Programmazione Avanzata per il Calcolo Scientifico
Advanced Programming for Scientific Computing
Lecture title: A class for numerical quadrature

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2014/2015

An example of design of a library for numerical quadrature

Clonable classes

A class for composite numerical integration in 1D

We want to

- Create a class for composite numerical integration able to use different integration rules with the minimal duplication of code;
- Have a code open to extensions;
- Be able to load integrands and quadrature rules dynamically!

The open-closed principle

A good code is open to extensions (open) and at the same time extensions should change as little as possible the existing code (closed).

Remember that a good code has been coded at least thrice (H. Sutter) and that programming is in the future tense (Ritter). That is don't be surprised if you discover that the first design eventually is not the right one, and you have to change it. And when designing a code try to foresee all possible future usage and verify if the chosen design is suitable.

However, generality/extensibility and efficiency are sometimes in contrast, so code design is also a matter of compromise and depends on the objectives.

The chosen design

A composite quadrature needs a mesh, that is a partition of the interval (a, b) where we want to approximate the integral, and a rule to integrate in each sub-interval. The most general formula is then

$$\int_a^b f(x)dx \simeq \sum_{j=0}^{N-1} \text{rule}(x_j, x_{j+1}, f)$$

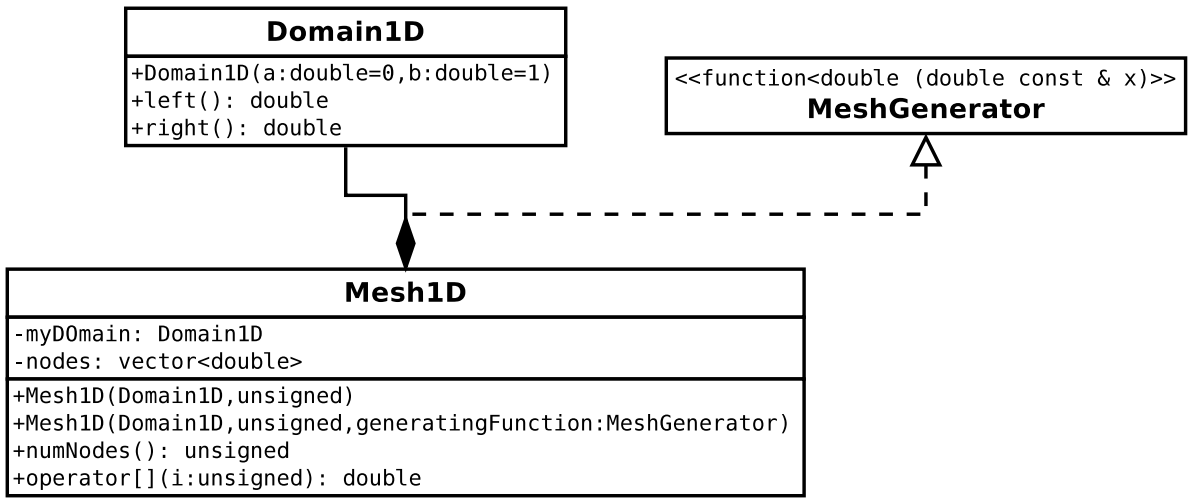
For standard numerical quadrature rule we have

$$\text{rule}(x_j, x_{j+1}, f) = \frac{x_{j+1} - x_j}{2} \sum_{k=0}^{m-1} \hat{w}_k f(\overline{x}_j + h\hat{y}_k)$$

where the weights $\hat{w}_k$ and the nodes $\hat{y}_k$ are usually given in the reference element $(-1, 1)$.

The mesh

In the directory OneDMesh you have an example of a rather simple class for 1D domain and mesh.



You need to compile and install (have a look at the `README` file!).

The MeshGenerator

The Mesh1D class takes two constructors. The first takes a Domain1D (a simple class with the two ends of an interval of $\mathbb{R}^2$) and the number of elements. It creates a uniformly distributed mesh. The second takes in addition a function<**double** (**double const** &)> that returns the desired spacing $h(x)$ for $x \in \Omega = [a, b]$.
The lib1DMesh library provides two functions that generates a uniform mesh and a variable size mesh. The latter generates nodes according to the given $h(x)$ and it uses the function rk45 of the librk45 library.

