Acyclic Visitor Pattern in Formulation of Mathematical Model

Ales CEPEK and Jan PYTEL, Czech Republic

Key words: acyclic visitor pattern, mathematical model.

SUMMARY

This paper discusses the use and advantages of the acyclic visitor pattern in object oriented design of mathematical model. In geodetic methology the mathematical model plays a key role in formulation of the functional relations between the unknown parameters and observables.

The paper will endeavor to demonstrate the benefits of object oriented adjustment of observations in a linear mathematical model. With a design based on the visitor pattern, new functions and data structures can be added to existing class hierarchies without affecting them.

Acyclic Visitor Pattern in Formulation of Mathematical Model

Ales CEPEK and Jan PYTEL, Czech Republic

1. MATHEMATICAL MODEL

The spectrum of observation types dealt by geodesy is very wide and ranges from classical astro-geodetic observations (astronomical longitude and latitude, variations and position of the Earth pole), measurements of geophysical quantities (gravity acceleration and its local anomalies), through traditional geometric observables like directions, angles and distances to photogrammetrical measurements of historical monument, but of the main importance in geodesy today are satellite global positioning systems.

General formulation of the functional relation between the unknown parameters and observed quantities is thoroughly discussed in a classical book *Geodesy: The Concepts* by Vanicek and Krakiwsky [1]. For our needs of object oriented software design, implicit mathematical model can be expressed in terms of parameters $\boldsymbol{x}$ and observations $\boldsymbol{l}$, as

$$\boldsymbol{f(x, l) = 0}. \tag{1}$$

Corresponding to the main components of our model are three mathematical spaces: parameter, observation and model space. Three basic components of the mathematical model (1) are depicted in Fig. 1, where **A**, **B**, **G** and **H** are matrices of corresponding linearized relations.

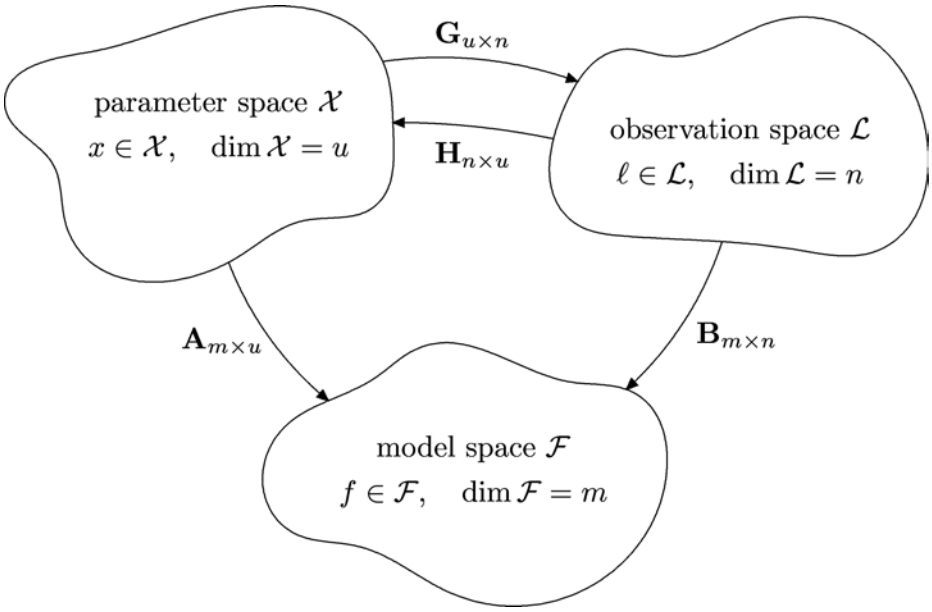


Fig. 1. Linear relations between parameter, observation and model spaces.

Models can be direct, indirect or implicit, linear or nonlinear, can occur individually or in combinations

model explicit in ***x***:	$\boldsymbol{x = g(l)}$,	$\mathbf{x = Gl + v}$
model explicit in ***l***:	$\boldsymbol{l = h(x)}$,	$\mathbf{l = Hx + v}$
implicit model:	$\boldsymbol{f(x, l) = 0}$,	$\mathbf{Ax + Bl + v = 0}$

In geodetic methodology the mathematical model plays a key role in formulation of the functional relations between the unknown parameters and observables [1]. Clear definition of the model components and their relations are equally essential for object oriented software design, where the mathematical spaces' elements are described in terms of classes.

