, with a NumericalSample to describe its points, etc.

NumericalSample

The NumericalSample is an elementary type made up of NumericalPoints. It is an homogeneous collection of points that have the same dimension. It covers the mathematical notion of a set of points. Given the multi-dimensional aspect of the points, the NumericalSample is also multi-dimensional. Its behavior is also deterministic. It can be used as a sample from a vector.
The NumericalSample gives access to its elements.

Matrix

The Matrix (see Figure 2.9) is an elementary type made up of NumericalScalars. It covers the mathematical notion of a matrix in its most general form (rectangular, asymmetric, etc.). The Matrix has a deterministic behavior. It can be specialized into more specific concepts such as square matrices, symmetric matrices, covariance matrices, etc.

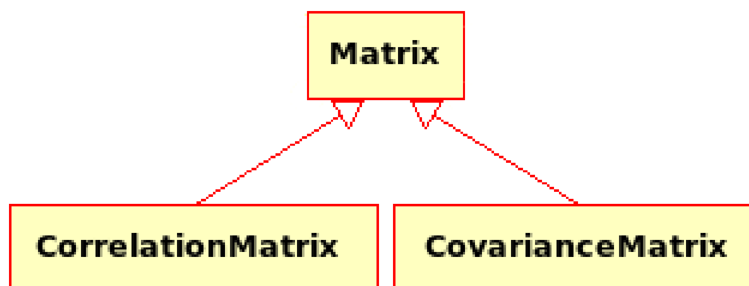


Figure 2.9: Matrix concepts.

©2005-2012 EDF - EADS - PhiMeca

2.3.3 Uncertainty concepts

RandomVector

The RandomVector (see Figure 2.10) is, from the user's point of view, the core concept. It covers both the mathematical notion of random vector and the programming concept of variable. It is essentially a multi-dimensional object, which means it always has a dimension (even if this dimension is 1). As a corollary, *there is no notion of random scalar*: within the platform, such an object is represented by a random vector whose dimension is 1.

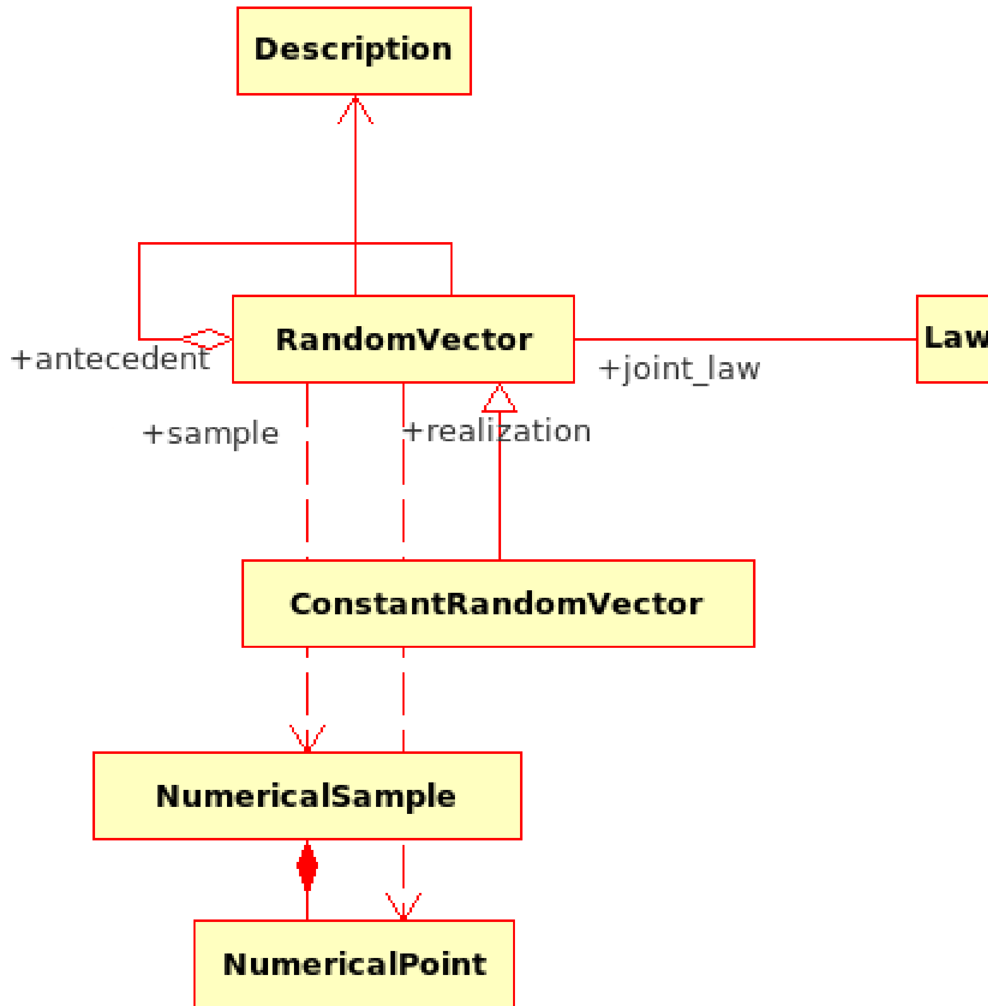


Figure 2.10: The concept of random vector.

The RandomVector is always associated with a Distribution called a joint distribution. This distribution models the vector's uncertain behavior. The RandomVector can be derived into a ConstantRandomVector whose realization is deterministically always the same NumericalPoint. The RandomVector can produce a NumericalPoint called realization and a NumericalSample called sample.
The RandomVector can be created from other RandomVectors; therefore there is a reflexive aggregation of the concept on itself, which bears the name *antecedent* on the thus created RandomVector end, so as to allow the created RandomVector to find its antecedents through the creation process. In other words, this mechanism allows a RandomVector $Y = f(X)$ (where X is the antecedent RandomVector of Y) to identify X and (if the

©2005-2012 EDF - EADS - PhiMeca

RandomVector definitions are chained) all of the previous antecedents up to the first RandomVector.

FailureEvent

The FailureEvent (see Figure 2.11) is a concept that allows to define the limite state of a RandomVector "vector" with respect to a NumericalPoint "threshold", depending on a comparison operator "operator".

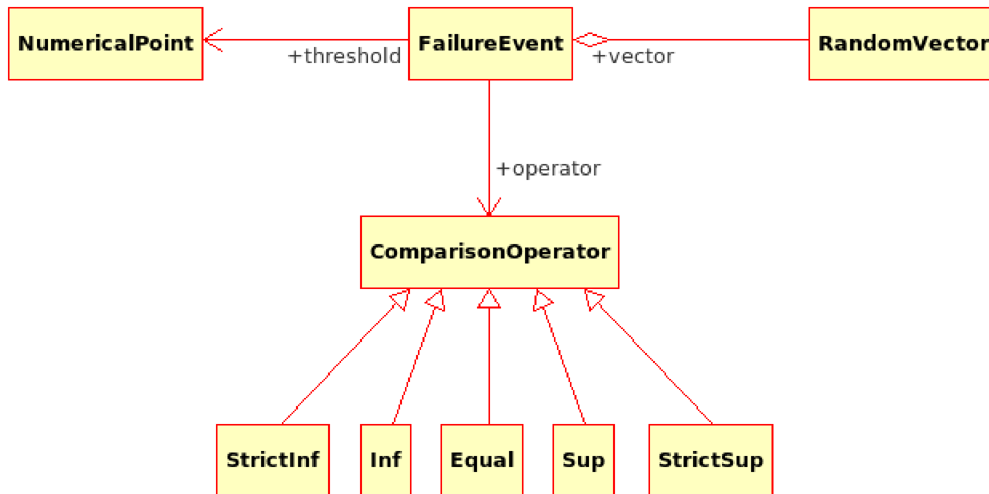


Figure 2.11: The concept of failure event.

Distribution

From the modeling point of view, the Distribution is the core concept. It is the richest concept that encompasses most of the notions related to the treatment of uncertainties, as shown in Figure 2.12. The distribution is at the crossroads of the deterministic and uncertain domains. Like the other concepts of the platform, it is a multi-dimensional concept.
The previous section about the RandomVector already showed the relationship that links it with the Distribution: each RandomVector is defined by a joint Distribution. This Distribution can in turn be derived into UsualDistribution, FunctionalDistribution, MixtureDistribution and AssemblyDistribution.

UsualDistribution

The UsualDistribution is the basic brick for the Distribution concept. The concept of UsualDistribution can in turn be derived into various distributions: Exponential, Normal, LogNormal, Triangular, Uniform, GumbelMax, GumbelMin, Pearson, WeibullMax, WeibullMin, Beta, Spline, Gamma and Histogram. To each distribution corresponds an analytical formula describing the exact behavior of the said distribution. These distributions are by definition multi-dimensional. However, in some cases, there is not always an analytical formulation for all of the dimensions. The implementation will therefore not be able to cover all possibilities.

AssemblyDistribution

When a UsualDistribution cannot be directly defined for a RandomVector of a given dimension, either because the distribution is unknown, or does not exist, or for any other reason, it is possible to combine Distributions describing the marginal distribution of the RandomVector with the help of a Copula, which determines the dependency structure of this RandomVector. The Distribution assembly mechanism is supported by the concept

©2005-2012 EDF - EADS - PhiMeca

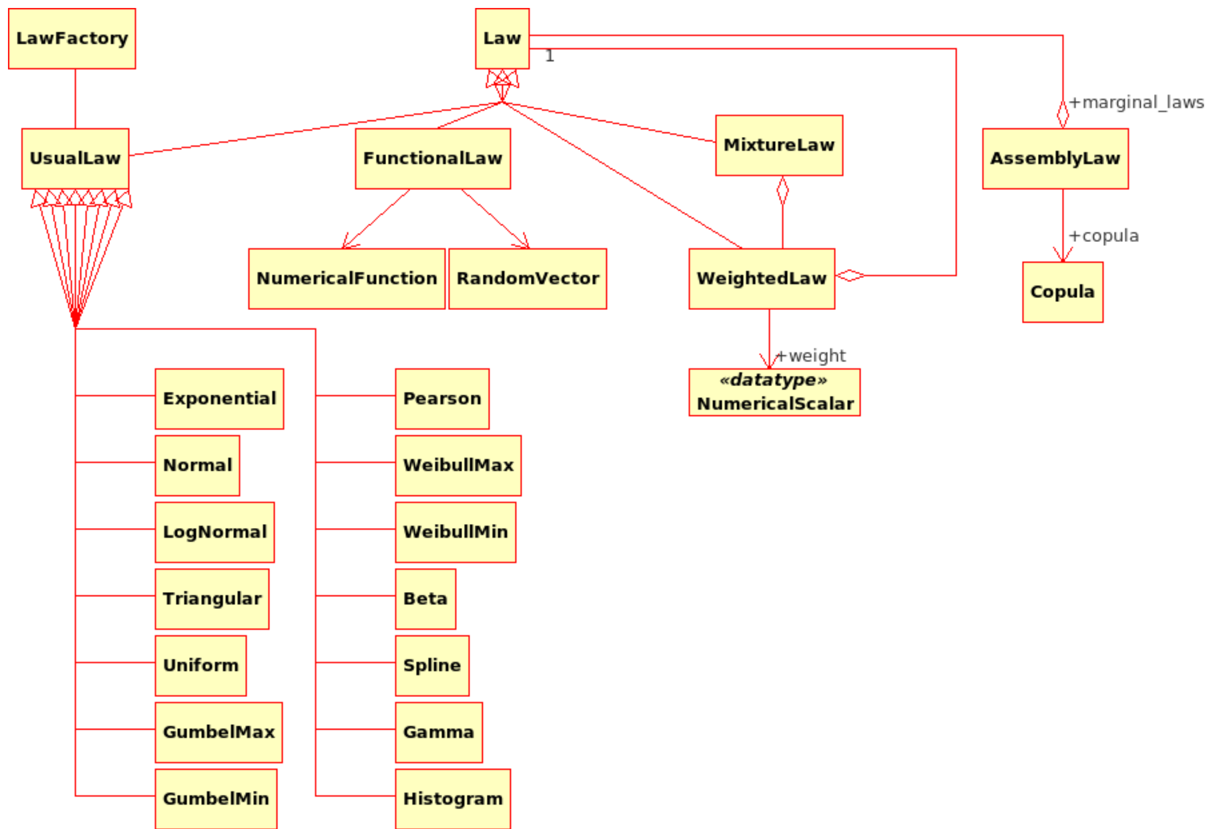


Figure 2.12: The concept of uncertain distribution.

of AssemblyDistribution. The aggregation of Distributions (called marginal_distributions) is ordered and has the same size as the RandomVector.

WeightedDistribution

The WeightedDistribution is an arbitrary Distribution associated with a NumericalScalar that serves as a weight value in a linear combination of Distributions. Each WeightedDistribution is associated with a unique Distribution for which it defines a weight.

MixtureDistribution

The MixtureDistribution is a concept defining a linear combination of Distributions. It is modeled as an aggregation of WeightedDistributions, whose underlying relationship is the sum; each WeightedDistribution brings the weight of the Distribution in the linear combination. The mathematical definition requires the *weights* to belong to the [0,1] interval and their sum to be 1.
Like all Distributions, the MixtureDistribution is a multi-dimensional concept and de facto requires the aggregation to include same-size Distributions only.

FunctionalDistribution

The FunctionalDistribution is the Distribution resulting from a RandomVector being passed to a Numerical-Function. This sort of Distribution cannot be precisely defined in the general (non analytical) case, therefore

©2005-2012 EDF - EADS - PhiMeca

the FunctionalDistribution is determined by the input parameters that define it.
NB: it was decided to use the RandomVector rather than its joint distribution, because the RandomVector (contrary to the Distribution) can access all of its antecedents. This history of antecedents is also necessary to define the Distribution.

DistributionFactory

The DistributionFactory is a concept whose need arises during the design stage. It answers the need to designate a Distribution by its type when a Distribution-type object designates a specific instance (on this topic, refer to section 3.2.2; for more details, see [GHJV]). The DistributionFactory therefore designates a family of Distributions. It is an abstract concept that needs to be derived into concrete sub-concepts such as ExponentialFactory, NormalFactory, and so on, as many times as there are concrete instantiable classes.

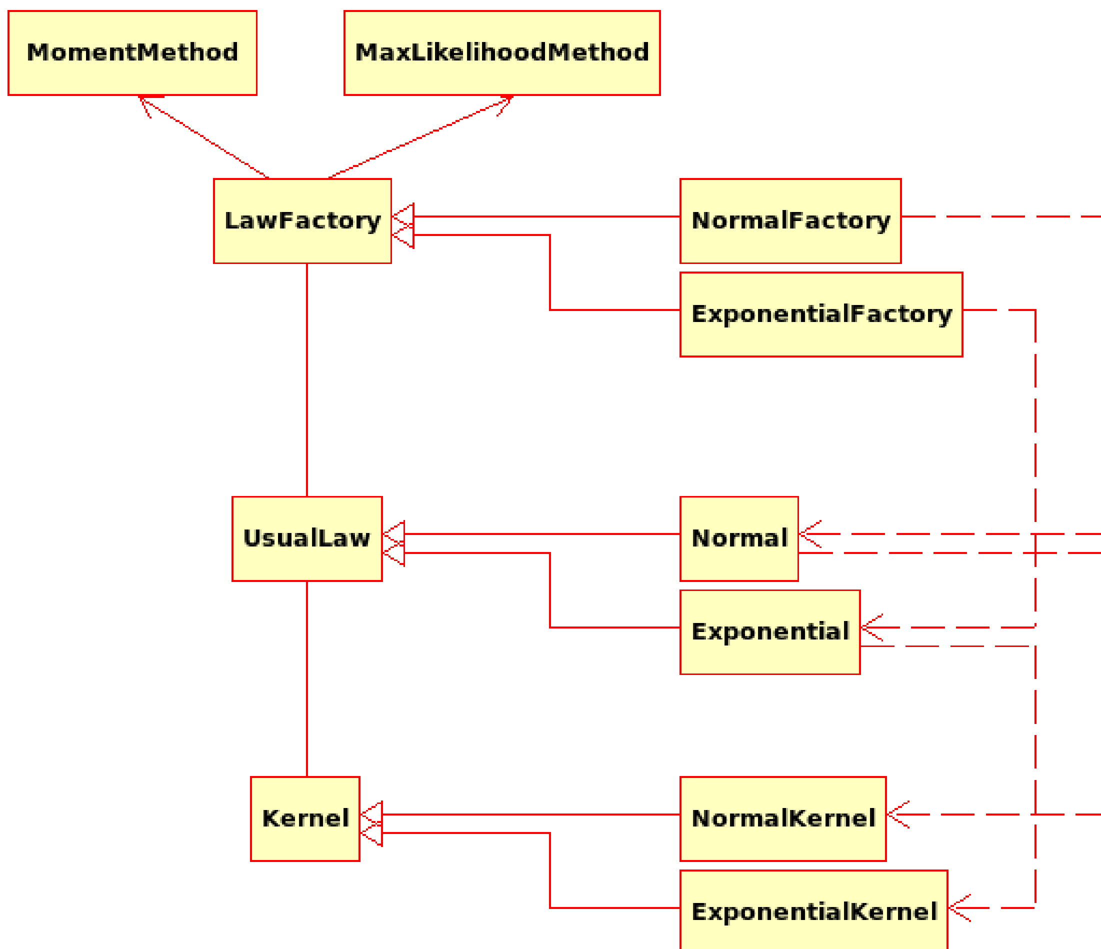


Figure 2.13: The concept of distribution family.

Therefore, to each specialized UsualDistribution corresponds a specialized DistributionFactory that can instantiate a specialized object: NormalFactory corresponds to Normal, ExponentialFactory to Exponential, and so on. NormalFactory can produce any Normal distribution, ExponentialFactory can produce any Exponential distribution, and so on for any other UsualDistribution: DistributionFactory therefore covers the mathematical concept of a distribution family.
The DistributionFactory can produce Distributions in different ways, therefore the concept is linked to the

©2005-2012 EDF - EADS - PhiMeca

MomentsMethod and MaxLikelihoodMethod concepts. Both provide the algorithms needed to produce distributions.