The generation of a non-uniform mesh

The spacing function h is used to construct a map $M : \Omega \to [0, n]$ such that

$$\int_a^b M(x)dx = n, \quad \int_a^b |h^{-1}(x) - M(x)| \leq 0.5.$$

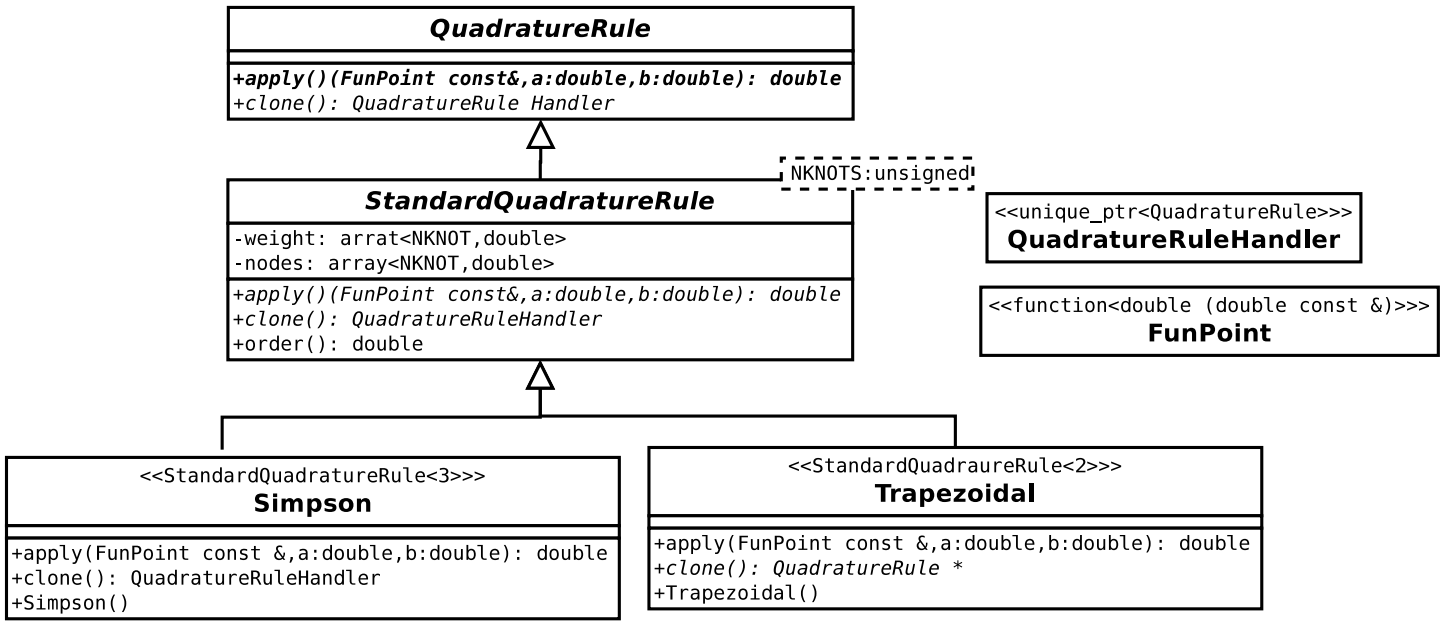
In practice this is approximated by solving numerically

$$\frac{dy}{dx} = h^{-1}(x), \quad x \in (a, b) \quad y(0) = 0,$$

re-scaling y so that $y(b) = n$, and finally finding the values of x such that $y(x) = i$ for $i = 1, n - 1$.

The general design for QuadratureRule

The class QuadratureRule in QuadratureRule.hpp defines the abstract interface of all rules. It is a very simple interface. While StandardQuadratureRule<N> that of a rule using N nodes and weight. Both are abstract classes.
The general layout is



The concrete StandardQuadratureRule

A concrete standard quadrature rule is obtained by inheriting from StandardQuadratureRule<N>, with N set to the appropriate value. The constructor of the concrete rule is responsible of filling the values of the weights and nodes. For instance:

```
Simpson::Simpson():
StandardQuadratureRule<3>({{1./3,4./3,1./3}},{{-1.0,0.0,1.0}},5)
  {}
```

Note that the actual filling is delegated to the base class constructor that takes nodes and weights (and the rule order).

The apply method of a StandardQuadratureRule

```
double
StandardQuadratureRule<N>::apply(
    FunPoint const &f,
             double const &a,
             double const &b)const{
  double h2((b-a)*0.5); // half length
  double xm((a+b)*0.5); // midpoint
  auto fscaled=[&](double x){return f(x*h2+xm);};
  double tmp(0);
  auto np=_n.begin();
  for (auto wp=_w.begin();wp<_w.end();++wp,++np)
          tmp+=fscaled(*np)*(*wp);
  return h2*tmp;}
```

The clone() method

The classes are clonable, in particular there is a virtual method called clone that return a unique_ptr<QuadratureRule>, aliased to QuadratureRuleHandler with a copy of the concrete object.

