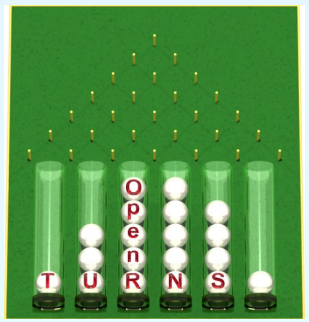
OpenTURNS Developer training
Elements of architecture

Trainer : Julien Schueller
Phimeca
schueller@phimeca.com

Developers training



Elements of architecture

Introduction

Objectives

The objectives of this course are:

- To present the main programming paradigms used in OpenTURNS;
- To present some of the most used design patterns;
- To illustrate these points based on a specific hierarchy: the distribution hierarchy.

A few words about C++

Why C++?

- The C++ language is by many aspects an awful language, so there is no point to start a programing language war with a C++ developer: he already knows all that you can say!
- Nobody can pretend to be a C++ expert, even its conceptors, even the compiler developers! As such, it is always possible to blame gcc for a bug of your own...
- Despite its numerous defects (not really an object language, too low level for the object community, too high level for the performance-inclined developers), it allows for the definition of a domain specific language that hide most of the ugly aspects from the numerician in favor to a much natural language where the concepts are named in the proper way, with the attended behaviour.
- It was one of the key points that lead to the choice of C++ as a programming language for OpenTURNS: the separation between the architect in charge of the low-level aspects of the programming language, and the numerician in charge of the algorithmic development.

Resource management

Resource Acquisition Is Initialization (RAII) paradigm

Definition [From wikipedia]: Resource Acquisition Is Initialization is a programming idiom used in several object-oriented languages like C++, D and Ada. The technique was invented by Bjarne Stroustrup[1] to deal with resource deallocation in C++. In this language, the only code that can be guaranteed to be executed after an exception is thrown are the destructors of objects residing on the stack. Resources therefore need to be tied to the lifespan of suitable objects in order to gain automatic reclamation. They are acquired during initialization, when there is no chance of them being used before they are available, and released with the destruction of the same objects, which is guaranteed to take place even in case of errors.
RAII is vital in writing exception-safe C++ code: to release resources before permitting exceptions to propagate (in order to avoid resource leaks) one can write appropriate destructors once rather than dispersing and duplicating cleanup logic between exception handling blocks that may or may not be executed.

Resource management

Resource Acquisition Is Initialization (RAII) paradigm

Typical use [From wikipedia]: The RAII technique is often used for controlling mutex locks in multi-threaded applications. In that use, the object releases the lock when destroyed. Without RAII in this scenario the potential for deadlock would be high and the logic to lock the mutex would be far from the logic to unlock it. With RAII, the code that locks the mutex essentially includes the logic that the lock will be released when the RAII object goes out of scope. Another typical example is interacting with files: We could have an object that represents a file that is open for writing, wherein the file is opened in the constructor and closed when the object goes out of scope. In both cases, RAII only ensures that the resource in question is released appropriately; care must still be taken to maintain exception safety. If the code modifying the data structure or file is not exception-safe, the mutex could be unlocked or the file closed with the data structure or file corrupted.
The ownership of dynamically allocated memory (such as memory allocated with new in C++ code) can be controlled with RAII, such that the memory is released when the RAII object is destroyed.

Resource management

Memory management and smart pointers

OpenTURNS uses extensively dynamic memory allocation. In order to tie to the RAII paradigm, all the memory management is delegated to smart pointers (the Pointer class). The benefits of this approach are:

- An easy to implement copy on write mechanism, that permit a significant reduction of the memory footprint by allowing for a large data sharing between objects;
- No C-like pointers in members of classes, which permits an automatic generation of the copy constructor, the assignment operator and the destructor of almost all the classes: there is no problem of deep copy versus reference copy;
- The resource is released automatically when the objects are outside of the current scope and there is no more reference on the allocated memory;
- There is a unique point where to prevent concurrent access in a parallel context, which is a key property for parallelism.

Design patterns

Definition

A design pattern is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

Design patterns

Singleton pattern

The singleton pattern is a design pattern used to implement the mathematical concept of a singleton, by restricting the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.
In C++ it also serves to isolate from the unpredictability of the order of dynamic initialization, returning control to the programmer.

Usage in OpenTURNS

This pattern is used in the following classes:

- Catalog, that records all the object's factories into a dictionary.
- IdFactory, that builds unique Ids for all PersistentObjects.
- RandomGenerator, that holds the (pseudo) random generator state.
- ResourceMap, that defines a catalog containing all default values of algorithms parameters.

Design patterns

Factory method pattern

The factory method pattern is an object-oriented design pattern to implement the concept of factories. It deals with the problem of creating objects (products) without specifying the exact class of object that will be created. The creation of an object often requires complex processes not appropriate to include within a composing object. The object's creation may lead to a significant duplication of code, may require information not accessible to the composing object, may not provide a sufficient level of abstraction, or may otherwise not be part of the composing object's concerns. The factory method design pattern handles these problems by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created.

Usage in OpenTURNS

This pattern is used in many places, amongst which one can find:

- PersistentObjectFactory, that allows for the creation of the persistent objects in OpenTURNS. It is part of the save/load mechanism of OpenTURNS.
- DistributionFactory, that corresponds to the statistical inference of probability distributions based on realizations.
- OrthogonalFunctionFactory, that corresponds to the notion of countable infinite sequence of orthogonal functions.

Design patterns

Strategy pattern

The strategy pattern is a particular software design pattern, where algorithms can be selected at runtime.
The strategy pattern is intended to provide a means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable. The strategy pattern lets the algorithms vary independently from clients that use them.