Kernel

On top of the distribution family notion supported by the concept of DistributionFactory, it is necessary to have a specific instance of each UsualDistribution at one's disposal. In a way, this instance represents the generator for the distribution family. This specific instance is described by the Kernel concept. Each UsualDistribution can generate its Kernel: therefore a Normal generates NormalKernel, Exponential generates ExponentialKernel, and so on.

2.3.4 Function concepts

NumericalFunction

The NumericalFunction is the concept covering the notion of mathematical function. It is an entity that takes a NumericalPoint as input and produces a NumericalPoint as output. The dimensions of both NumericalPoint can differ.

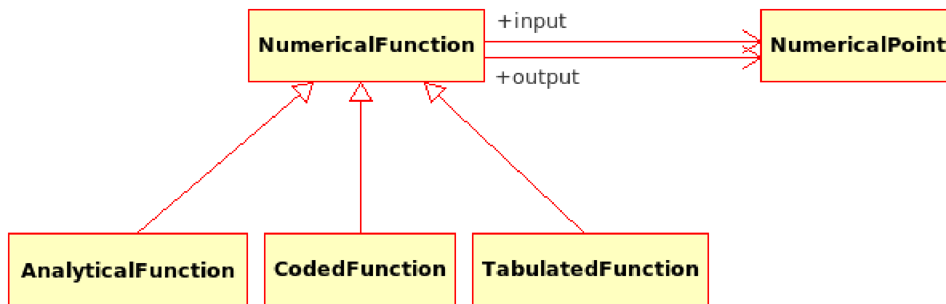


Figure 2.14: The concept of numerical function.

The NumericalFunction can be derived into AnalyticalFunction, CodedFunction and TabulatedFunction. The analysis is not directly concerned with these derived concepts, but they indirectly show:

- the link with the external code that the CodedFunction will have to support;
- the need to have pre-wired functions within the platform (in the AnalyticalFunction);
- the need for user-defined functions (TabulatedFunction).

Copula

The copula is the entity that covers the notion of mathematical copula. Mathematically speaking, it is as much a numerical function as the NumericalFunction; however, these concepts have been made distinct because they are used in very different contexts : the NumericalFunction is used in conjunction with RandomVectors and NumericalSamples, whereas the Copula is used only to define an AssemblyDistribution. Moreover, as we will see in the design model, this separation is emphasized by the operations required from each concept.

©2005-2012 EDF - EADS - PhiMeca

2.3.5 Algorithm concepts

SimulationAlgorithm

The SimulationAlgorithm is the concept covering all of the uncertainty propagation methods in the platform. It is an abstract concept that derives into sub-concepts such as MonteCarlo, FORM, SORM, SRSS, and so on. A SimulationAlgorithm produces a SimulationResult as output, which stores all the information needed for the prioritization stage.

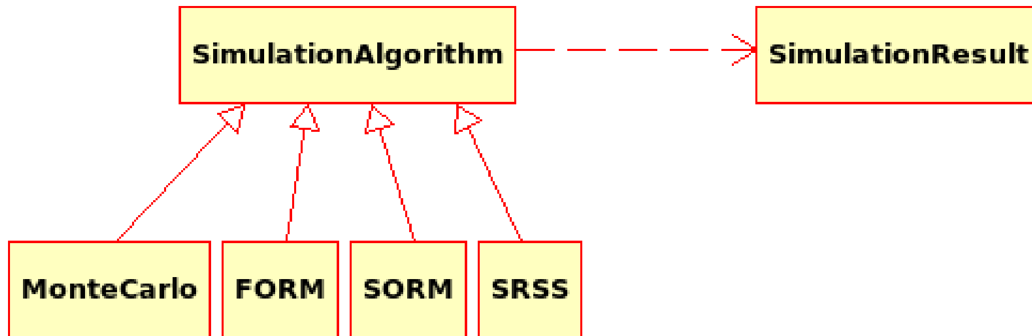


Figure 2.15: The concept of simulation algorithm.

2.4 Synopsis of a functional architecture

2.4.1 Packages

The tasks of modeling, analysis (described in this chapter) and design (described in next chapter) enable us to define concepts and objects closely related to one another. These will be grouped in the following packages:

- *GUI*: this package encompasses all objects directly linked to the graphical interface (e.g. Qt objects);
- *Graphical objects*: this package encompasses all of the objects whose role is to adapt field-specific and technological objects into GUI objects. It can be seen as the graphical interface for field-specific objects (e.g. histograms, pie charts, graphs of functions, and so on);
- *TUI*: this package encompasses all the objects interfacing with field-specific objets for a text-based use of the platform (e.g. SWIG objects);
- *Procedures*: this package encompasses any processing that is not (or cannot) be modeled with field-specific or technological objects (e.g. conditional random vector). It also contains sequences that represent typical action chains frequently used in uncertainty treatment studies;
- *Uncertain objects*: this package encompasses objects and concepts linked to the modeling of field-specific data with uncertain variables (e.g. random vectors, distributions, etc.);
- *Simulation algorithms*: this package encompasses the various uncertainty propagation algorithms supported by the platform and the objects associated with the results (e.g. Monte-Carlo, FORM, SORM, etc.);
- *Statistical objects*: this package encompasses the objects and concepts associated with the statistical processing of field-specific data (e.g. sample);

©2005-2012 EDF - EADS - PhiMeca

- *Optimization functions*: this package encompasses the standard optimization methods needed for the simulation algorithms (e.g. SQP);
- *Basic numerical functions*: this package encompasses the mathematical and numerical functions standardly distributed with the platform (e.g. log, exp, sin, cos, etc.);
- *Basic classes*: this package encompasses the utility classes on which all of the higher level packages rely (e.g. String, FileName, Matrix, Tensor, etc.).

2.4.2 Layers

The previously introduced packages can also be grouped into software layers, each layer representing a different abstraction level of the Open TURNS platform.
As can be seen in Figure 2.16, the layers can be stacked as follows:

- *External services*: these are all the development prerequisites on which the platform relies, as well as the services offered by passive actors such as the external code or the storage system;
- *Basis brick*: it encompasses the packages offering core statistical and mathematical services for the platform;
- *Field-specific brick*: it is the essential layer that represents the core of the platform, in which are to be found packages responsible for the modeling and the propagation of uncertainties;
- *Sequences*: this layer assembles components from the field-specific brick to carry out complex uncertainty processing tasks;
- *User interface*: it is the layer responsible for the display and abstraction of field-specific objects into cognitively representable elements.

The perimeter of the Open TURNS platform is the solid border that surrounds the four top layers. The bottom layer representing the external services does not strictly belong to the platform: it is made up of prerequisites that are not included in the developments and in the modeling.
Cutting out the functional architecture into layers like this has direct implications on the platform's development process: the top layers will have to be developed on the basis of the bottom layers. Therefore, the development will be carried out chronologically, from the bottom layers to the top ones. This does not mean that one layer has to be entirely finished before the development of the next one starts. However, the packages on which one layer relies will have to be at least partially developed before actually beginning the development of the said layer's packages.

2.4.3 General diagram

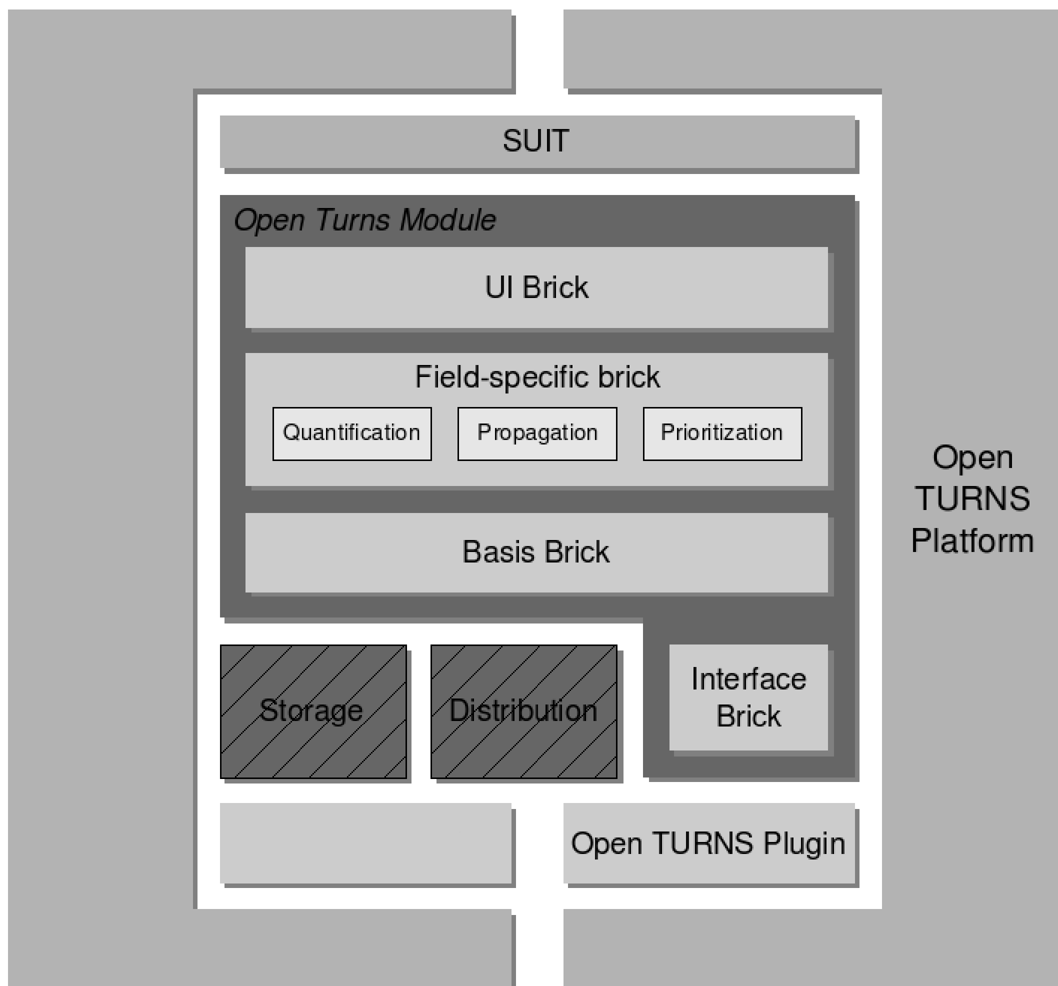
Figure 2.16 shows the general diagram of the Open TURNS platform and its brick structure:

- *Interface brick*: used to provide communication with the operating system and the platform's prerequisites;
- *Basis brick*: basic components of the platform on which the functional, higher-level bricks rely;
- *Field-specific brick*: components used to model an uncertainty treatment study (quantization, propagation, prioritization, etc.);
- *UI brick*: components used to manipulate the platform and to visualize results.

©2005-2012 EDF - EADS - PhiMeca

The Storage service has the responsibility to store the computated data; the Distribution service's role is to distribute computations on a computer network. Both are considered non-critical: their development can be postponed in time in order to focus on the platform's core.
All of these elements make up the development perimeter within the Open TURNS project.
The Plugin component is in charge of the communication between the platform and the external code. The platform will standardly offer a version providing a mechanism for generic communication with external codes, but aside from this version, the Plugin component is outside of the project's scope: its implementation is the responsibility of the external code, which will have to follow the API provided by Open TURNS.
Figure 2.17 describes the architecture of the Open TURNS platform from a functional angle. It shows the packages resulting from the previously described concept groups, as well as other packages resulting from the design stage, the use cases or the general specifications of the platform. The layers are also represented as frames around the packages.

©2005-2012 EDF - EADS - PhiMeca



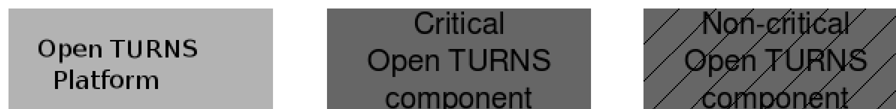


Figure 2.16: General diagram.

©2005-2012 EDF - EADS - PhiMeca

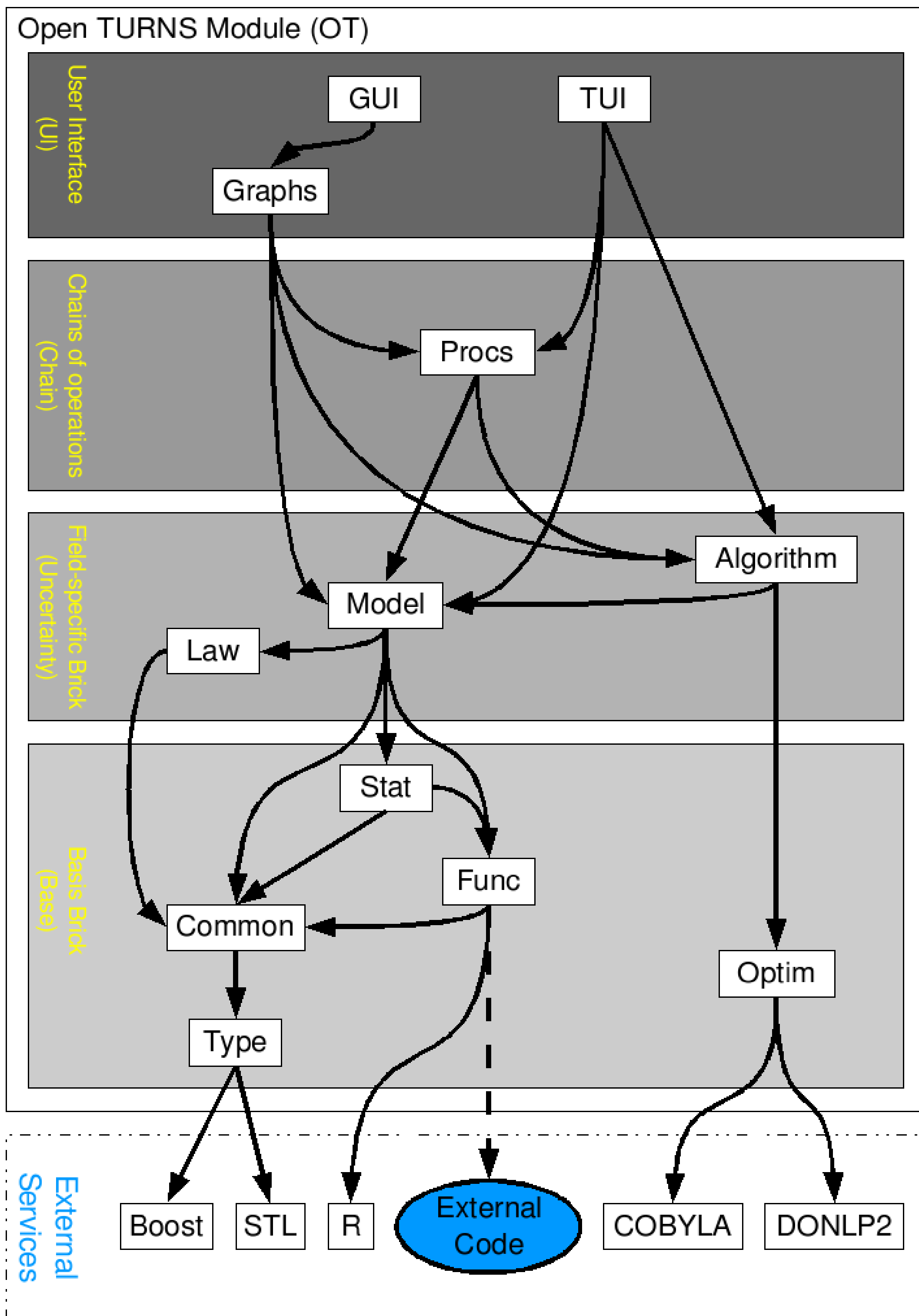


Figure 2.17: Functional diagram.

©2005-2012 EDF - EADS - PhiMeca

Chapter 3

Design model and software architecture

Before describing in detail the design of each class (package after package), this chapter introduces a few design models that are widely used within the Open TURNS platform.

3.1 Notation and glossary

The presentation of the design model relies on a graphical UML notation (see [UML]) that needs to be introduced for the readers who may not yet be familiar with this type of representation. This notation largely draws upon the one used for the analysis, especially as far as associations are concerned, but it needs to be supplemented to take into account the new elements (which will be described afterwards). Please refer to section 2.3 for the notations that were already introduced.

3.1.1 Class

The class is the natural entity that materializes the analysis concept. It is a type that defines the behavior of any object belonging to the class. It is represented by a rectangle horizontally divided into three parts: in the top part is the name of the class; the middle section gives the list of attributes; the bottom part contains the list of methods. The attribute and method sections are optional (see Figure 3.1).

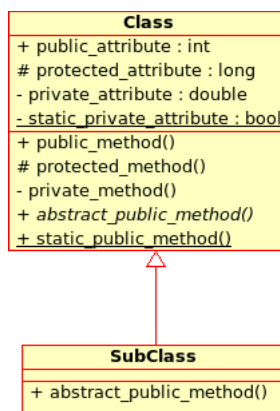


Figure 3.1: Representing classes.

3.1.2 Object

An object is an instance of a class. It is graphically represented by a rectangle containing the name of the object and its class, separated by a colon (:) and underlined. When the name of the object is not significant, it can be omitted; the class can never be.

3.1.3 Attribute

The attribute is an internal element of the object or of the class that describes (partially or entirely) the state of the object. It is defined by a name, a type, a visibility and a scope.

3.1.4 Method

The method is an operation that affects an object or a class. It is the natural place to define how an object can be processed. The method is defined by a signature (see [Del]), a visibility and a scope.

3.1.5 Visibility

Visibility defines the possibility of accessing an attribute or a method of a given class. There are three possible values:

- *Public*: the attribute or method can be accessed by any object from any class.
- *Protected*: the attribute or method can be accessed by any object from a class deriving from the aforementioned class, but not by any other object .
- *Private*: the attribute or method can only be accessed by objects from the aforementioned class.