This is not strictly necessary and it is linked to a previous implementation that made extensive use of the so called prototype pattern.
....Anyway, it is the moment of explaining cloning.

Clonable classes

We need to find a nice way to construct the rule to be stored as composed object. There are different ways of doing it, we present here a nice technique, based on the template design pattern (nothing to do with C++ templates!), that simplifies the work.

The technique is also called virtual constructor or clonable class. We use a virtual method, called clone() whose role is to return a pointer (or a unique_ptr) to a clone of the object to which it is applied. If B is the base class of D its usual layout relies on the copy constructor (the class must be copy-constructable) and is:

```
B * D::clone() const{return new D(*this)} ;
```

The class D is said to be clonable.

What's the use of a clonable class?

It simplifies the composition with a polymorphic object:

```
using namespace std;
class Myclass{
public:
Myclass(BaseType const & b ):base(b.clone())
..
private:
unique_ptr<BaseType> base;
}
```

I may use a pointer instead

```
Myclass(BaseType const * b ):base(b->clone())
```

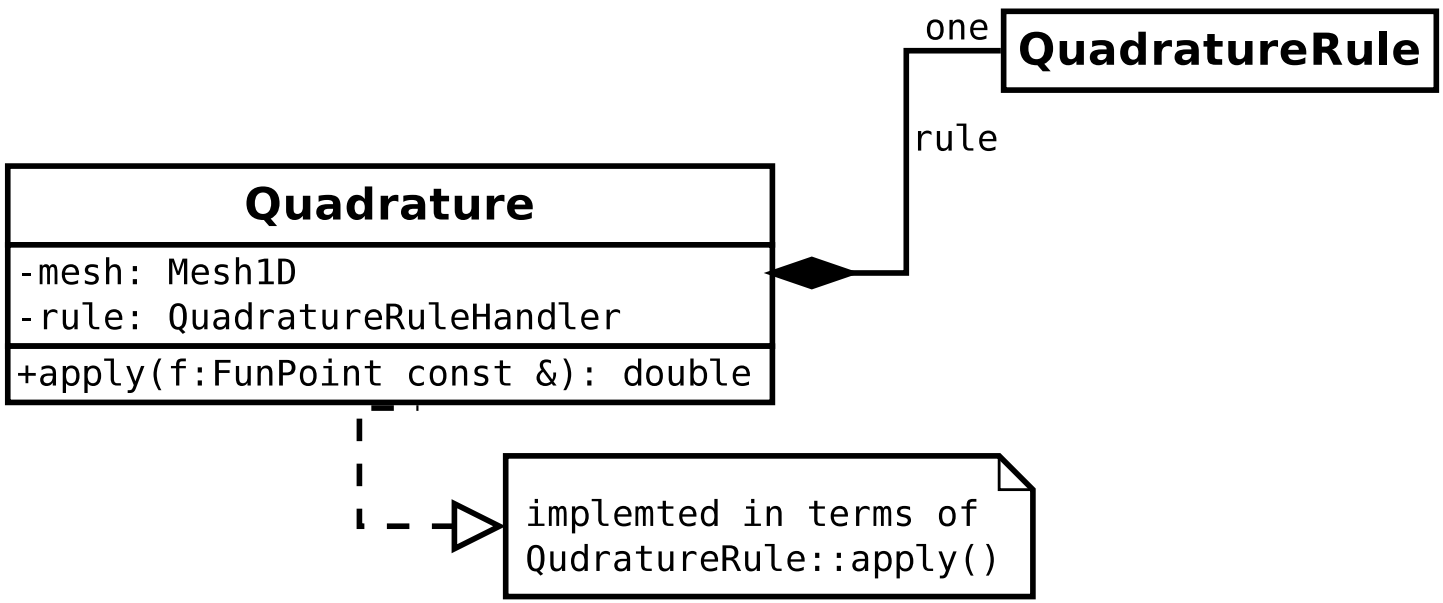
Then, to compose the object

```
Derived d;
Myclass pippo(d);\\reference version
Myclass pluto(new Derived());\\pointer version
```

Now MyClass owns an object of Derived type. Here d is completely independent from pippo, we have stored a clone!.

