Acyclic Visitor Pattern in Formulation of Mathematical Model

Ales Cepek
Jan Pytel

Mathematical model

- general formulation of the functional relation between the unknown parameters and observed quantities is discussed in *Geodesy: The Concepts* by Vanicek and Krakiwsky
- three main components of mathematical model:
  - parameter space
  - observation space
  - model space
- relations between parameter, observation and model spaces.

Mathematical model - linear relations between parameter, observation and model spaces

Main problem

how to implement a set of classes describing linear relations between parameter, observation and model spaces in C++ language and define polymorphic functions like

```
observation->derivation(parameter);
```

where we need to select from MxN virtual functions.

Class Parameter

Class *Parameter* represents unknown parameter and contains these information:
- initial value
- correction (correction of parameter – from least squares adjustment)
- correspondent number of column in coefficient matrix
- type of parameter (unused, free, constrained, ...).

Class *Parameter* has list of other parameters on which is given parameter dependent.
Class Observation

- every type of observation has (his own) list of parameters
- linearization of observation is performed by set of partial derivatives of the functions with respect to the elements of list of parameters
- member function which represented linearization was changed

"Procedural" pattern

- two dimensional array of pointers to function of analytic derivations
- dimension 1 represents observations
- dimension 2 represents parameters
- two enumeration types:
  - enumeration type type_observation – named constants represents type of observations
  - enumeration type type_parameter - named constants represents type of parameters

"Procedural" pattern - example

```cpp
eenum type_observation { 
  obsDistance = 0,
  obsAngle,
  e_number_of_observations }; 

eenum type_parameter { 
  paramB = 0,
  paramL,
  paramH,
  e_number_of_parameters }; 

typedef double (*derivation)(Observation*);

derivation derivation_distance_L(Observation* obs);

derivation ListDerFun[e_number_of_observations][e_number_of_parameters] = { 0 }; 
ListDerFun[obsDistance][paramL] = derivation_distance_L; 
ListDerFun[obsAngle][paramH] = derivation_angle_H; 
```

"Visitor" pattern

- the base class of visitor hierarchy – abstract base class Parameter, which represents "visitor"
- class Parameter contains pure virtual member function
  virtual double Parameter::deriveXX(XX& ) = 0
- every new parameter have to be derived from the class Parameter and rewrote all member function deriveXX for all observations
- class Observation has defined only one pure virtual function
  virtual double Observation::derivation(Parameter v) = 0

"Visitor" pattern - example

```cpp
class Parameter {
  public:
  virtual double deriveDistance (Distance&) = 0;
  virtual double deriveAngle (Angle& ) = 0; 
};
class ParameterB : public Parameter {
  public:
  double deriveDistance (Distance& ) { return 10; } 
  double deriveAngle (Angle& ) { return 20; } 
};
class ParameterL : public Parameter {
  public:
  double deriveDistance (Distance& ) { return 30; } 
  double deriveAngle (Angle& ) { return 40; } 
};
class Observation {
  public: virtual double derive(Parameter v) = 0; 
};
class Distance: public Observation {
  public:
  double derive(Parameter v) { return v.deriveDistance(*this); } 
};
```
“Visitor” pattern disadvantages

- We have a cycle of dependencies that causes observation to transitively depend upon all its derivatives.
- When adding a new observation type (derived from class Observation) the Parameter class and all its derived subclasses need to be rewritten (add new virtual function).
- We have to define appropriate virtual function in all descendants classes of the class Parameter, clearly not all geodetic models need to define linearization for all observation types.

“Acyclic Visitor” pattern

- “Acyclic Visitor” pattern can avoid most of the major drawbacks of using the “Visitor” pattern or procedural pattern.
- Problems were solved by using multiple inheritance and dynamic_cast.
- Role of “visitor” has degenerate class Parameter in this pattern.
- We have to define abstract classes of derivations DeriveXX for every class (type of observation) derived from the class Observation; these classes have only pure virtual function virtual double derive(Angle*)=0;

“Acyclic Visitor” pattern - definition of new type of observation

When we need to add a new type of observations (by deriving from abstract class Observation) we have to define member function double derivation(Parameter* v):

```cpp
double Distance::derivation(Parameter* v)
{
    DeriveDistance* ad = dynamic_cast<DeriveDistance*>(v);
    if (ad)
        return ad->derive(this);
    else // error handling
        return 0;
}
```

“Acyclic Visitor” pattern - example

```cpp
class Parameter
{
public:
    virtual ~Parameter() {};
}
class Observation
{
public:
    virtual double derivation(Parameter* v) = 0;
};
class Angle;
class DeriveAngle
{
public:
    virtual double derivation(Angle*) = 0;
};
class Distance;
class DeriveDistance
{
public:
    virtual double derivation(Distance*) = 0;
};
class ParameterB : public Parameter,
                    public DeriveDistance,
                    public DeriveAngle
{
public:
    double derivation(Distance*);
    double derivation(Angle*);
};
```

“Acyclic Visitor” pattern advantages

- When we want define for example, analytic derivation of observation Angle by parameter ParameterH, we have to only the class ParameterH derive from class Observation and from class DeriveAngle and define virtual member function
double ParameterH::derive(Angle*)
- We can not define analytic derivations for all combination observations-parameters (contrast with “Visitor” pattern)
- The main advantage of the acyclic visitor pattern is that when defining a new observation type or a new model, the existing software is not affected (no dependency cycles).
The main advantage of the acyclic visitor pattern is that when defining a new observation type or a new model, the existing software is not affected (no dependency cycles). This design enables highly general level of abstraction in a software implementation of the mathematical model.

Thank you for your attention.