This (very simple) definition of visibility is the one given by the C++ language (see [Del]), which is the language chosen for the platform's implementation. Other languages may have different definitions of visibility.
Visibility is graphically represented by a plus sign (+) for public, a hash sign (#) for protected and a minus sign (-) for private.

3.1.6 Scope

An attribute or method belongs by default to the object instantiated from the class where it is defined. For example, the lifespan of a standard attribute is the same as the lifespan of the object carrying it. This behavior can be modified by declaring the object as static: its lifespan then become the same as the application's, and it can only be accessed through the class (and not through the object).
In the same way, a method that was declared static can no longer process objects instantiated from its class; it can only process the class itself and its static attributes.
The static scope of an attribute or a method is graphically represented by underlining the said attribute or method.

3.1.7 Abstract class and method

A class or a method can be defined as abstract. For a class, it means that it cannot (and should not) be instantiated and that there are no available objects belonging to this class. This class must be derived into a sub-class in order to be usable.
An abstract method is a method whose signature is declared but for which no definition has been given. A class with an abstract method is de facto abstract, since it cannot be instantiated due to the undefined method.

©2005-2012 EDF - EADS - PhiMeca

In order to be able to derive the abstract class into a concrete sub-class, it is necessary to define the abstract method in the sub-class; if the abstract method is not defined, the abstract trait is propagated to the sub-class. Graphically speaking, an abstract class or method is represented by a name or signature written in italic font.

3.1.8 Method call

The UML notation defines a representation of interactions between objects defined as class instances. This representation allows to visualize the links between these objects and the communication that they establish. It is shown in a collaboration diagram in which objects are represented as instances (and no longer as classes), and the links between objects support the name of a method issued from the calling object to the called object. Figure 3.2 shows an example of such a diagram.

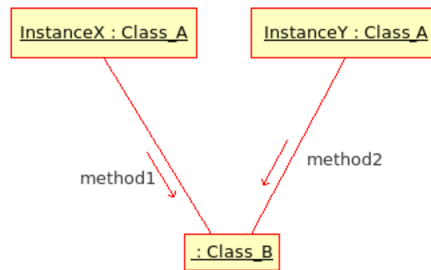


Figure 3.2: Collaboration diagram.

3.2 Design patterns

3.2.1 Introduction

Software design shows the recurrence of some patterns, whether within the same piece of software or in several applications (which can differ in many ways). These patterns have been catalogued ([GHJV] can be referred to with great benefit on this subject), described and implemented in numerous situations that prove their universality and their ability to solve recurring problems that the software architect is faced with.
The following sections give an overview intended as much for the reader's understanding of the document as to establish a common vocabulary for software architect. The latter ones will find here standard design diagrams applied to the specific case of Open TURNS, which can help them better apprehend the tool's specificities and the design and implementation choices that were made.

3.2.2 Singleton pattern

The Singleton is a pattern used to ensure that at any given time, there is only one instance of a class (A); it provides an access point for this unique instance.
This is implemented by creating a class (Singleton) with a static private attribute (uniqueInstance) initialized with an instance of class A and whose reference (or pointer) is returned by a static method (instance). Figure 3.3 illustrates the Singleton pattern.
It is a very common pattern that allows to find and share an object (which must remain unique) in different portions of code. Examples of such objects include shared hardware resources (standard output, error, log, etc.), but also internal functions that cannot or must not be duplicated (e.g. a random number generator).

©2005-2012 EDF - EADS - PhiMeca

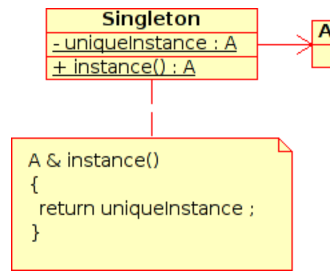


Figure 3.3: Singleton structure.

3.2.3 Factory pattern

This pattern allows to define a unique interface for the creation of objects belonging to a class hierarchy withouth knowing in advance their exact type. Figure 3.4 illustrates this pattern. The creation of the concrete object (ClassA or ClassB) is delegated to a sub-class (ClassAFactory or ClassBFactory) which chooses the type of object to be created and the strategy to be used to create it.

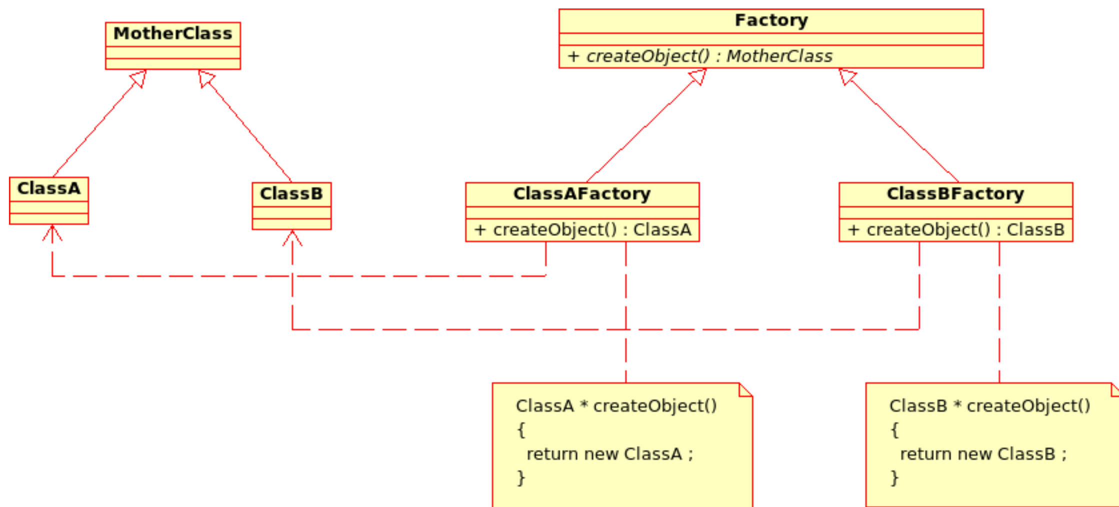


Figure 3.4: Factory structure.

This pattern is often used to dynamically create objects belonging to related types (e.g. to instantiate objects within a GUI according to the user's behavior). It can also be used to back up and read again a document written in a file by automatically re-instantiating objects. It is a pattern that makes code maintenance easier by clearly separating the objects and their instantiation in distinct and parallel class hierarchies.

3.2.4 Strategy pattern

The Strategy pattern defines a family of algorithm and makes them interchangeable as far as the client is concerned. Access to these algorithms is provided by a unique interface which encapsulates the algorithms' implementation. Therefore, the implementation can change without the client being aware of it.
This pattern is very useful to provide a client with different implementations of an algorithm which are equivalent from a functional point of view. It can be noted that the Factory pattern described earlier makes use of the Strategy pattern.

©2005-2012 EDF - EADS - PhiMeca

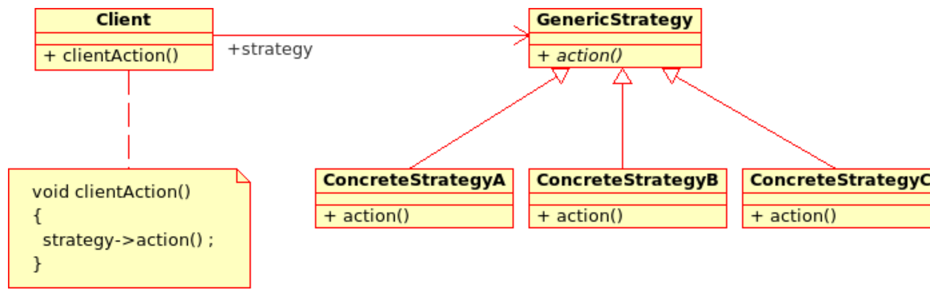


Figure 3.5: Strategy structure.

3.2.5 Composite pattern

The Composite pattern is used to organize objects into a tree structure that represents the hierarchies between component and composite objects. It hides the complex structure of the object from the client handling the object.

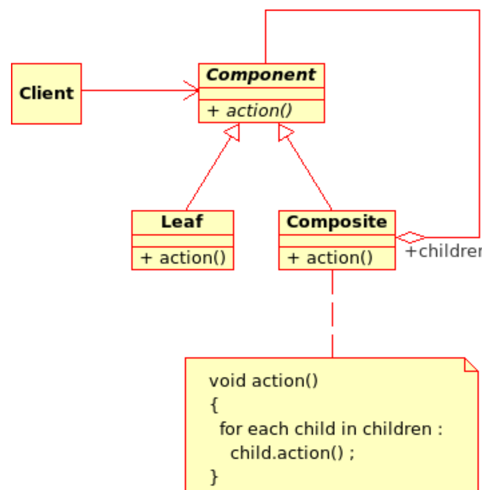


Figure 3.6: Composite structure.

Figure 3.7 shows an example of tree modeled by the Composite. The Composite objects make up the tree nodes whereas the leaves can be any concrete object deriving from Component.
The Composite pattern is an essential element of the design model for the Open TURNS platform. It can be used to model numerical function composition, random vector composition, etc. It can be found in several aspects of the modeling brick. Any related objects tree structure can rely on the Composite pattern with benefit.

3.2.6 Iterator pattern

The role of this pattern is to provide a means of sequentially accessing elements of a collection or of an aggregate, without breaking the encapsulation principle that forbids to expose the collection's underlying implementation. For an iterator to be able to access the elements of an aggregate encapsulating its internal representation, it must have privileged access. Figure 3.8 illustrates this pattern.
The iterator is jointly used with the Composite pattern to traverse trees relying on internal objects (vectors, functions, etc.).

©2005-2012 EDF - EADS - PhiMeca

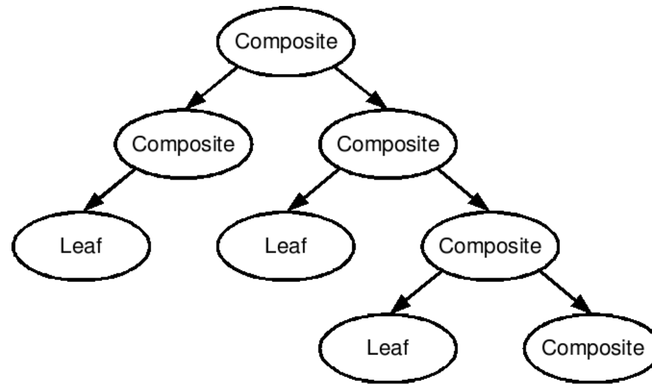


Figure 3.7: Example of tree modeled by the Composite.

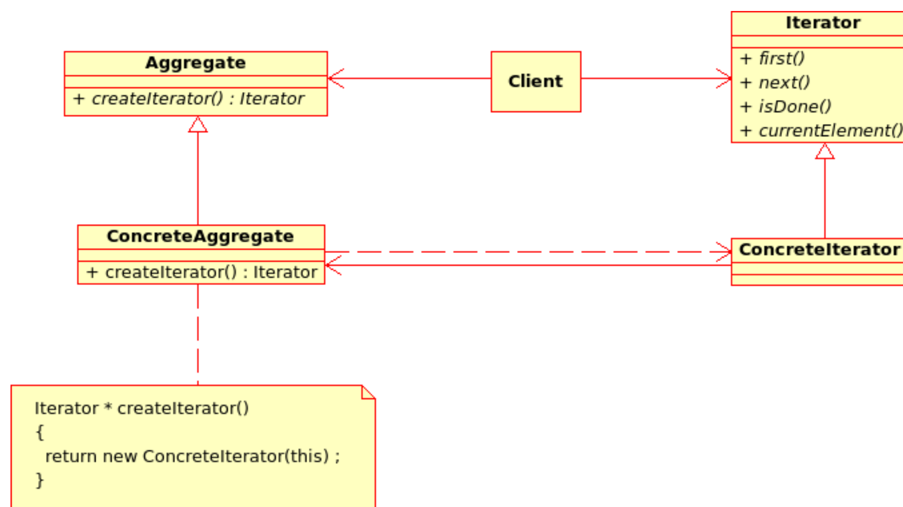


Figure 3.8: Iterator structure.

3.2.7 Visitor pattern

The Visitor pattern allows to conduct a given operation on a hierarchy of objects without changing the classes of these objects. The processing is supported by the Visitor which moves from one object to another and applies the correct treatment on each object according to its class. By changing the Visitor, we can therefore apply different treatments. The class hierarchy must not substantially evolve through these operations.

Figure 3.9 illustrates the twofold class hierarchy of Nodes and Visitors which are orthogonal (contrary to the Factory in which they are parallel): the Visitor has as many methods are there are Nodes in the hierarchy.

As far as the client is concerned, the processing is limited to instantiating a Visitor of the right type and applying it to the class hierarchy.. The Visitor pattern is often jointly used with the Composite pattern (to model object trees) and with the Iterator pattern (to traverse the collection of objects).

The Visitor can be used within the Open TURNS platform to conduct global operations on the trees representing composite functions (e.g. for derivation), on random vectors (to save them or evaluate a realization or a sample), etc.

©2005-2012 EDF - EADS - PhiMeca

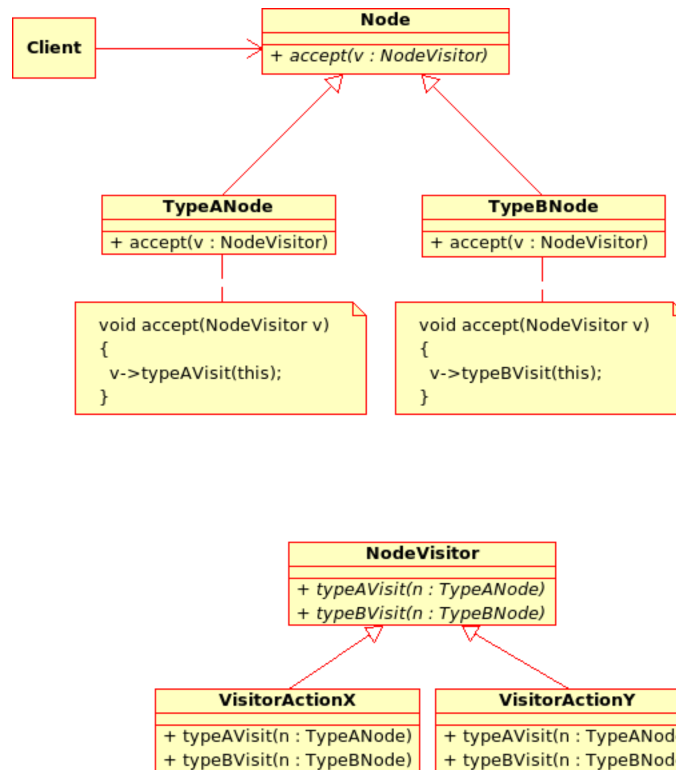


Figure 3.9: Visitor structure.

3.3 Package description

In this section, we begin the study of the classes and objects developed for the Open TURNS platform. The definitions given here are those implemented in the tool's final code. Contrary to the analytical approach (which aimed at viewing the platform's constituting elements from a functional and more general point of view), this chapter adopts a development oriented point of view. We shall begin with the lowest layers of the system and move up to the top-level ones.

The reader may find symbols symbols made up of a name followed by two colons (::): it designates a namespace as described in section 4.1. For example, OT:: designates the namespace called OT.

All of the elements developed for the Open TURNS platform belong to an OT namespace.

NB: Very often, the class definitions indicate that the values used to create an object cannot be modified once the object is created. This conservative mechanism was intentionally adopted in the early stages of development so as to narrow the object's interface and to simplify the mechanisms handling dependencies between objects. However, this may prejudice the ergonomy of the platform. Shall this decision cause problems, the constraint can be lifted. It would then only require to add the appropriate methods to the object's interface, which is easier to handle than withdrawing methods already implemented.

3.3.1 Basis brick

This encompasses all of the classes supporting the field-specific brick, which means the programming classes associated with the language or the implementation, as well as the classes offering elementary functions associated with the field-specific aspects of the platform.

©2005-2012 EDF - EADS - PhiMeca

Data types

The source files of these bricks are stored in the Base/Type subfolder of the source tree.

Id The Id is a *unique* identifier for each object instantiated by the Open TURNS platform. Unicity is guaranteed at the level of a study but not between different studies. The identifier must be compact so as not to weigh down on the objects (especially the smaller ones) and it must be generated quickly so as to not substantially slow down the object creation. It is thus preferable to associate it with an elementary type of the language, for example an *unsigned long*. Moreover, it must support a weak order relation (operator <) and a comparison relation (operators == and !=) so that it can be used as a key in a container.
Not all objects possess an Id: only those which need to be persistent have one. Therefore temporary objects, volatile objects and any object that can be re-created from its value carry no Id. By default, an object which can be backed up in a study has an Id.
When re-creating a backed up object that had an Id, the Id is restored to its value prior to the backup: this means that the Id is saved within the object in the study.

NumericalScalar In its standard version, the Open TURNS platform only performs computations on real numbers. Future versions may include other numerical types.
This type of real numbers is designated as NumericalScalar and must be able to store any floating point number. Thus the NumericalScalar type is associated with the *double* type of the programming language.

String So as to provide the user with a means of naming the objects they are handling according to the current study, the platform will use a type allowing to store character strings of any length and encoding type (see [UNI]).
Please note that this type is not appropriate for naming files, directories, URL, etc. For more details on this, refer to the FileName type described in next section.
The String type is associated with the std::string type from the C++ STL (see [Del] and [Meyb]). It depends on the locale setting of the user.

FileName The FileName type differs from the String type in that it is meant to store the names of files, directories, URLs, etc., not any type of word the user might need. The names stored in a FileName object have to be interpretable on the computer executing the Open TURNS platform. The character set of the encoding system used by FileName may be more limited than that of String.
The FileName type is associated with the std::string type from the C++ STL (see [Del] and [Meyb]).

Bool In order to store or return a boolean value, the platform uses the Bool type. The Bool type is based on the standard C++ bool type.