The design of the composite quadrature class

I have decided that a composite quadrature class is composed with the corresponding rule, that is it owns the corresponding object. Since the object is polymorphic I decided to use a unique_ptr to store the resource. Here the layout in numerical_integration.hpp.



The apply method of the Quadrature class

```
double Quadrature::apply(FunPoint const & f) const{
  double result(0);
  for(unsigned int i=0;i<=mesh.numNodes()-1;++i){
    double a=mesh[i];
    double b=mesh[i+1];
    result += rule->apply(f,a,b);
  }
  return result;}
```

Some critics

- The implementation of a new standard rule is simple. You just have to fill nodes and weights and indicate the order of the rule (this is required only for some later use).
- Quadrature rules are composed into the class. The Quadrature object is self-contained.
- The integrand is passed as arguments. An alternative would be to store it as a member. It would make the class even more self-contained with a loss of flexibility.
- The implementation of the composite quadrature is not the most efficient one. An efficient implementation would avoid to call the function twice on the same node. But it will be less flexible!.

Compilation

We are supporting both static and dynamic libraries.

- make dynamic DEBUG=no compiles the dynamic libraries
- make static DEBUG=no compiles the static libraries
- make install installs header files and libraries in Examples/include and Examples/lib
- make exec LIBTYPE=STATIC/DYNAMIC compiles executable against static/dynamic libs in the current directory.
- make exec DEBUG=no LIBTYPE=STATIC/DYNAMIC compiles executable against STATIC/DYNAMIC in the Example/lib directory.

What if the QuadratureRule classes where not clonable?

The virtual constructor technique is nice but maybe we decide not to have a clone() method in our QuadratureRule classes. In this case copying a Quadrature object would be difficult since we store only a pointer to the base class! How can we then create the object of the derived class to be stored in the copy??

The library `Loki` of Alexandrescu gives the implementation of a generic *clone factory*. It is rather complex: it uses run type identification and metaprogramming techniques. However its use is not so complicated, but goes beyond the scope of this lecture.

Alternative: we abandon composition and use simple aggregation. Beware then that the lifespan of the aggregated QuadratureRule must not be shorter than that of Quadrature.

Second alternative: make Quadrature a template class and pass the concrete QuadratureRule as template argument. This is a fine solution (and indeed a very good alternative to the proposed design!). It has been used in the next examples.

Extending the class

We have developed a class for numerical quadrature. We want now to extend it in several directions:

- Implement a Montecarlo quadrature.
- Be able to estimate the error of a standard quadrature rule, in order to be able to implement an adaptive quadrature method.

All that by modifying the little as possible what has been written so far.

Adding the computation of the error estimate

We want to enrich s StandardQuadratureRule class so that we can compute an estimate of the error using the formula

$$e_{h_k} = |I_{\frac{h_k}{2}} - I_{h_k}| \frac{2^{p-1}}{2^{p-1} - 1},$$

being p the order of the rule. We also want to be able to use the computed error externally to the class (to implement the adaptive strategy).

Adding the computation of the error estimate

We have a first problem. QuadratureRule is a private member of the composite quadrature class used in the apply() method. We do not want to change the apply() method. We want to reuse what we have done so far! We need an external object whose role is to accumulate the computed error.

Second problem. We need to modify the apply method of the StandardQuadratureRule so that it computes the error estimate. A possibility is to override the method in each derived class, creating SimpsonPlusError, TrapezoidalPlusError etc. But this technique is not scalable! We need an alternative that allows us to have a new apply() method for all the hierarchy of StandardQuadratureRule classes in one shot!!