Usage in OpenTURNS

This pattern is used in many places in order to implement the notion of cardinal product of algorithms:

- HistoryStrategy, in charge of the management of the incremental building of collection of data.
- SamplingStrategy, in charge of the sampling of an hypersphere in the directional sampling algorithm.
- RootStrategy, in charge of the exploration of a specific direction in the directional sampling algorithm.
- ProjectionStrategy, in charge of the computation of the coefficients of an L^2 projection in the functional chaos algorithm.
- AdaptiveStrategy, in charge of the selection of partial bases in the functional chaos algorithm.

Design patterns

Composite pattern

The composite pattern is a partitioning design pattern. The composite pattern describes that a group of objects are to be treated in the same way as a single instance of an object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.

Usage in the library

This pattern is used in many places in order to permit recursive definition of concepts:

- ComposedFunction, that implements the functional composition of the function algebra.
- ComposedDistribution, for the creation and manipulation of distributions defined by a copula and a collection of marginal distributions.
- ComposedCopula, for the creation and manipulation of copulas defined by a collection of lower dimensional copulas.
- CompositeRandomVector, in charge of the creation and manipulation of random vectors built as the image by a numerical function of other random vectors.

Design patterns

Bridge pattern

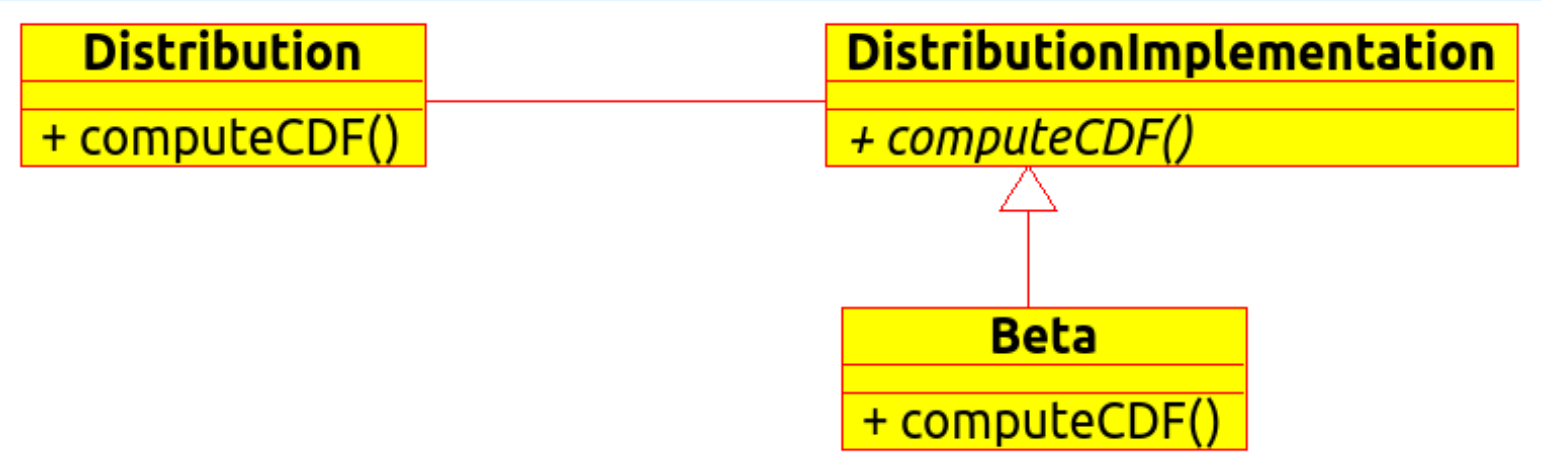
The bridge pattern is a design pattern used in software engineering which is meant to "uncouple an abstraction from its implementation so that the two can vary independently". The bridge uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes.
When a class varies often, the features of object-oriented programming become very useful because changes to a program's code can be made easily with minimal prior knowledge about the program. The bridge pattern is useful when both the class as well as what it does vary often. The class itself can be thought of as the implementation and what the class can do as the abstraction. The bridge pattern can also be thought of as two layers of abstraction.

Design patterns

Bridge pattern

This pattern is one of the most widely used. Some examples are:

- Drawable, that separate the generic high level interface of a drawable from the specific low level interface of the several drawable specializations.
- Distribution, that expose a high level interface of the concept of probability distribution whereas the DistributionImplementation class expose the low level interface of the same concept.



Design patterns

Other patterns

Many more patterns are involved in the platform. The notion of design pattern is a higher granularity of description of an object oriented software that is more adapted to describe the interaction of concepts than the class hierarchy view.
In this respect, it is possible to produce maintainable sofware by reproducing a class organization that has proved its adequation to concretize given mathematical concepts for other concepts that are mathematically organized the same way.
The identification of such patterns is one of the early phases of the detailed design of a new functionality.

A detailed hierarchy

The concept of probability distribution and its implementation

The concept of probability distribution has been presented in the course "probabilistic uncertainty propagation". It is the core concept of probability modelling and is characterized by:

- A large set of generic aspects (common notion of CDF, mean, covariance, quantile);
- A large set of specific aspects (different notions of PDF, existence of services only for specialized distributions such as discrete ones);
- A huge variety of algorithms, some with a high level of specificity, some other with a very generic usage;
- Several composition mechanisms (mixture, random mixture, composition).

As such, we will find a bridge pattern in order to separate the high level interface of the concept (the Distribution class) from its low level interface (the DistributionImplementation class). Several composite patterns will be used for the several composition models (ComposedCopula, ComposedDistribution, Mixture, RandomMixture, TruncatedDistribution, SklarCopula).

A detailed hierarchy

Navigation in the Doxygen documentation

The internal documentation can be extracted using Doxygen and the target `html` in the lib/ subdirectory: make html.