UnsignedLong In order to store natural numbers, the platform uses the UnsignedLong type. This type appears everywhere where a collection or table size, an index, a quantity, etc. are involved.
It can be derived into more specialized sub-types such as Size, Index, etc.
The UnsignedLong type is programmed as an *unsigned long*.

LibraryHandle In order to store or return a handle for a loaded dynamic library, the platform uses the LibraryHandle type. The LibraryHandle type is based on the standard C++ void * type.

LibrarySymbol In order to store or return an identifier for a symbol belonging to a library, the platform uses the LibrarySymbol type. The LibrarySymbol type is based on the standard C++ void * type.

©2005-2012 EDF - EADS - PhiMeca

Common classes

The common classes are programming objects used to support more elaborate classes. They are meant to provide common services for all objects handled in the platform. The source files of these classes are in the Base/Common sub-folder of the source tree.

Object The vast majority of objects instantiated in the Open TURNS platform derive from the Object class. This abstract class (described in Figure 3.10 provides support for the object's name *name_* . It can also be used to implement other global properties for the objects, which will be defined in future versions of the platform.
The object's name can be accessed with the *setName()* and *getName()* methods. *Neither one of these methods is virtual, thus they cannot be redefined.*
The object's name is passed to the Object class constructor. The default value for the object's name is the constant value: Unnamed.

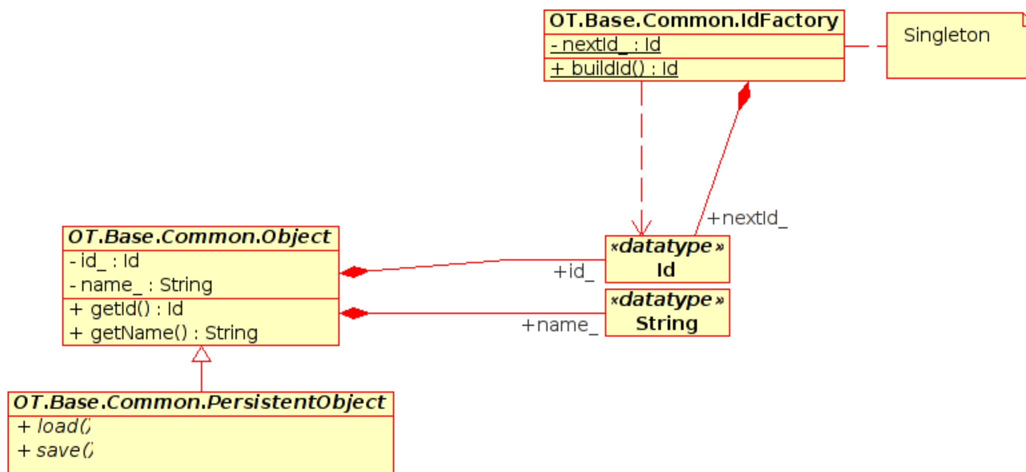


Figure 3.10: Object, PersistentObject and IdFactory classes.

The Object class also provides a virtual method *str()* which allows any instance deriving from Object to be displayed as a stream. Each class deriving from Object must overdefine this method.
The Object class defines a comparison operator *operator ==()* which tests if the values of two objects are equal. Note that this is different from an identity test (which is defined in the PersistentObject class).

PersistentObject The PersistentObject class is a specialization of Object which provides backup and restore services for the objects based on a stream (std::stream). This stream can be connected to a file, a socket, etc. It is described in Figure 3.10.
It offers a *load()* method which allows to re-read an object from the stream, and a *save()* method to write the object in the stream. Both methods are pure virtual methods and thus they will have to be redefined in the derived classes. The stream handling mode on which the reading and writing will be based on is not defined yet: method arguments, global object, study related object, etc. are different possibilities that currently still need to be studied.
This class also defines an id_ attribute which contains a unique identifier for the object. This identifier is automatically assigned when the object is created and it cannot be redefined. It is a read-only attribute that can be accessed with the *getId()* method. The identity of two objects can be tested with the *is()* method, which returns a *true* boolean value if both objects are identical.

©2005-2012 EDF - EADS - PhiMeca

IdFactory The IdFactory is a factory (see section 3.2.2) which produces a unique identifier Id each time its *buildId()* is invoked. It is at the IdFactory level that the unicity and sequentiality of the Ids are handled. The IdFactory must be unique so as to prevent generating several identical Ids. This object must therefore implement the Singleton design pattern (see section 3.2.1). The rules that govern the Ids (see section 3.3.1) imply that this singleton is managed at the study level.
The IdFactory is described in Figure 3.10.

Exception Unless in duly justified and allowed cases, any error detected within the Open TURNS platform is brought all the way up to the user level through exceptions. The Exception class, which derives from std::exception, is the mother class to all of the platform's exceptions. It is forbidden to develop an exception class which would not derive from Exception.
Exception and all of its sub-classes must redefine the methods of std::exception. The Exception class is described in Figure 3.11.

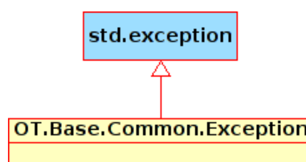


Figure 3.11: Exception class.

Exception can be derived into various sub-classes such as:

1. FileNotFoundException
2. InvalidArgumentException
3. NoWrapperFileFoundException
4. WrapperFileParsingException
5. WrapperInternalException
6. XMLException
7. XMLParserException
8. DynamicLibraryException
9. etc.

FileNotFoundException This exception is raised when a file required by the platform is not found.

InvalidArgumentException This exception is raised when an argument passed to a method is incorrect and prevents the method from following its usual course of actions.

NoWrapperFileFoundException This exception is raised when the wrapper description file cannot be found.

WrapperFileParsingException This exception is raised when an error is detected in the wrapper description file, which prevents the file parsing.

©2005-2012 EDF - EADS - PhiMeca

WrapperInternalException This exception is raised when a fatal error occurs in the wrapper or one of its methods.

XMLException This exception is raised when an error occurs during the XML description file's parsing.

XMLParserException This exception is raised when the XML parser for the description file detects an internal error.

DynamicLibraryException This exception is raised when an error is detected while loading or unloading one of the platform's dynamic libraries.

Iterator The Iterator class is the abstract class from which all iterator classes derive. As described in Figure 3.8, the Iterator defines the following pure virtual methods:

- *begin()* to point at the beginning of the collection;
- *end()* to point at the end of the collection;
- *more()* to determine if there are any elements in the collection that were not reached yet;
- *currentElement()* to access the element currently pointed at in the collection by the iterator.

These methods must be re-defined by Iterator's sub-classes. Iterator only defines an interface.

Pointer Figure 3.12 describes the Pointer class as using the BOOST shared_ptr class (see [BOO]). It is a *smart pointer* supporting copy construction and assignment. The ownership of the object is shared by all of Pointer's instances pointing at it and the last pointer is in charge of deleting the object.
The use of the Pointer class allows to noticeably reduce the platform's memory consumption and enhances its performance since it prevents useless object copy operations and costly memory allocations. In return, however, this requires a subtle management of the objects and a careful study of their life-cycle within the platform: deleting the pointer does not ensure that the object itself is deleted, since the object ownership may have been transfered to another pointer. *More specifically, the use of Pointer de facto forbids the use of C++ classic pointers. The only exception to this rule is the parameter transfer between the platform and the wrapper, since the API is written in C.*
The Pointer class is a template: there will have to be as many specialized Pointer classes as there are classes subject to this mechanism. A naming convention will be described in [OT] so that the pointer's name can be deduced from the class name.

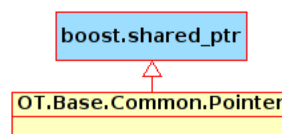


Figure 3.12: Pointer class.

The Pointer class offers the following services:

- constructors and destructors respectively support and delete the object pointed at;
- the assignment operator *operator =()* and the copy constructor transfer and share responsibility for the object;

©2005-2012 EDF - EADS - PhiMeca

- the *reset()* method replaces the object pointed at;
- the operators *operator *()* and *operator ->()* give access to the object pointed at;
- the *swap()* method allows to swap objects pointed at by two pointers;
- the *unique()* method allows to check if the pointer is the object's sole owner;
- the *use_count()* method allows to check the number of owners for the object pointed at.

By default, the Pointer class uses the BoostPointerImplementation class for its implementation. This choice can be re-defined when using the Pointer class by passing the name of the implementation class as a second argument for the template. In the initial version of the Open TURNS platform, only the BoostPointerImplementation class will be offered.

BoostPointerImplementation This class represents a shared pointer based on the BOOST library (see [BOO]). It is used through the Pointer class.

Thread The Open TURNS platform is designed to launch and control computations that can take quite some time. The general ergonomy implies that the user remains in control of the tool during these costly operations. It is thus necessary to provide a mechanism to launch computations asynchronously, based, for example, on threads (see [LB]). The Thread class, which is described in Figure 3.13, implements this mechanism. A Thread object is created with the support of a Threadable object encapsulating the computation to be carried out. The execution itself starts when the object's *run()* method is called. This method then launches the Threadable object's *run()*. The *run()* method of the Thread object is asynchronous: it immediatly returns and the execution of the programm goes on parallelly to the computation. This method may raise exceptions if the computation launch fails for any reason.
The Thread object can be interrogated with the *query()* to know the computation's progress.
The *wait()* method is used to wait for the computation to terminate. It sets a lock until the computation terminates. It returns if the computation terminates in a normal or abnormal state. In the latter case, an exception is propagated if it was emitted by the computation.

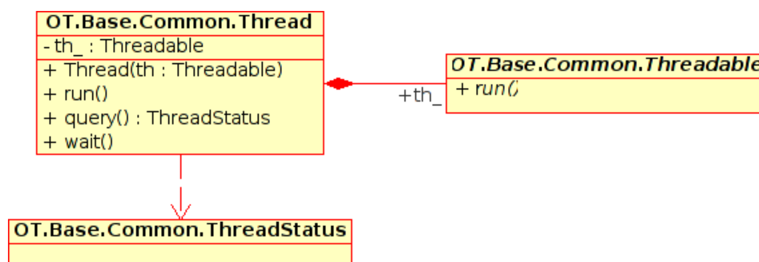


Figure 3.13: Thread, Threadable and ThreadStatus classes.

ThreadStatus The *query()* method of the Thread class returns a ThreadStatus object which gives access to information about the computation's course: its state (running, terminated, stopped, etc.), its progress (x% computed), etc.
Details about ThreadStatus methods will be given in a future version of this document.

©2005-2012 EDF - EADS - PhiMeca

Threadable The Threadable class is an abstract class defining an interface for all classes which can be passed as arguments to a Thread object. It defines a pure virtual *run()* method which will have to be re-defined in sub-classes.
This *run()* method must execute the computation implemented by the class.
Though not defined here yet, the Threadable class must provide a means of communicating with the Thread class, so as to notify if of the computation's state and progress. This way, the Thread class can in turn relay this information to the user.

Lockable / Lock The abstract Lockable class defines an interface for the derived classes. All of the classes used in a multithreaded environment (which will have to support concurrent access from several threads) must derive from Lockable.
Lockable defines a sub-class Lock which sets up a limited scope locking mechanism. The definition of a lock on the object deriving from Lockable helps ensuring the serialization of the block in which this lock is defined. This mechanism allows to easily declare critical sections without having to worry about the lock's lifespan or its release.

Path Path is a non-instantiable class which defines static methods to search through the computer's filesystem tree. These methods implement algorithms that are common to all classes of the platform needing them. Thus all mechanisms are gathered in a single and well identified structure, which makes the maintenance easer.
The static *GetDirectoryList()* returns the list of directories in which the search must be conducted. The returned object is a list of directories which can be modified.
The static *FindFileByNameInDirectoryList()* returns the full path to the first file whose name is passed as an argument, in the directory list also passed as an argument. The function simply searches the filesystem. It does not provide any locking mechanism or any sort of preemption on the file, which means that the file can have disappeared or changed names when the result of *FindFileByNameInDirectoryList()* is actually used.

StorageManager This abstract class defines an interface for all of the storage managers handling data on the filesystem. It defines the methods that any instantiable manager must provide.
A StorageManager browses through all of the PersistentObjects and saves them on the filesystem. Conversely, a StorageManager can restore saved PersistentObjects to their prior saving state without any noticeable change in the application's execution once the data is reloaded.

WrapperFile The WrapperFile class offers a means of handling the description file for an external code wrapper. The class constructor allows to read the file in the filesystem. Once the file has been read and parsed, the class data structures are updated with the file's descriptive elements.
The class offers a set of accessors to update and retrieve the data read in the description file.
It also provides a static *FindWrapperPathByName()* method which relies on the Path class to search the file in the filesystem.
Depending on how the platform's code is compiled, the WrapperFile class allows to read and parse files in different formats: XML, plain text, etc. Depending on the case, the class calls the appropriate parser which may have been included in the platform.

WrapperData This structure stores the data that can be exchanged between the wrapper and the Open TURNS platform. This data is read in the wrapper's description file and stored in a WrapperFile object. It can be accessed through the *getWrapperData()* method of the WrapperFile object.
The WrapperData class allows access to the list of the wrapper's input and output files with the *getFileList()* method. The list of variables to be substituted in these files can be accessed with *getVariableList()*.

©2005-2012 EDF - EADS - PhiMeca

This data cannot be directly transferred to the wrapper, whose interface is written in C. Therefore, equivalent methods are provided to return this information as a C chained list. These methods are respectively known as *getNewFileListForCInterface()* and *getNewVariableListForCInterface()*. As their name indicates, these methods dynamically allocate the objects, which will thus have to be deallocated once they become useless. To achieve this, two methods are provided, namely *freeFileListForCInterface()* and *freeVariableListForCInterface()*. Though the data is sent to the wrapper, it is not up to the wrapper to ensure that the memory is deallocated. It is therefore up to the wrapper's caller to create and free the appropriate memory space.

XMLStringConverter When an XML parser is used, it is necessary to convert character strings read in the file by the parser into String elements. This conversion is the reason why the XMLStringConverter class exists. The constructor reads the XML string and the operator *operator String()* converts it into a String.

XMLWrapperErrorHandler This class allows to implement an error handler for the XML parser integrated in the Open TURNS platform. The parser automatically invokes it when an error occurs, to ensure that the error is correctly handled.

Base type

The base types of the Open TURNS platform are grouped into a package which corresponds to classes stored in the Base/Type sub-folder of the source tree. This package includes higher level classes which have a meaning in the context of statistics. They will be the foundation for the definition of the tool's data model.

Collection Figure 3.14 shows the definition of the Collection class as a class deriving from std::vector. The Collection represents an ordered set of homogeneous objects. Thus an iterator can traverse it, the user can interrogate its size, elements can be added or removed, etc.
As the std::vector type, the Collection class is a template that will have to be instantiated with the collection's element type. Figure 3.14 shows a DistributionCollection sub-class instantiated with the Distribution type.

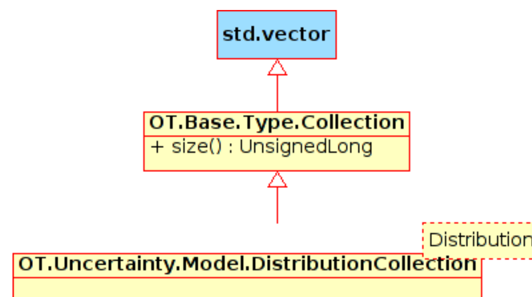


Figure 3.14: Collection and DistributionCollection classes.

NumericalPoint The NumericalPoint class implements the notion of numerical point. It is an ordered collection of numerical scalars representing the coordinates of a point in a space of any given dimension ($\Re^n$), where n is the point's dimension. NumericalPoint's *dim()* method returns this dimension. *Note that NumericalPoint does not inherit the size() method from Collection.* Neither can a NumericalPoint's dimension be modified the way a Collection's size attribute can. A NumericalPoint's dimension is defined once and for all when the object is created.
The operator *operator[]()* is provided to access the NumericalPoint's coordinates. It returns a reference to the coordinate.

©2005-2012 EDF - EADS - PhiMeca

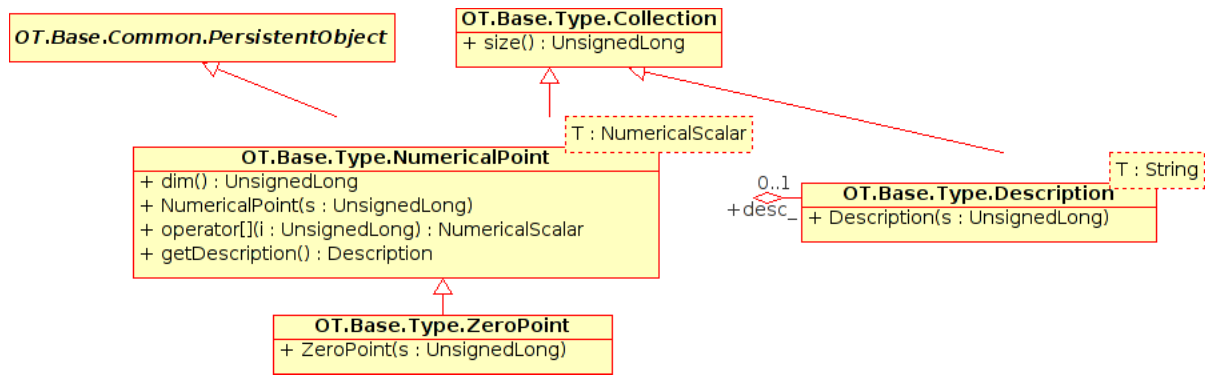


Figure 3.15: Collection, NumericalPoint, ZeroPoint and Description classes.

The *getNorm()* method returns the point's norm.
The NumericalPoint can be associated to a Description which has mandatorily the same size and details each coordinate, for example by associating it with a meaningful name for the user. It can also have a specific name provided by the PersistentObject class from which it inherits.