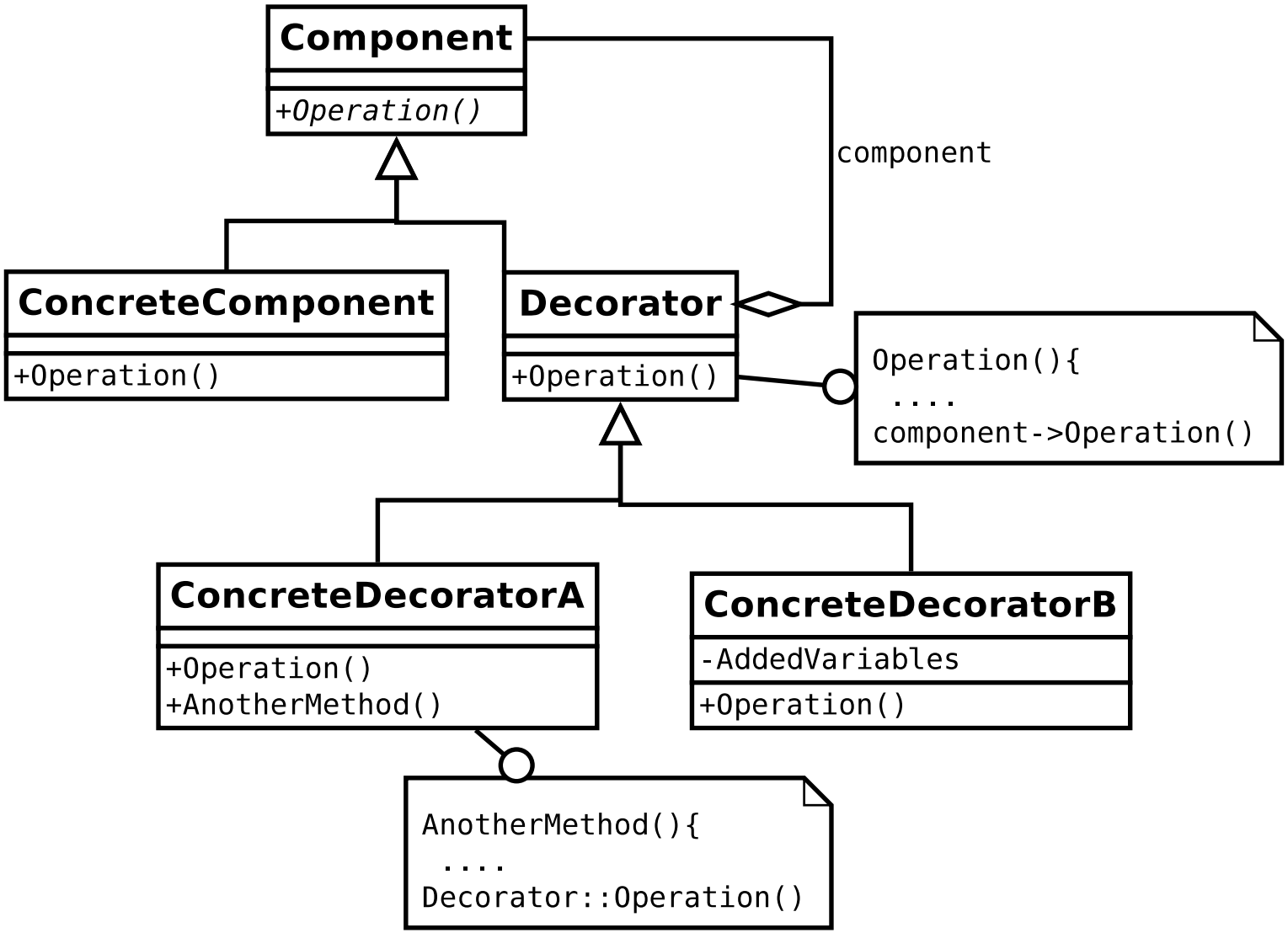
Decorators and extractors

We use two design patterns

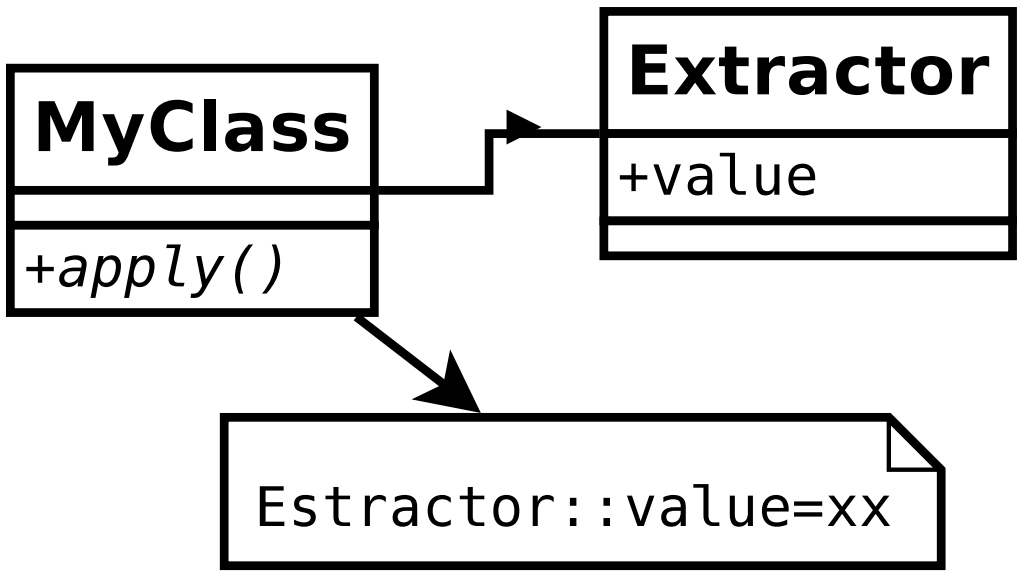
- Decorator. A decorator is a class that extends (decorates) the functionality of a polymorphic hierarchy of classes. This is achieved by designing a decorator class that wraps the original class by combining public inheritance and aggregation (or composition). The decorated class is implemented-in-term-of the aggregated object. Inheritance ensure its usability in all contests where we use a QuadratureRule.
- Extractor. The extractor technique delegates to an external object the role of collecting information elaborated in a class method, without the need of changing the interface of the class.

Decorator

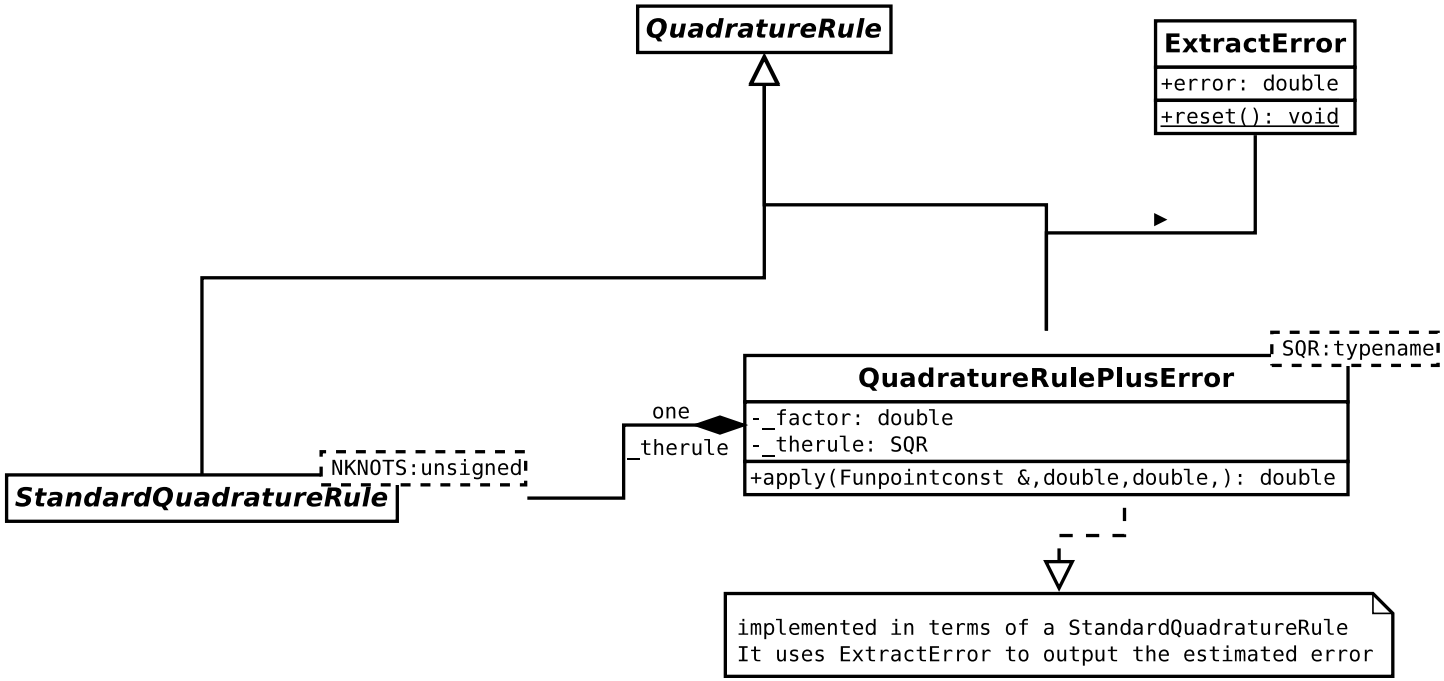
General design of a decorator



Extractor



QuadratureRulePlusError



Details in QuadratureRulePlusError.hpp.

```
 struct ExtractError{
  static double error;
  static void reset(){error=0;}
};
..
 template<class SQR>
 class QuadratureRulePlusError : public StandardQuadratureRule
 public:
  int num_nodes() const{_therule.num_nodes(i);}
  double node(const int i) const {_therul.node(i);}
  ...
  double apply(FunPoint const &,
               double const & double const &) const
 private:
  static SQR _therule;
```