ZeroPoint The origin point playing a distinctive and recurring role in modeling, a specific class is created to represent it and accelerate computations. Functionnally speaking, it is a NumericalPoint; however, all of its coordinates have a zero value and they cannot be modified.

Description The Description is a String collection which can be associated to any object of a multi-dimensional nature. It allows to describe the components of the entity it is associated with by giving each one a name (or a comment, etc.) that is meaningful for the platform user. No processing can be carried out on the basis of the description which is only available for informative purposes.
However, Descriptions can be shared between objects of the same nature: for example, all of the NumericalPoints of a given sample and the sample itself may use the same Description.

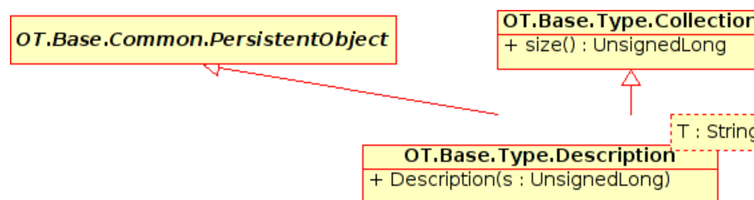


Figure 3.16: Collection and Description classes.

Matrix The Matrix class (see Figure 3.17) implements the notion of numerical scalar matrix, which is a two-dimensional ordered Collection of NumericalScalars. Its size is fixed when the object is created and it cannot be modified later on. The *getNbRows()* and *getNbColumns()* methods respectively return the matrix's number of rows and its number of columns.
As in all matrices, the operator *operator()* gives access to the numerical scalar designated by the row and column indexes passed as arguments.
The *transpose()* method returns the transposed matrix of the original matrix.

©2005-2012 EDF - EADS - PhiMeca

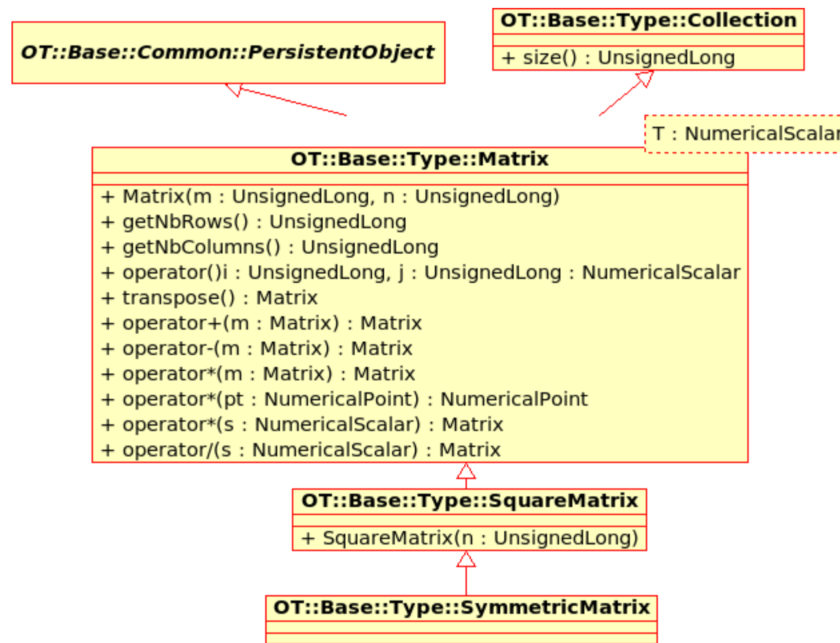


Figure 3.17: Collection, Matrix, SquareMatrix and SymmetricalMatrix classes.

The Matrix class supports an elementary algebra for which the following operations are defined: addition and substraction (with another Matrix), product (with another Matrix, with a NumericalPoint, with a NumericalScalar), division (with a NumericalScalar). The set of algebraic operations may be extended later on. These operations all rely on the underlying external library LAPACK.

SquareMatrix The SquareMatrix class derives from Matrix and restricts the behavior of the Matrix class to the case of square matrices. Specifically, the SquareMatrix constructor only accepts one argument.

SymmetricMatrix As described in Figures 3.17 and 3.18, the SymmetricMatrix class specializes SquareMatrix for the symmetric matrices. Specifically, the operator *operator()* is implemented so as to compel the user to create a symmetric matrix: the matrix is stored in an upper triangular form, and only the elements above the diagonal are read or modified. This ensures that the matrix is constantly consistent with its definition.

CorrelationMatrix The CorrelationMatrix class specializes SymmetricMatrix for the correlation matrices. The distinctive definition rules for these matrices require a specific method to ensure that, if modified, the matrix elements' values still obey these rules: the value of the diagonal elements must always be 1, the value of non-diagonal elements belong to the $[-1, 1]$ interval, etc.

FictiveCorrelationMatrix The FictiveCorrelationMatrix is a template class deriving from CorrelationMatrix, whose constructor implements mechanisms to convert different types of correlation matrices (KendallCorrelationMatrix, SpearmanCorrelationMatrix, etc.) into a fictive correlation matrix. The conversion mechanism is defined in a FamilyPolicy class.

KendallCorrelationMatrix The KendallCorrelationMatrix class is a CorrelationMatrix whose coefficients are Kendall correlation coefficients. This class will be developed in a future version of the platform.

©2005-2012 EDF - EADS - PhiMeca

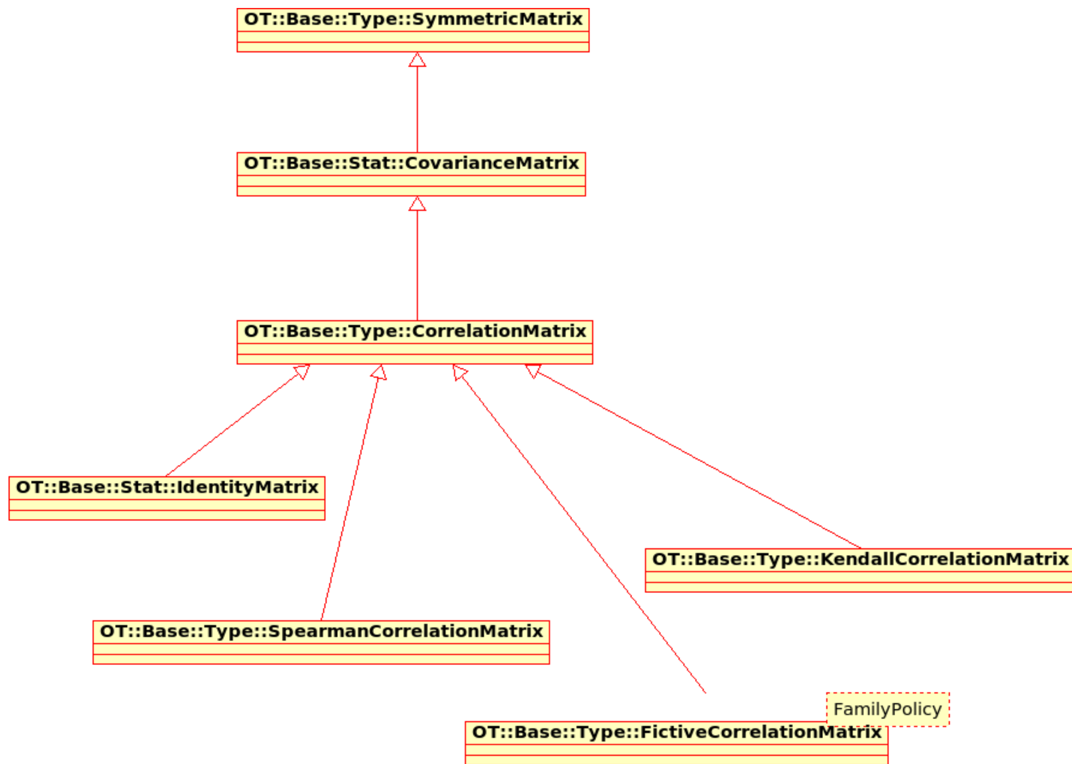


Figure 3.18: SymmetricMatrix, CovarianceMatrix, IdentityMatrix, CorrelationMatrix, and related classes.

SpearmanCorrelationMatrix The SpearmanCorrelationMatrix class is a CorrelationMatrix whose coefficients are Spearman correlation coefficients. This class will be developed in a future version of the platform.

Tensor The Tensor class described in Figure 3.19 is a three dimensional ordered collection of numerical scalars. It implements the concept of mathematical tensor. To this date, no algebra is defined on this class.

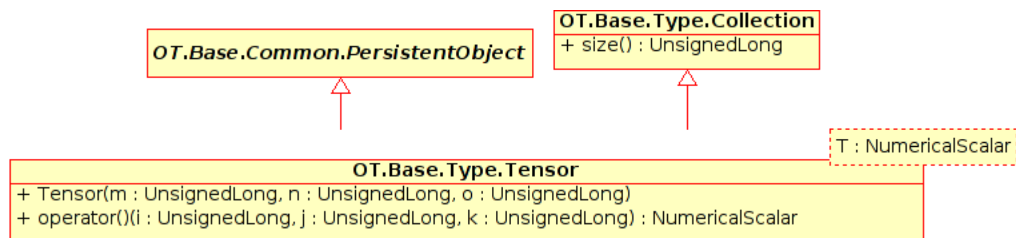


Figure 3.19: Collection and Tensor classes.

FamilyPolicy Figure 3.20 describes the abstract FamilyPolicy class which defines an interface for all of the correlation matrix conversions. It is implemented as a template instantiated for each conversion operating on a pair of correlation matrices.
The *transform()* is applied element by element on the intial matrix to produce the final matrix (generally a fictive matrix).

©2005-2012 EDF - EADS - PhiMeca

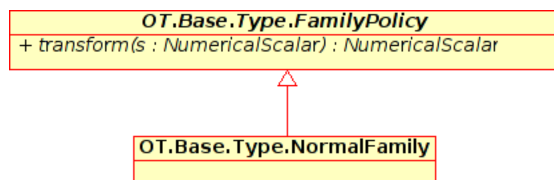


Figure 3.20: FamilyPolicy and NormalFamily classes.

NormalFamily The NormalFamily class specializes the FamilyPolicy class for Gaussian fictive matrices. It implements several definitions for the *transform()* method depending on the original matrix.

Basic numerical functions

This package encompasses classes implementing all of the mechanisms linked to the numerical functions of the Open TURNS platform.
These elements are grouped in the Base/Func sub-folder of the source tree.

NumericalMathFunction The Open TURNS platform offers a definition for the numerical function (on this subject, refer to section 2.3.4) implemented as a class called NumericalMathFunction. It is an abstract class, thus defining a common interface for all of the tool's numerical functions. Therefore, it is derived into several sub-classes implementing different types of functions (see Figure 3.21 and the following paragraphs).
The NumericalMathFunction class defines a pure virtual operator *operator()* which returns the result of the function call for a given point.
The NumericalMathFunction can be interrogated to retrieve the numerical point dimentions used for input and output with the *inNumericalPointDim()* and *outNumericalPointDim()*.
The pure virtual methods *gradient()* and *hessian()* respectively return the function's gradient matrix and hessian tensor at the given point.

AnalyticalNumericalMathFunction The AnalyticalNumericalMathFunction class derives from NumericalMathFunction and implements an analytical function directly coded in the Open TURNS platform.
Among these functions can be found all of the standard mathematical functions whose analytical expression as well as its gradient and hessian are known, commonly used in uncertainty treatment studies (e.g. sin, cos, exp, etc.).

TabulatedNumericalMathFunction The TabulatedNumericalMathFunction class derives from NumericalMathFunction and implements a function whose values are known only for given intervals, generally in an empirical fashion. This type of function is designed either to have its values directly created by the user through the platform's GUI or to be read from a description file.
The values taken by the function are given as a collection of values when the object is created and they cannot be modified later on. It is conceivable to design a factory allowing to create such functions by reading a stream using different formats.

ComputedNumericalMathFunction The ComputedNumericalMathFunction class derives from NumericalMathFunction and represents a function implemented within a compiled code linked to the Open TURNS platform. It is here that the connection with external code occurs and the ComputedNumericalFunction class virtualizes the external code into a mathematical function.

©2005-2012 EDF - EADS - PhiMeca

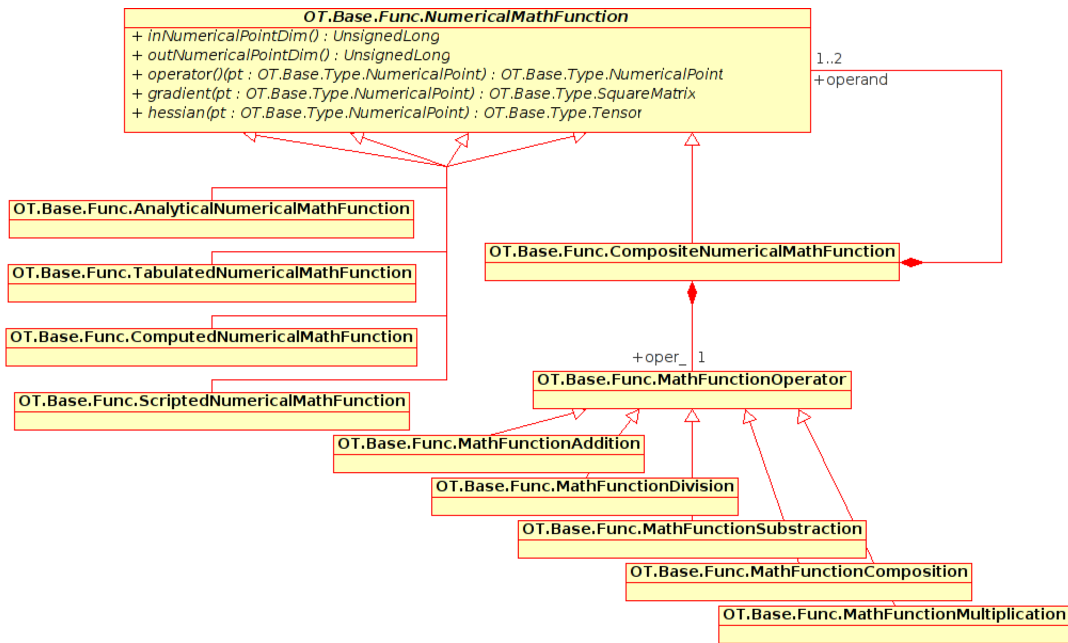


Figure 3.21: NumericalMathFunction and related classes, MathFunctionOperator and related classes.

The external code is not directly linked to the ComputedNumericalMathFunction operator *operator()* but rather to an intermediary object that establishes the programming link between the code and the mathematical function.

ScriptedNumericalMathFunction The ScriptedNumericalMathFunction class derives from NumericalMathFunction and represents a function implemented within the integrated Python interpreter of the Open TURNS platform. This function can be called as well from the Python interpreter in which it is defined as from the C++ code, yet without offering the same performance level as a C++ coded function.

CompositeNumericalMathFunction The Open TURNS platform intends to provide for the combination of numerical functions with an elementary algebra allowing to add, substract, multiply, divide and compose numerical functions. The CompositeNumericalMathFunction class is where this combination takes place. It points to one or two numerical functions (type NumericalMathFunction) which behave as operands and play symmetric or dissymmetric roles (depending on the operation). Figure 3.21 clearly shows the Composite design pattern (see section 3.2.4) where the AnalyticalNumericalMathFunction, TabulatedNumericalMathFunction, ComputedNumericalMathFunction, etc. classes represent the tree's leaves.
With this Composite pattern, it is possible to represent any combination of functions linked with unary or binary operators. This mechanism allows to set up chains of external codes within the Open TURNS platform.

MathFunctionOperator Figure 3.21 shows that the CompositeNumericalMathFunction class is related to an abstract MathFunctionOperator class representing the operator to be applied on the combination of numerical functions. This operator must be derived into sub-classes actually implementing the expected combinations.

MathFunctionAddition The MathFunctionAddition class derives from MathFunctionOperator and implements the sum of two numerical functions. It checks that both numerical functions return NumericalPoints of

©2005-2012 EDF - EADS - PhiMeca

the same dimension, and raises an exception if it is not the case when creating the CompositeNumericalMathFunction object.

MathFunctionSubstraction The MathFunctionSubstraction class derives from MathFunctionOperator and implements the substraction of two numerical functions. It checks that both numerical functions return NumericalPoints of the same dimension, and raises an exception if it is not the case when creating the CompositeNumericalMathFunction object.

MathFunctionMultiplication The MathFunctionMultiplication class derives from MathFunctionOperator and implements the product of a numerical function by a numerical scalar (NumericalScalar).

MathFunctionDivision The MathFunctionDivision class derives from MathFunctionOperator and implements the division of a numerical function by a numerical scalar (NumericalScalar). It raises an exception in case of a division by zero when invoking the function or one of its methods.

MathFunctionComposition The MathFunctionComposition class derives from MathFunctionOperator and implements the composition of two numerical functions. It checks that both numerical functions return / uses NumericalPoints of compatible dimensions, and raises an exception if it is not the case when creating the CompositeNumericalMathFunction object.

Library This class abstracts access to dynamic libraries loaded in the platform by hiding the operations into an object. The Library object thus retains the reference to the dynamic library and allows to search for symbols exported by this library with the *getSymbol()* method.
Objects from the Library class cannot be directly instantiated. They are created by a LibraryLoader object.

LibraryLoader The LibraryLoader class provides a factory mechanism to produce Library objects. The LibraryLoader class also provides a singleton mechanism through the *getInstance()* static method. Once the LibraryLoader object is retrieved, it is possible to load a library with its *load()*. This method returns a Library object representing the library, which allows access to the symbols exported by the library.
The LibraryLoader object is multithread-safe.