The method apply of the decorated class

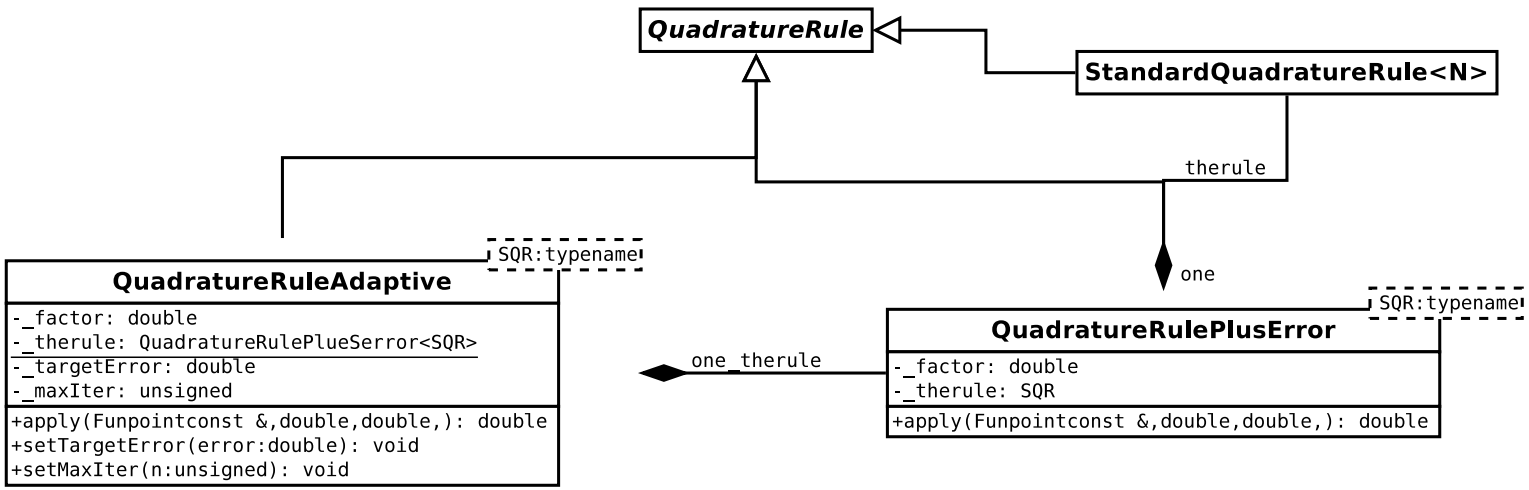
```
double QuadratureRulePlusError::apply(FunPoint const & f,
                                      double const & a,
                                      double const & b) const
{
  double result=_therule.apply(f,a,b);
  double xm=(a+b)/2.0;
  double result2=
   _therule.apply(f,a,xm)+_therule.apply(f,xm,b);
  // Record the error in the extractor.
  ExtractError::error+=(result2-result)*_factor;
  return result;// I should return result2
}
```

The main program

```
  ...
  Domain1D domain(a,b);
  Mesh1D mesh(domain,nint);
  Simpson simpsonRule;
  Quadrature s(QuadratureRulePlusError<Simpson>,mesh);
...
  auto approxs=s.apply(f);
  auto computerError=ExtractError::error;
....
```

The adaptive quadrature

In fact the quadrature rule with computation of the estimate of the error is only an intermediate point for the adaptive rule. It has been implemented again using a Decorator pattern.



The code

The complete code is in QuadratureRuleAdaptive.hpp

Montecarlo integration

We now want to implement a Montecarlo integration, which is not a standard quadrature rule!
On each interval h_k we compute

$$\int_{x_k}^{x_{k+1}} f dx \simeq h_k < f >_{n_k}$$

$< f >_{n_k}$ being the average value of $f(x)$ on n_k random samples uniformly distributed in the interval. The value of n_k is incremented until $h_k \sigma(f, n_k)/\sqrt{n_k} \leq \epsilon$, ϵ being the desired error. We need to fix a minimal number of samples, otherwise the estimate does not make sense, and a maximal number of samples to avoid "infinite cycles".