NumericalWrapperFunction The NumericalWrapperFunction class implements general mechanisms allowing the use of an external wrapper for access to a numerical function located in an external code. Access to the wrapper relies on a loaded dynamic library.
Since the wrapper can be called concurrently by the platform, a state management mechanism must be provided and the state needs to be stored in the ComputedNumericalMathFunction object. This state, whose internal representation is specific to the wrapper, can be accessed through the *createNewState()* and *deleteState()* methods. It is entirely handled by the ComputedNumericalMathFunction object.
The NumericalWrapperFunction object defines an interface to access the wrapper with the following pure virtual methods:

- *getInNumericalPointDimension()*: access to the dimension of the function's input vector;
- *getOutNumericalPointDimension()*: access to the dimension of the function's output vector;
- *initialize()*: initializes the wrapper before the first access to the function occurs;
- *execute()*: executes the external code function: the input vector is passed to the function and the output vector is retrieved;

©2005-2012 EDF - EADS - PhiMeca

- *finalize()*: finalizes the wrapper after the last access to the function occurs.

The NumericalWrapperFunction object is built based on the path to a library, a symbol anme and initialization data for the wrapper. Parsing the wrapper's description file (cf. WrapperFile et WrapperData) provides all of the necessary data.

UsualNumericalWrapperFunction The UsualNumericalWrapperFunction class derives from NumericalWrapperFunction and implements the interface it defines, namely the getInNumericalPointDimension(), getOutNumericalPointDimension(), initialize(), execute() and finalize() methods.

Statistical objects

This package encompasses classes offering higher-level statistical functionalities in the Open TURNS platform. This package integrates notions such as numerical sample, covariance matrix, etc.
These elements are grouped into the Base/Stat sub-folder of the source tree.

CovarianceMatrix The CovarianceMatrix class, described in Figure 3.18, specializes SymmetricMatrix for covariance matrices.

IdentityMatrix The Identity matrix playing a distinctive, recurrent role in modeling, it is associated to a specific class IdentityMatrix in order to improve performance. From a functional point of vew, it is a SymmetricMatrix; the value of its diagonal elements is 1, the value of its non-diagonal elements is 0, and the elements cannot be modified.

NumericalSample The NumericalSample class described in Figure 3.22 is an abstract class representing a one-dimensional ordered collection of numerical points (NumericalPoint objects). Thus it defines an interface for all of the classes deriving from it.

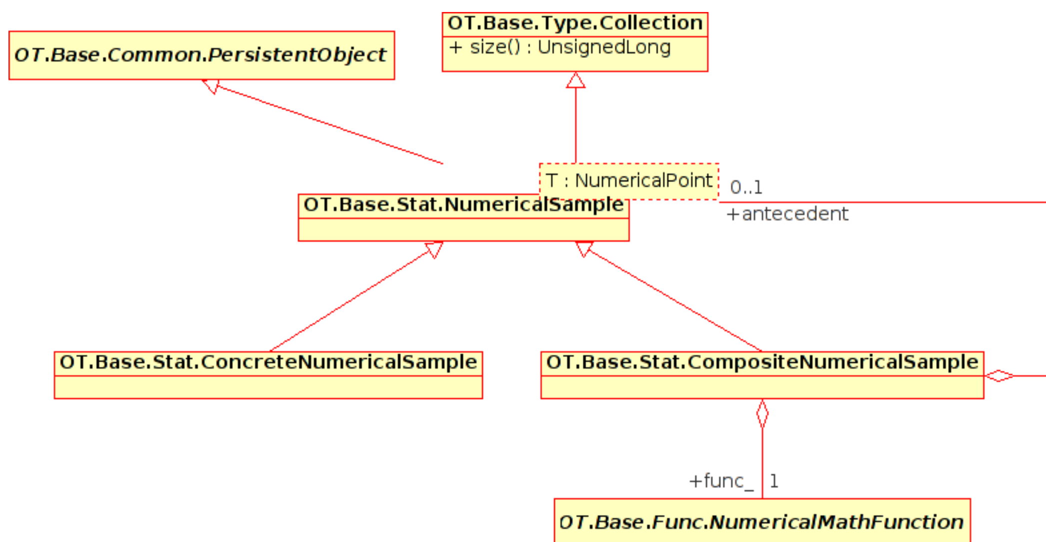


Figure 3.22: Collection, NumericalSample, and related classes.

Objects from the NumericalSample class provide a pure virtual *getMeanValue()* which returns a numerical point giving the mean value of the sample's points.

The pure virtual method *getCovariance()* returns the sample's covariance matrix.
The operator *operator()* gives access to the sample's individual points.
Figure 3.22 shows that the NumericalSample class matches a Composite design pattern as described in section 3.2.4.

ConcreteNumericalSample The ConcreteNumericalSample represents a leaf in the Composite pattern. It derives from the abstract class NumericalSample and defines instantiable samples on which it is possible to carry out statistical treatments.
Points can be added or removed from the collection with both methods *add()* and *remove()*. It is the standard way for the user to handle the sample's content.

CompositeNumericalSample The CompositeNumericalSample class derives from the abstract NumericalSample class and defines a sample whose elements are based on another sample and a numerical function (NumericalMathFunction). It takes part in the implementation of a Composite pattern and sets up the chain between numerical samples.
When a CompositeNumericalSample is created, the collection contains no element. These are determined on demand when methods are applied to the sample. They result from the call of the numerical function on the elements of the antecedent sample. Since the amount of elements can be very high, it is compulsory to consider a parallel processing of the calls within the CompositeNumericalSample class.

NumericalSampleFactory The content of a numerical sample can be directly provided by a file, a stream, etc. These possibilities are not directly cared for by objects from the NumericalSample class by a distinct class hierarchy of NumericalSampleFactory objects implementing a Factory pattern (see section 3.2.2).

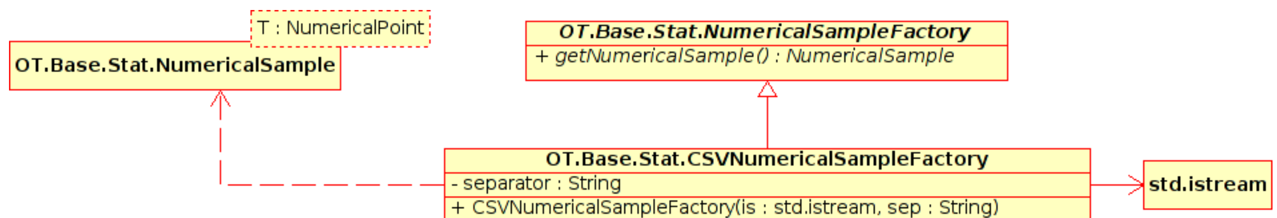


Figure 3.23: NumericalSampleFactory, CSVNumericalSampleFactory, and NumericalSample classes.

The abstract NumericalSampleFactory class defines an interface that all sub-classes must implement.
The pure virtual *getNumericalSample()* method returns a numerical sample (NumericalSample).

CSVNumericalSampleFactory The Open TURNS platform is intended to allow the use of CSV (Comma Separated Values) files as sources for the content of a numerical sample.
Objects from the CSVNumericalSampleFactory class are created with a stream initially pointing to the file storing the sample, and with the optional indication of a delimiter used to parse the file. This delimiter is a String that may contain a regular expression (such as those defined in [BOO]).

3.3.2 Field-specific brick

This brick encompasses all of the modeling, propagation and prioritization classes needed for the treatment of uncertainties.
All of this brick's elements are included in the Uncertainty/ sub-folder of the source tree.

©2005-2012 EDF - EADS - PhiMeca

Data modeling

This package contains all of the classes needed to carry out an uncertainty treatment study as defined by the Open TURNS platform's goals. The objects in this package can be directly handled by the user, as well with the TUI as with the GUI (see following sections). What this package essentially defines are not any concrete objects, but rather a set of class interfaces that provide user with a unified view.
These elements are grouped into the Uncertainty/Model/ sub-folder of the source tree.

Distribution The distribution (Distribution) is the core type in uncertainty modeling. It is described in Figure 3.24. It is an abstract class declaring an interface implemented by the derived UsualDistribution, AssemblyDistribution, Mixture, and FunctionalDistribution classes.

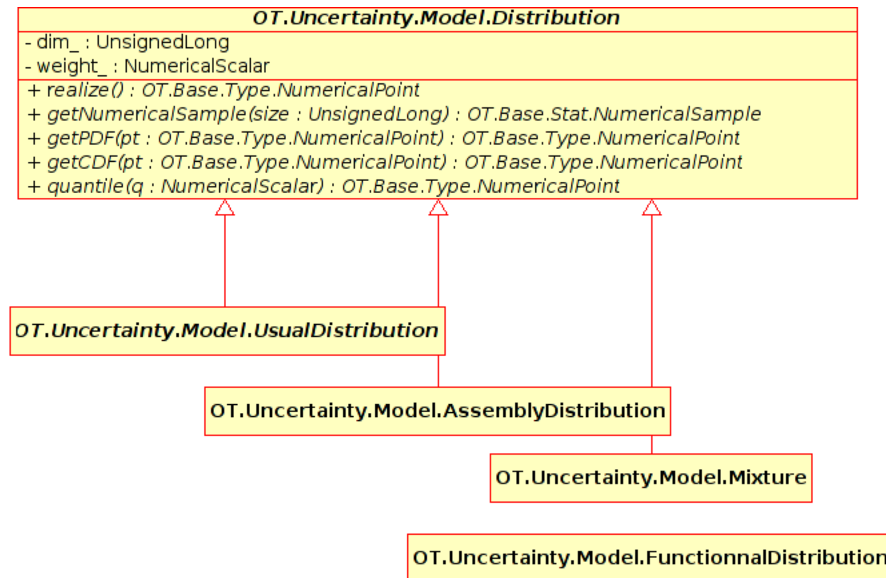


Figure 3.24: Distribution, UsualDistribution, AssemblyDistribution, Mixture, and FunctionalDistribution classes.

The distribution is a multi-dimensional object within the Open TURNS platform, even though the user has direct access only to given dimensions. As it turns out, some distributions may not exist for all possible dimensions of the vectorial space that the user is interested in. This results in an exception being raised for any unimplemented case.
The dimension of a distribution can be accessed with the *dim()* method. Once the distribution is created, it cannot be modified.
A distribution can produce a realization value and return a NumericalPoint with the pure virtual *realize()* method. It can also directly produce a numerical sample of a given size with the pure virtual *getNumerical-Sample()* method.
The distribution's probability density function and cumulative distribution function can respectively be retrieved with the *getPDF()* and *getCDF()* methods.
The *quantile()* method returns the quantile of a distribution. It has to be noted that this method is only implemented (and only makes sense) for 1-dimensional distributions.
The *weight* parameter is described in the Mixture section (later on in this section).

UsualDistribution The UsualDistribution class derives from the abstract Distribution class and defines an interface for all usual distributions supported by the Open TURNS platform. Usual distribution means any

©2005-2012 EDF - EADS - PhiMeca

distribution for which either the analytical form or an evaluation method is known, that can be defined by the user without any information other than the distribution's parameters.
The UsualDistribution class declares a pure virtual method *getKernel()* which returns a Kernel object matching the distribution's kernel. If a class deriving from UsualDistribution has no kernel, an exception is raised.
All distributions deriving from the UsualDistribution class are declared in an independent package, namely OT::Uncertainty::Distribution.

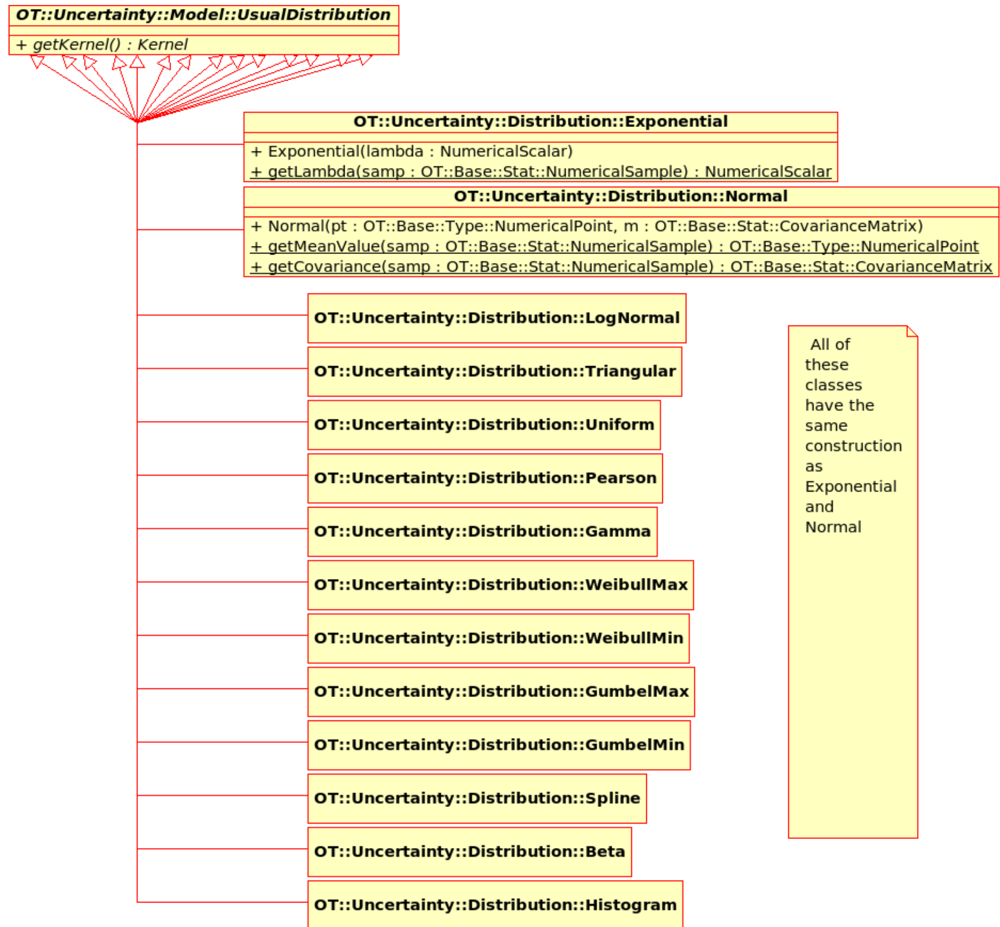


Figure 3.25: UsualDistribution and related classes.

AssemblyDistribution The AssemblyDistribution class, described in Figure 3.26, derives from the abstract Distribution class and implements the distribution assembly mechanism. To achieve this, it relies on a distribution collection and on a copula which defines the correlation between the collection's distributions. The distribution order in the collection is relevant: it determines the order of the assembly distribution components. The copula's dimension must match the number of components in the distribution collection, which will also the dimension of the final AssemblyDistribution object.

Mixture The Mixture class derives from the abstract class Distribution and implements the mechanism of mixture distribution (i.e. linear combination of distributions). The analysis showed that it was necessary to draw upon the concept of weighted distribution in order to represent the product of a distribution by the distribution's weight (a numerical scalar) within the linear combination. So as to not introduce another specific

©2005-2012 EDF - EADS - PhiMeca

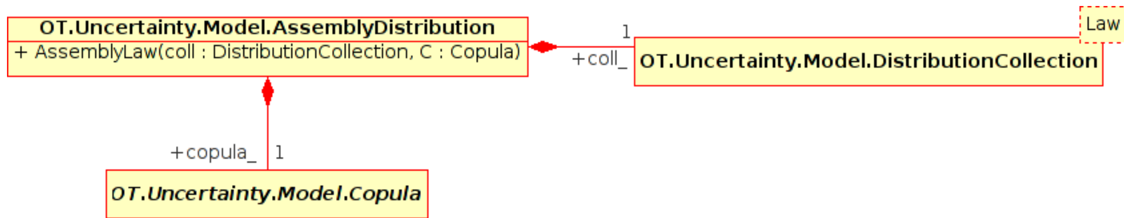


Figure 3.26: AssemblyDistribution, DistributionCollection and Copula classes.

class to support the distribution / numerical scalar pair, and since this concept is only used in the platform in this very context, it is not a problem to have the Distribution class directly support this weight (attribute weight).

Figure 3.27: Mixture and DistributionCollection classes.

Thus the Mixture class allows to create a linear combination of any distributions, whose weight must be set with the Distribution's *setWeight()* method. The *getWeight()* allows to retrieve this value.

FunctionalDistribution The FunctionalDistribution class derives from the abstract Distribution class and implements the transformation mechanism of a distribution through a numerical function. The antecedent distribution belongs to a random vector passed as an argument to the functional distribution when it is created. This class encompasses all of the operations the CompositeRandomVector must support.

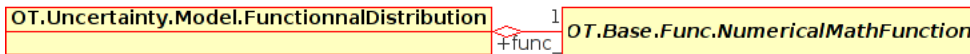


Figure 3.28: FunctionalDistribution and NumericalMathFunction classes.

DistributionCollection The distribution collection is implemented by the DistributionCollection class deriving from the Collection class and defining an ordered set of Distribution objects.

DistributionFactory The usual distributions (UsualDistribution) can be either directly instantiated by the user, or instantiate through a factory mechanism. The DistributionFactory class implements the latter mechanism according to the Factory pattern described in section 3.2.2, which shows two parallel class hierarchies. The abstract DistributionFactory class mirrors the UsualDistribution class, and the derived classes of each of these classes are symmetric.
DistributionFactory is an abstract class and, as such, requires to be derived into specific classes for each distribution family. This way, the concept of distribution factory covers the concept of distribution family. Anywhere where the distribution family appears in the analysis model, the distribution factory can be used as an implementation.
The pure virtual method *getDistribution()* generates a usual distribution belonging to the factory's family; to achieve this, it uses either a numerical sample passed as an argument, or a numerical point and a covariance matrix. However, there are different algorithms working out the parameters of the distribution object to be

©2005-2012 EDF - EADS - PhiMeca

created. These algorithms are determined by the distribution factory policy template parameter (DistributionFactoryPolicy) which is used to instantiate the distribution factory.
The *getKernel()* method allows to retrieve the Kernel object associated with the factory's distribution family.

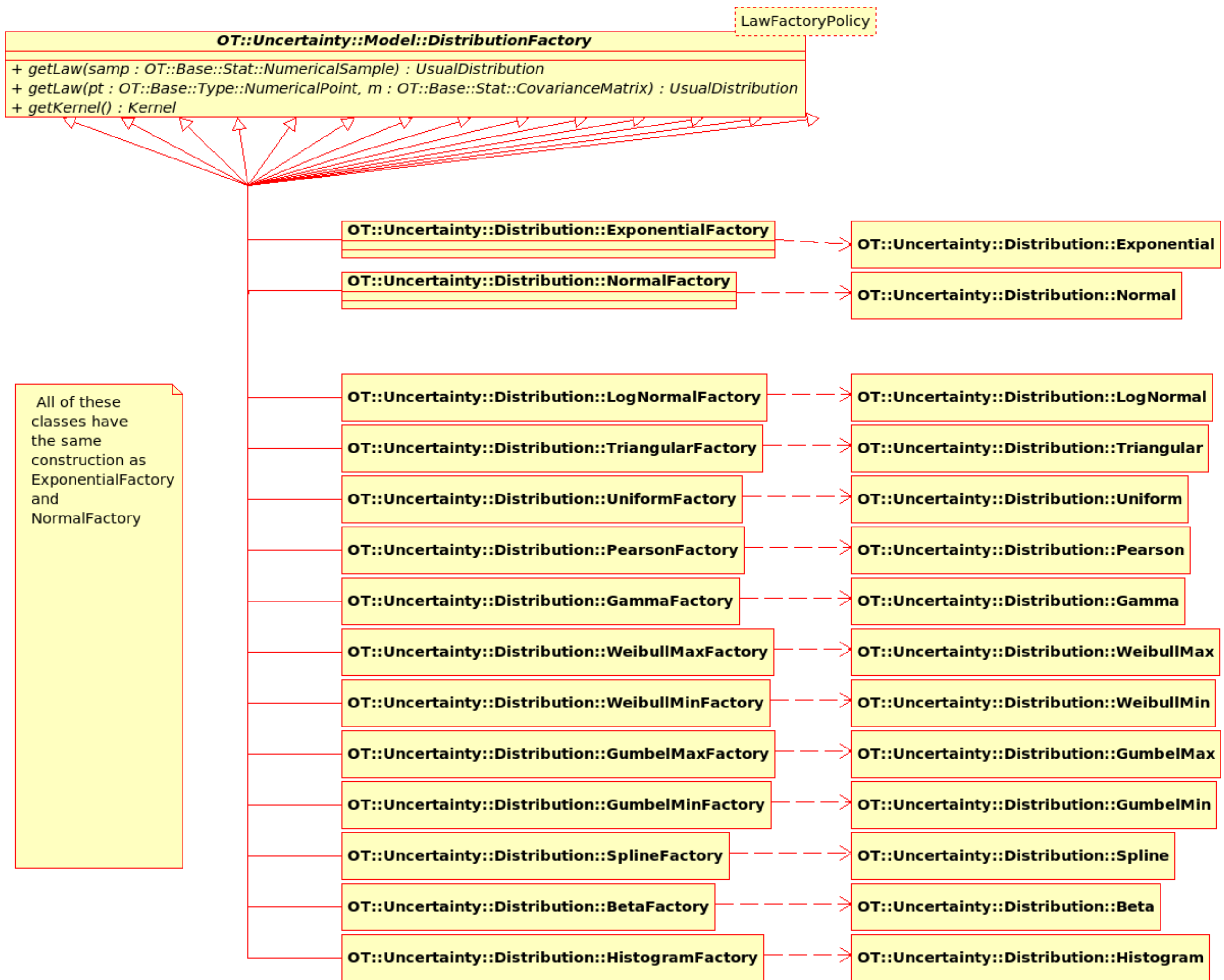


Figure 3.29: DistributionFactory and related classes.

DistributionFactoryPolicy As described in Figure 3.30 and in the previous section, the DistributionFactoryPolicy class defines an interface for the distribution factory policies.
The DistributionFactoryPolicy is derived into MaximumLikelihood and MomentMethod.

Kernel The Kernel interface gives access to the kernel defined for each usual distribution. Each usual distribution is supposed to return a Kernel object.
This interface defines the legal operations on Kernel objects, which are an *L2Norm()* method which returns the L2 norm of the kernel, and a *covariance()* method which returns the covariance matrix of the said kernel.
This Kernel interface is implemented into as many derived classes as there are usual distributions supported by the platform (see Figure 3.31). These derived classes all belong to the OT::Uncertainty::Distribution package.

©2005-2012 EDF - EADS - PhiMeca

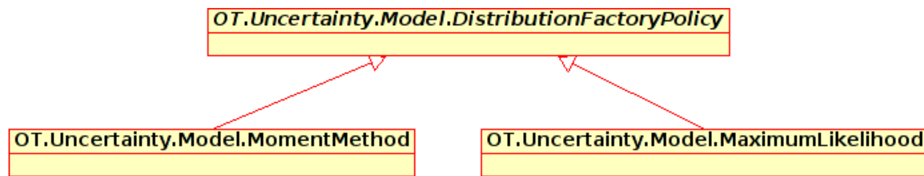


Figure 3.30: DistributionFactoryPolicy, MomentMethod, and MaximumLikelihood classes.

GenericRandomVector The GenericRandomVector, described in Figure 3.32, defines an interface for all of the tool's random vectors. Though central from the user's point of view and for the mathematical modeling of their study, the concept of random vector is secondary compared to the concept of distribution within the platform. The random vector is rather a concept related to distribution instantiation than an entity in its own right.
However, it differs from the distribution, since it allows to chain random vectors so as to model the dependency that may exist between them. Specifically, a random vector can be built from one or several other random vectors, which defines a dependency tree. This tree itself is modeled with the Composite pattern (see section 3.2.4).
The GenericRandomVector abstract class defines an interface which must be implemented by the derived classes RandomVector, ConstantRandomVector, CompositeRandomVector, etc.
The random vector implements the methods defined for the distribution. If needed, the execution of these methods can be delegated to an embedded object which offers the user the same interface as the distribution.
A Description may be added to the random vector; it informs the user of the nature of its components.
The *accept()* method allows to launch a RandomVectorVisitor visitor object on a RandomVector object or a derived object. The Visitor pattern is described in section 3.2.6.

RandomVector The RandomVector class, which derives from GenericRandomVector, implements a random vector that can be manipulated by the user. This random vector is defined by a joint distribution.
A random vector is by default built from a distribution (Distribution) which describes its behavior.

ConstantRandomVector The ConstantRandomVector, which derives from GenericRandomVector, implements a constant random vector, that is, a vector whose values are all associated with the same numerical point (NumericalPoint).
It is naturally built from a NumericalPoint.

CompositeRandomVector The CompositeRandomVector class, which derives from GenericRandomVector, implements a random vector defined on the basis of another random vector and a numerical function. This definition mechanism for random vectors allows the user to model the dependency between the vectors in their study while postponing the computation (the propagation) to a later stage.
The CompositeRandomVector is built on the basis of a GenericRandomVector and a NumericalMathFunction.
It implements a *getNumericalMathFunction()* method which returns the numerical function associated to the CompositeRandomVector. This function cannot be modified once the object is created.
It also provides a *getAntecedent()* method which returns the antecedent random vector on which the composite vector is based. As for the function, the antecedent cannot be modified once the object is created.

Copula The Copula class described in Figure 3.33 defines a generic interface for all of the copulas that can be implemented in the Open TURNS platform. The Copula class is an abstract class.

©2005-2012 EDF - EADS - PhiMeca

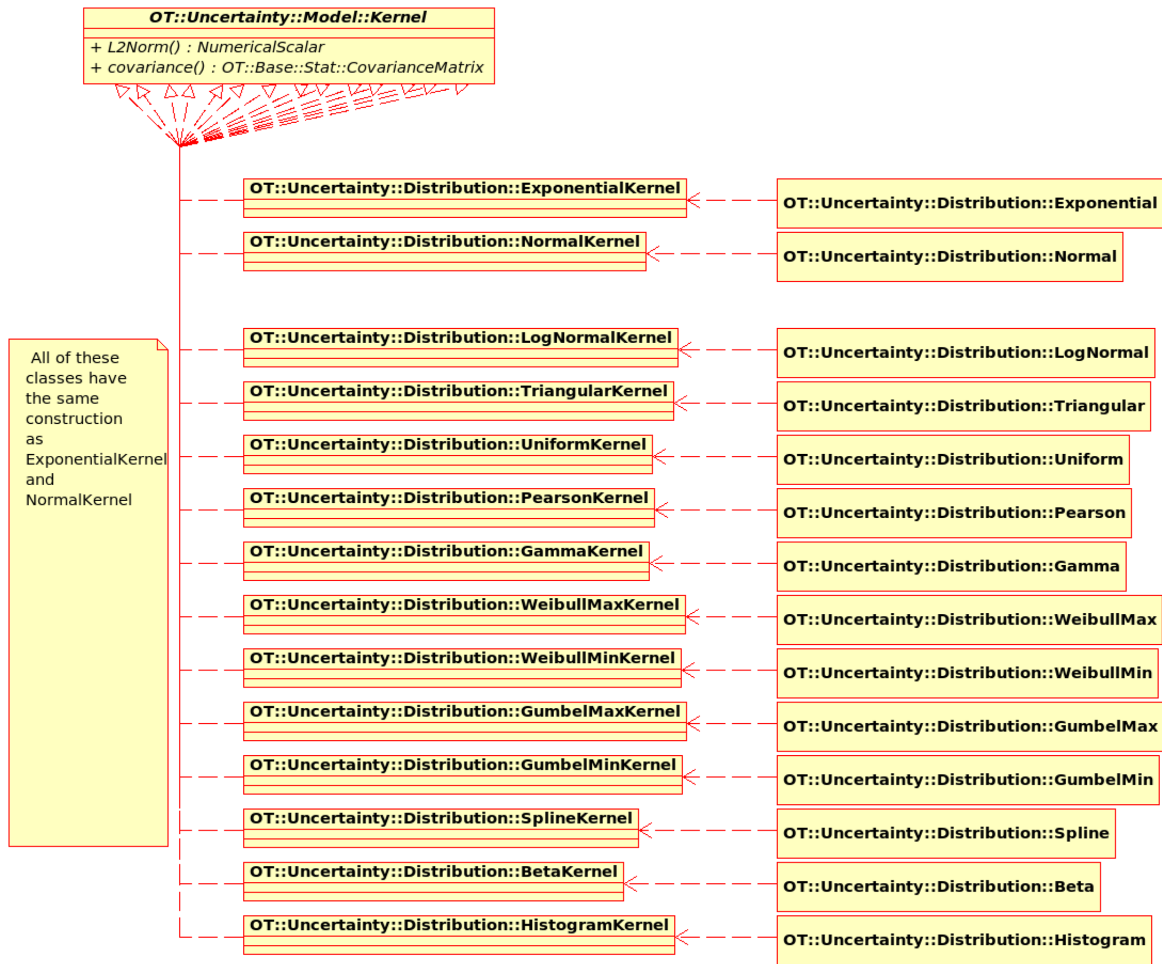


Figure 3.31: Kernel and related classes.

NormalCopula The NormalCopula class, which derives from Copula, implements a Gaussian copula.

FailureEvent The FailureEvent class, described in figures 3.32 and 3.34, is a specialization of GenericRandomVector. It implements the mechanisms related to a failure event.

The FailureEvent class introduces a second Composite pattern within GenericRandomVector's class hierarchy, similar to the one used for CompositeRandomVector.

A FailureEvent object is created from a random vector (GenericRandomVector), a comparison operator (ComparisonOperator), and a threshold (which is a numerical scalar). Because the failure event is itself a random vector whose dimension is 1, the antecedent random vector must have the same dimension. If it is not the case, an exception is raised during the object's creation.

The antecedent of the FailureEvent object can be accessed through the *getAntecedent()* method.

The comparison operator can be accessed through the *getOperator()* method.

The threshold value can be accessed with the *getThreshold()* method.

None of the values or objects used for the construction of the FailureEvent object can be modified once the object is created.

©2005-2012 EDF - EADS - PhiMeca

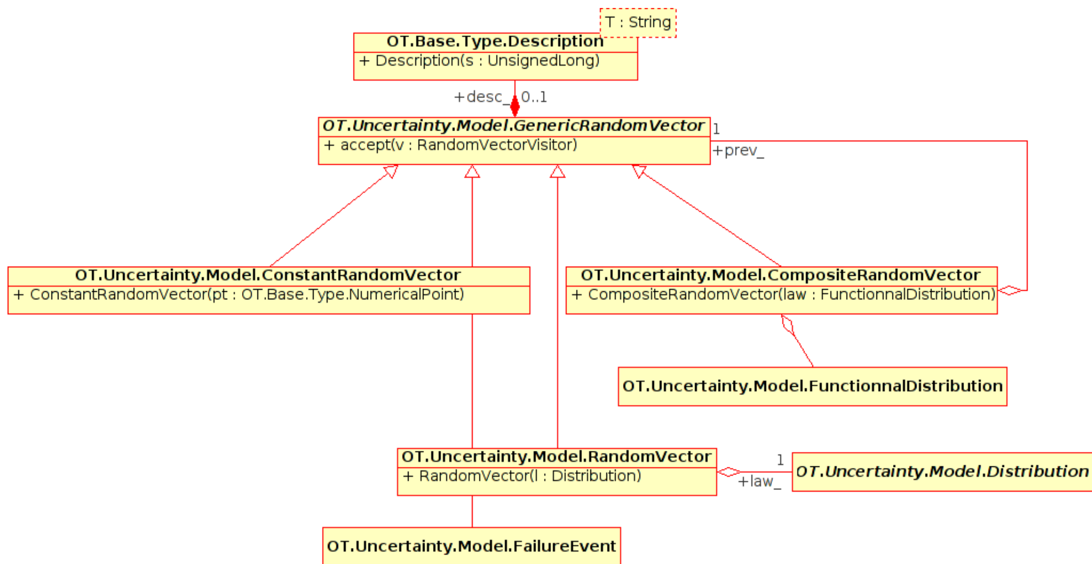


Figure 3.32: GenericRandomVector and related classes.

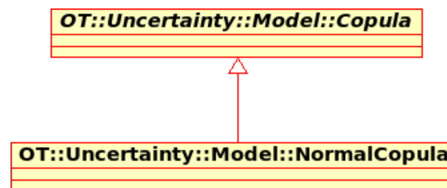


Figure 3.33: Copula and NormalCopula classes.

ComparisonOperator The FailureEvent objects rely on a comparison mechanism between the vector's value and a numerical scalar designated as the threshold. So as to be able to pass any type of comparison operator do the failure event, the Open TURNS platform provides an abstract class, ComparisonOperator, which defines the interface that any comparison operator must support.
Thus the class defines a pure virtual method *compare()* which compares two numerical scalars and returns a Bool whose value depends on the outcome of the comparison.
The ComparisonOperator class is derived into Less, LessOrEqual, Equal, GreaterOrEqual, and Greater.

Less The Less class, deriving from ComparisonOperator, implements the "less than" comparison.

LessOrEqual The LessOrEqual class, deriving from ComparisonOperator, implements the "less than or equal" comparison.

Equal The Equal class, deriving from ComparisonOperator, implements the "equal to" comparison.

GreaterOrEqual The GreaterOrEqual class, deriving from ComparisonOperator, implements the "greater than or equal to" comparison.

Greater The Greater class, deriving from ComparisonOperator, implements the "greater than" comparison.

©2005-2012 EDF - EADS - PhiMeca

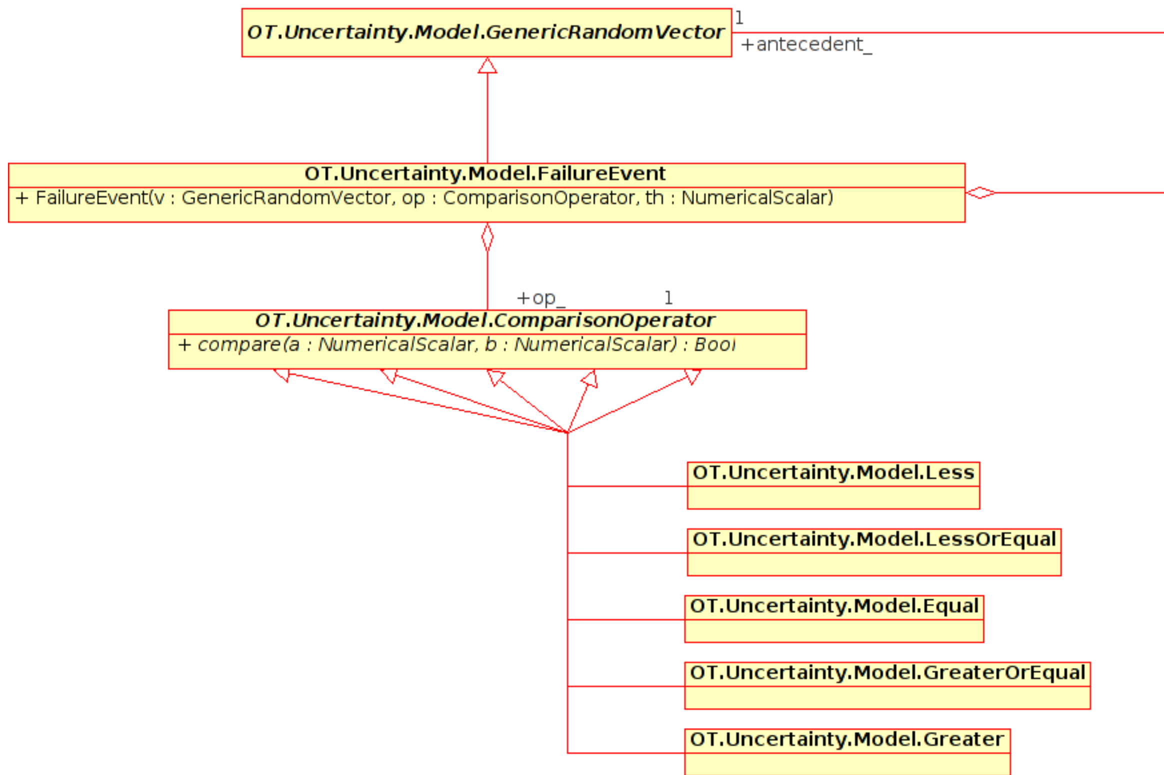


Figure 3.34: FailureEvent class.