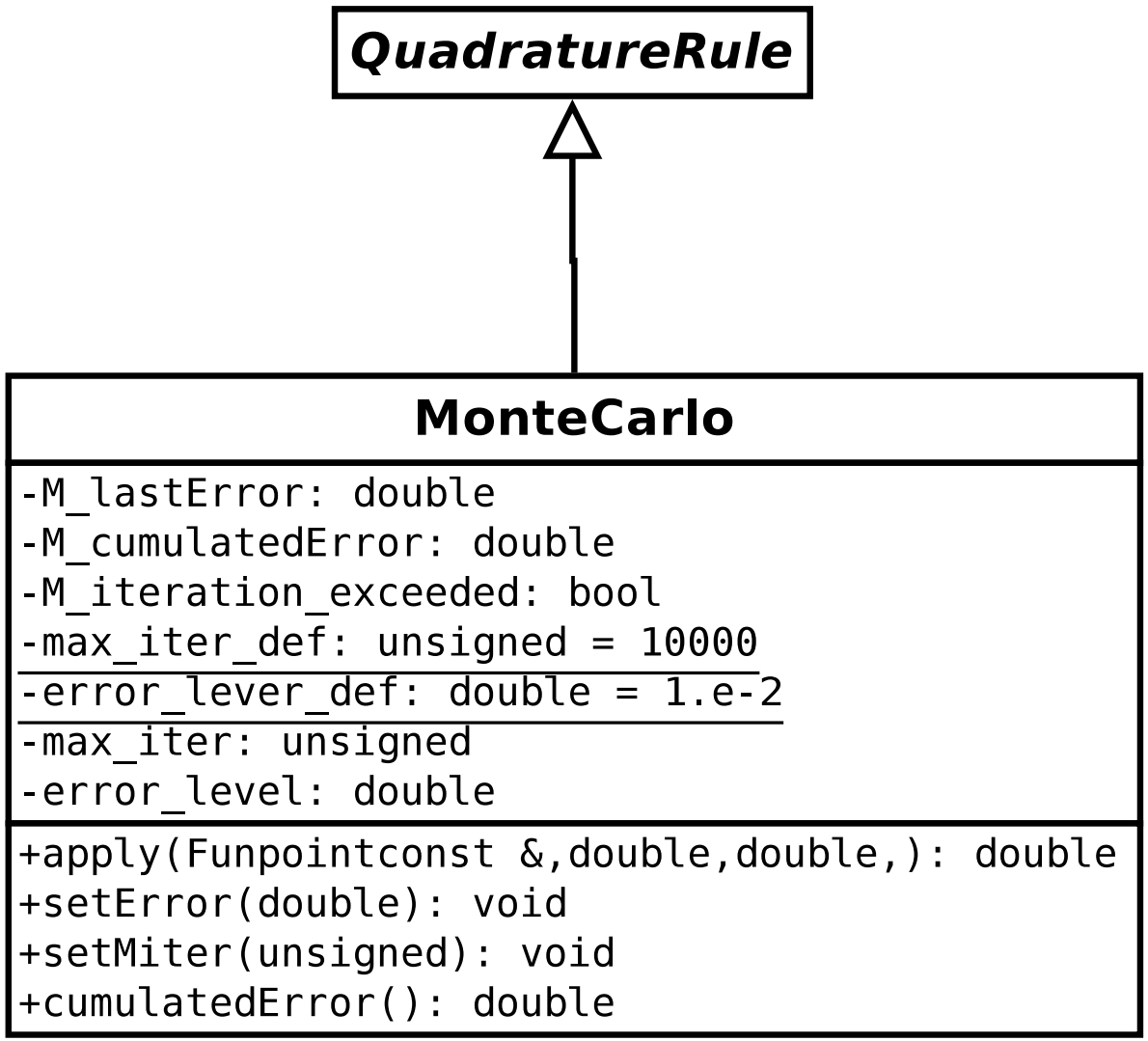
Running averages and variance

To compute running average and variance we use the formula originally proposed by D. Knuth.
Let $< f >_0 = 0$ and $S_0 = 0$.

$$i = 1, \ldots, n_k$$
$$x_i = \text{uniform}(x_k, x_{k+1})$$
$$\Delta_i = f(x_i) - < f >_{i-1}$$
$$< f >_i = < f >_{i-1} + \frac{\Delta_i}{i}$$
$$S_i = \Delta_i (f(x_i) - < f >_i)$$

Variance is the computed as $\sigma^2(f, n_k) = S_{n_k}/n_k$.

The layout



The full implementation is in montecarlo.hpp

The main

The main file main_integration.cpp, shows the usage of the different rules. It can be called using the script run_quadrule.sh.
`./run_quadrule.sh -h` prints a short help.
The file integrand.cpp contains the integrands that can be used and that are stored in the library `libintegrands.so`.