MaximumLikelihood The MaximumLikelihood class, deriving from DistributionFactoryPolicy, implements an algorithm that computes the parameters of a usual distribution (UsualDistribution) using the method of maximum likelihood. This class is used to provide parameters for the DistributionFactoryPolicy instances.

MomentMethod The MomentMethod class, deriving from DistributionFactoryPolicy, implements an algorithm that computes the parameters of a usual distribution (UsualDistribution) using the method of moments. This class is used to provide parameters for the DistributionFactoryPolicy instances.

RandomVectorIterator The RandomVectorIterator naturally derives from Iterator. It allows to travers the dependency tree of a random vector by visiting each child of each vector. However, the dependency tree of a vector is an acyclic oriented graph by construction, thus one child may be visited several times by the iterator in case of multiple dependencies.

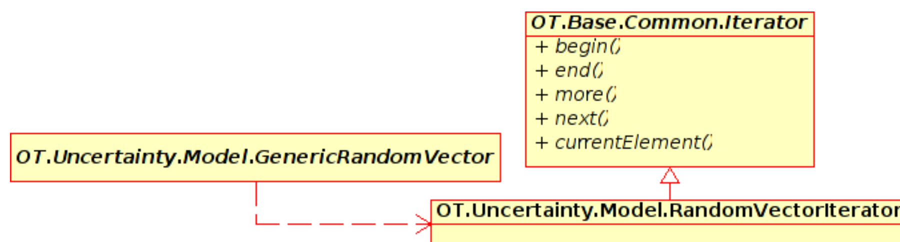


Figure 3.35: Iterator, RandomVectorIterator and GenericRandomVector classes.

©2005-2012 EDF - EADS - PhiMeca

OptimalWalkIterator The OptimalWalkIterator class derives from Iterator. Objects of this type are produced by an OptimalWalkVisitor which determines the optimal path to traverse the dependency graph of a random vector. During the traversal it controls, the object does not guarantee access to all of the graph's elements. Actually, it is quite the contrary: the goal of such an object is to reduce the tree so as to optimize a criterion determined by the OptimalWalkVisitor object.

RandomVectorVisitor The RandomVectorVisitor class defines an interface for all visitors of the class hierarchy stemming from GenericRandomVector. It implements the Visitor pattern described in section 3.2.6.

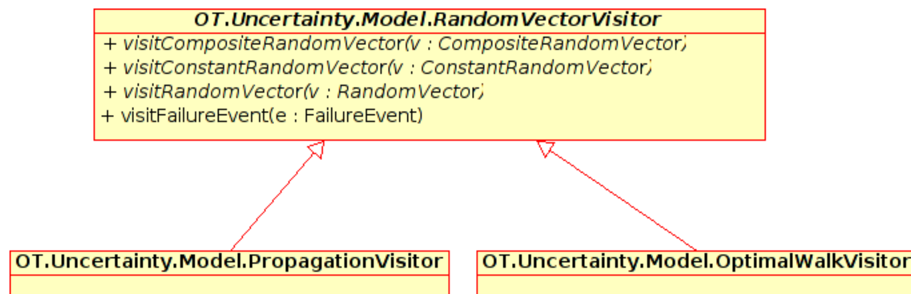


Figure 3.36: RandomVectorVisitor, PropagationVisitor and OptimalWalkVisitor classes.

The RandomVectorVisitor class is an abstract class. All of its methods are pure virtual methods and they must be re-defined in the derived classes.
The methods *visitCompositeRandomVector()*, *visitConstantRandomVector()*, *visitRandomVector()*, *visitFailureEvent()*, etc. are implemented according to the processing that must be performed for each type of random vector.

OptimalWalkVisitor The OptimalWalkVisitor class, which derives from RandomVectorVisitor, performs a complete traversal of the dependency tree of a random vector; this allows to determine the order in which the tree's vectors must be traversed so as to optimize a given criterion (internal to the visitor, such as limiting the memory usage, not re-evaluating a vector, etc.).
The OptimalWalkVisitor class generates an OptimalWalkIterator specific to the new traversal path.
This path can be used to ensure the propagation of uncertainties within the dependency tree, for example with the help of a PropagationVisitor.

PropagationVisitor The PropagationVisitor class, which derives from RandomVectorVisitor, allows to propagate uncertainties within the dependency graph of a random vector while relying on the optimal traversal determined by an OptimalWalkVisitor.

Distributions

These elements belong to the Uncertainty/Distribution/ sub-folder of the source tree.
The distributions described in this section are the usual distributions known to the platform. Table 3.1 describes the mapping between the distribution's name and the class prefix used within the Open TURNS platform.

©2005-2012 EDF - EADS - PhiMeca

Table 3.1: Mapping between distributions and classes

Usual name of the distribution	Class prefix (XDistribution)
Bêta	Beta
Exponential	Exponential
Gamma	Gamma
Gumbel	Gumbel
Histogram	Histogram
Log normal	LogNormal
Log uniform	LogUniform
Normal	Normal
Pearson	Pearson
Spline	Spline
Triangular	Triangular
Uniform	Uniform
Weibull	Weibull

XDistribution Each XDistribution implements a usual distribution as defined in the methodological reference of Open TURNS (see [Dut]). These distributions are objects that can be directly manipulated by the user. They can be created using a set of parameters, each parameter set specifically depending on the XDistribution. Each XDistribution also provides a *getKernel()* method which returns a Kernel object specific to XDistribution. This object cannot be modified: it is statically defined in the platform.

XDistributionFactory According to the Factory design pattern (see section 3.1), each XDistribution possesses a factory called XDistributionFactory, which allows to instantiate any distribution from the XDistribution class.
Each XDistributionFactory class defines a set of *getDistribution()* methods (which makes use of the exact same parameters defined for each XDistribution) as well as a *getKernel()* method which returns the Kernel object specific to XDistribution.

XDistributionKernel Each XDistribution should define a Kernel object specific to the distribution. Given the unicity and the specificity of this object on the one hand, and its state-less characteristic on the other, it is statically declared in the Open TURNS platform. The object XDistributionKernel defines the methods declared in the Kernel interface.

Simulation algorithms

This package encompasses all classes implementing the uncertainty propagation algorithms offered by the Open TURNS platform. These classes rely on a model of the problem created with the help of classes from the Model and Distribution packages. These elements belong to the namespace OT::Uncertainty::Algorithm.

UncertaintyAlgorithm Figure 3.37 shows the hierarchy of simulation algorithm classes, at the head of which is the abstract class UncertaintyAlgorithm. This class defines a common interface to all of the platform's algorithms.
The UncertaintyAlgorithm class derives from the Threadable class. Threadable defines a *run()* method within which the given algorithm must be implemented. Since a Thread object can handle it, such an algorithm can be used within the platform parallelly to other operations the user may want to carry out. However, this requires to handle the multi-threading aspects within the UncertaintyAlgorithm class in an appropriate manner,

©2005-2012 EDF - EADS - PhiMeca

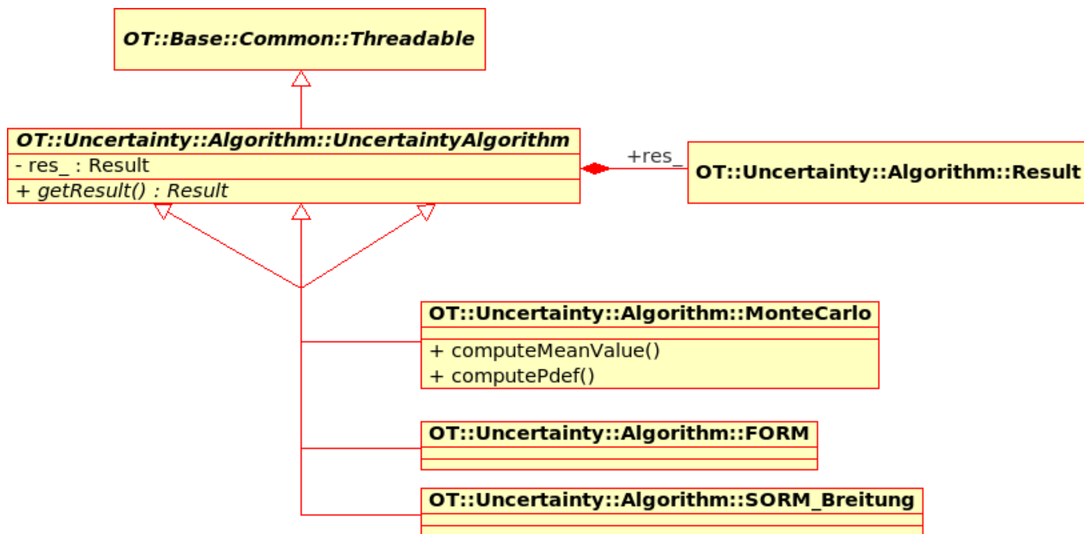


Figure 3.37: UncertaintyAlgorithm, Result, Threadable and related classes.

particularly as far as concurrent access to shared data (e.g. internal state) is concerned; if it is not dealt with appropriately, the risk is to produce inconsistent results, or even to crash the platform. Therefore (and unless explicitely mentioned), any of this class's methods must be multithread-safe.
When the computation taking place within the *run()* method is completed, the result can be accessed in a Result object returned by the pure virtual method *getResult()*.

Result The Result class defines the broad set of results that can be produced by a simulation algorithm. The default behavior is to create an empty Result object. It must be completed by adding data as the study progresses. Since access to this object can be performed within a multi-threaded procedure, the object needs to be protected from concurrent access. It possesses an internal lock to which its accessor must refer.
Details about the data that can be stored in a Result object will be given in a future version of this document.

MonteCarlo The Monte-Carlo class derives from UncertaintyAlgorithm and implements a Monte-Carlo algorithm to propagate uncertainties. It is described in Figure 3.38.

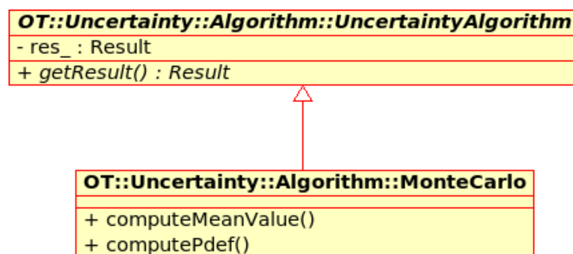


Figure 3.38: UncertaintyAlgorithm and MonteCarlo classes.

It defines several methods indicating which data should appear in the final Result object.
The *computeMeanValue()* method triggers the computation of the vector's mean value.
The *computePdef()* method triggers the computation of the event's probability of failure (the event being a vector).

©2005-2012 EDF - EADS - PhiMeca

FORM The FORM class derives from UncertaintyAlgorithm and implements a FORM algorithm to propagate uncertainties. It is described in Figure 3.39.

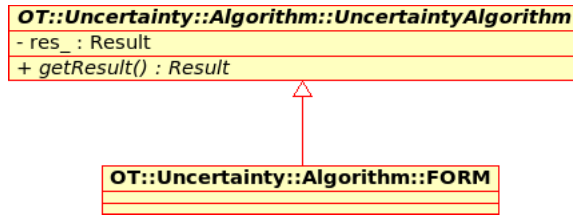


Figure 3.39: UncertaintyAlgorithm and FORM classes.

It defines several methods indicating which data should appear in the final Result object.

SORM_Breitung The SORM_Breitung class derives from UncertaintyAlgorithm and implements a SORM algorithm to propagate uncertainties. It is described in Figure 3.40.

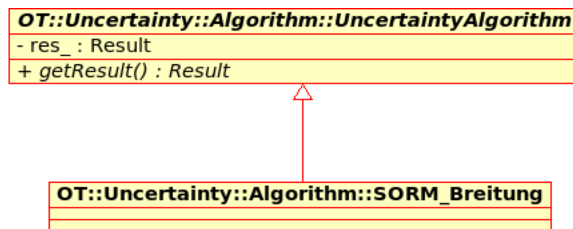


Figure 3.40: UncertaintyAlgorithm and SORM_Breitung classes.

It defines several methods indicating which data should appear in the final Result object.

©2005-2012 EDF - EADS - PhiMeca

Chapter 4

Technical architecture

This chapter details the technical elements required by the Open TURNS platform, namely the system requirements, the tools and the development environment of the project.

4.1 Target platforms

The Open TURNS platform is meant to carry out uncertainty treatment studies in a scientific environment. Most of the scientific codes being available on Unix platforms, Open TURNS is naturally designed to run on this family of systems. Unix being a standard with multiple implementations, available on different architectures, this gives a wide choice of target platforms.
Linux is currently the most attractive Unix system for the Open TURNS project, it was chosen as the main target system for the project's development as well as for the delivery of the different versions.
The partners involved in the project have each chosen different Linux distributions, for technical and historical reasons. Therefore, it was decided to support several distributions, a choice that should not be seen as final or minimal. The distributions considered here include for example the list given in Table 4.1

Table 4.1: Examples of Linux distributions supported by the project's partners

Linux distribution	Version
Debian	Sarge
Mandriva	2005

However, there are also uncertainty treatment studies carried out in the proprietary Windows environment. While this system is not identified as a target for the project, developments should be carried out so as to facilitate the port to Windows.

4.2 Namespace

All the classes of the OpenTURNS library are accessible within a single namespace named OT and aliased as OpenTURNS. It allows to insulate these classes from classes from another project that could share the same name.

4.3 Internationalization

The Open TURNS platform is meant to be widely distributed within the scientific community revolving around probability and statistics, which is essentially an international community. Therefore, the platform should be designed so as to be adjustable to the users, particularly those who do not speak English[1].
This involves not using any messages directly in the source code of the platform, but rather to create a resource catalogue that can be loaded, according to the locale setting of the user, when the application is launched.
Another consequence of internationalization is the need for the Unicode extended character set (see [UNI]) to be used for all strings.

4.4 Accessibility

The Open TURNS platform shall be accessible to disabled users. This has implications on the ergonomy and the design of the User Interface, particularly the GUI which should offer keyboard shortcuts for any available function as well as keyboard-based (rather than mouse-based) mechanisms to handle and select objects.

4.5 Tools

4.5.1 Tool evolution policy

The tools chosen for the development of the platform are listed in Table 4.2

Table 4.2: Software development tools

Category	Name	Version
Configuration	Autoconf	Current version. 2.59 or later
Compilation	Automake	Current version. 1.9 or later
Library support	Libtool	Current version. 1.5.6 or later
C++ compiler	Gcc	3.3.5 or later
Python language support	Python	2.3.5 or later
C++/Python wrapper	SWIG	1.3.24 or later
Graphic library	Qt	3.3.3 or later
Statistics library	R	2.0.1 or later
Version control	Subversion	1.1 or later
	flex	2.5.33 or later
	python-imaging (PIL)	1.1.6 or later
	xerces-c	2.7
	boost	1.30 or later
	BLAS	3.0 or later
	LAPACK	3.0 or later

The versions given here are only meant as indications and other versions may be used. However, in case of compatibility issues arising from the use of other packages than those suggested here, support may not be the responsibility of the project.

[1] English has been chosen as the native language for the Open TURNS platform.

©2005-2012 EDF - EADS - PhiMeca

4.5.2 Programming conventions

The present document does not deal with the the programming conventions. These are described in a separate document cited in the bibliography under the reference [?].

4.5.3 Version control

The present document does not deal with the version control policy, which is described in a separated document cited in the bibliography under the reference [?]

©2005-2012 EDF - EADS - PhiMeca

©2005-2012 EDF - EADS - PhiMeca

Bibliography

[Ale] Andrei Alexandrescu. *Modern C++ design, Generic programming and design patterns applied.* Addison Wesley.

[Aus] Matthew H. Austern. *Generic programming and the STL, Using and extending the C++ Standard Template Library.* Addison Wesley.

[BOO] BOOST website. `http://www.boost.org/`.

[CSFP] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Micheal Pilato. *Version Control with Subversion, for Subversion 1.1.* Book compiled from revision 1337.

[Dal] Matthias Kalle Dalheimer. *Programming with Qt.* O'Reilly.

[Del] Claude Delannoy. *Programmer en langage C++.* Eyrolles.

[Dut] Anne Dutfoy. Partenariat EDF-EADS-Phimeca, Contenu méthodologique d'Open TURNS version standard. Note EDF R&D HT-52/05/021/A.

[ES] Margaret A. Ellis and Bjarne Stroustrup. *The annotated C++ reference manual, ANSI base document.* Addison Wesley.

[GHJV] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns.* Addison Wesley.

[LA] Mark Lutz and David Ascher. *Learning Python.* O'Reilly.

[LB] Bil Lewis and Daniel J. Berg. *Threads primer, A guide to multithreaded programming.* Prentice Hall.

[Lut] Mark Lutz. *Programming Python.* O'Reilly.

[Meya] Scott Meyers. *Effective C++.* Addison Wesley.

[Meyb] Scott Meyers. *Effective STL, 50 specific ways to improve your use of the Standard Template Library.* Addison Wesley.

[Meyc] Scott Meyers. *More effective C++, 35 new ways to improve your programs and designs.* Addison Wesley.

[Mul] Pierre-Alain Muller. *Modélisation objet avec UML.* Eyrolles.

[OT] Open TURNS project public website. `http://www.openturns.org/`.

[OTD] Open TURNS project public website. `https://81.80.78.196/Incertitude`.

[PY] Python website. `http://www.python.org/`.

[R] R project website. http://www.r-project.org/.

[SAL] SALOME project web site. http://www.salome-platform.org/.

[SVN] Subversion website. http://subversion.tigris.org/.

[SWI] SWIG website. http://www.swig.org/.

[UNI] Unicode website. http://www.unicode.org/.

©2005-2012 EDF - EADS - PhiMeca