2. C++ DESIGN OF MATHEMATICAL MODEL

In the following section, we will endeavor to demonstrate the benefits of object oriented design of C++ classes for implementation of mathematical model. As this text is written for geodesists and not for programmers, first of all we have to explain informally what are the classes. Classes enable us to describe some real world entities in the terms of a programming language, C++ in our case, and classes are user defined abstract data types (definitions of new object types). In the definition of a class we can hide implementation details and define a clear interface for manipulating its objects, ie variables of the given object type.

If we return to the general definition of mathematical model (1), then in our software design we want to define it in the terms of abstract classes modeling parameters, observables and functional relations. If our design is adequately general, we will be able to derive specialized classes from their parent classes (inheritance) for any geodetic application. And if our classes are polymorphic (i.e. classes derived from abstract parent classes), we can manipulate corresponding objects through references, or pointers, to their parent classes (polymorphism) and we can add new data structures that can be used by already existing functions without affecting the existing software.

All three components of model (1) are mutually dependent but this dependency cycle in the source codes can be eliminated using forward declarations. The main goal of our design of model (1) is to allow new functions and classes to be added into the existing class hierarchies without affecting the existing software and without introducing dependency cycles.

Let us demonstrate the discussed design problem on a simple case of linearization of observations. We need to go through all available observation objects and call proper linearization functions for them. For a given observation type, the linearization function needs information on the model object in use (linearization depends on the mathematical model). In C++ we would like to write something like this

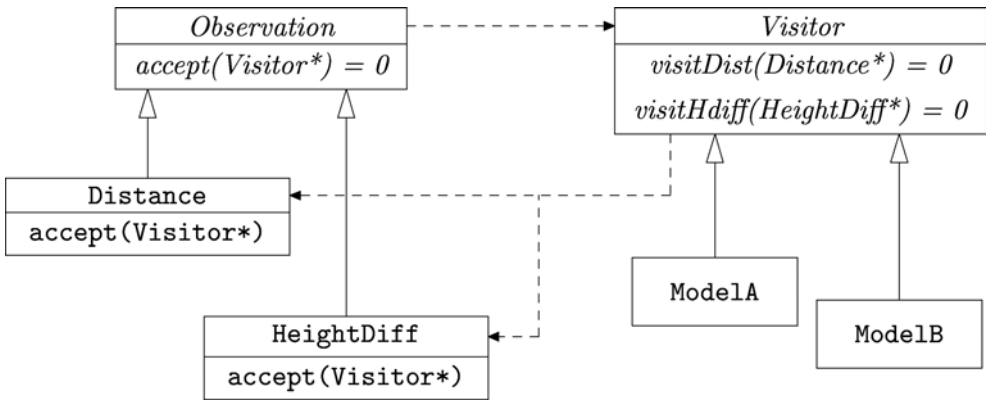
```
obs -> linearization(model);
```

where `obs` is a pointer to the base observation type and `model` is a base pointer to the current model object.

In the case of a single model the situation is trivial, the needed virtual function is simply called based on polymorphism (and in fact, model data even need not to be passed as the parameter; observation objects can keep a pointer to it). But for *N* observation types and *M* model types we have to select from *M*×*N* possible functions. For small values of *M* and *N* it might falsely seem to be acceptable to base the solution on a cascading of *if* statements or on some kind of enumerations added into observation and model classes, but this way could never lead to an acceptable general design. The general object oriented solution to this problem is known as the *Visitor pattern* [2].

2.1 Visitor Pattern

Linearization of observations in our discussion serves only as an example on which we try to demonstrate the technical problem. In the following text we split the *linearization* function into two complementary operations, conventionally named *accept* and *visit*, and introduce a new class *Visitor*. Resulting class hierarchies and dependencies are shown in the following figure (arrows represent inheritance, dashed arrows dependencies, and names of abstract classes and pure virtual functions are written in italics)



where implementations of *accept* functions contain only a single call to the corresponding virtual *visit* function, for example

```
void Distance::accept(Visitor* v)
{
   v -> visitDist(this);
}
```

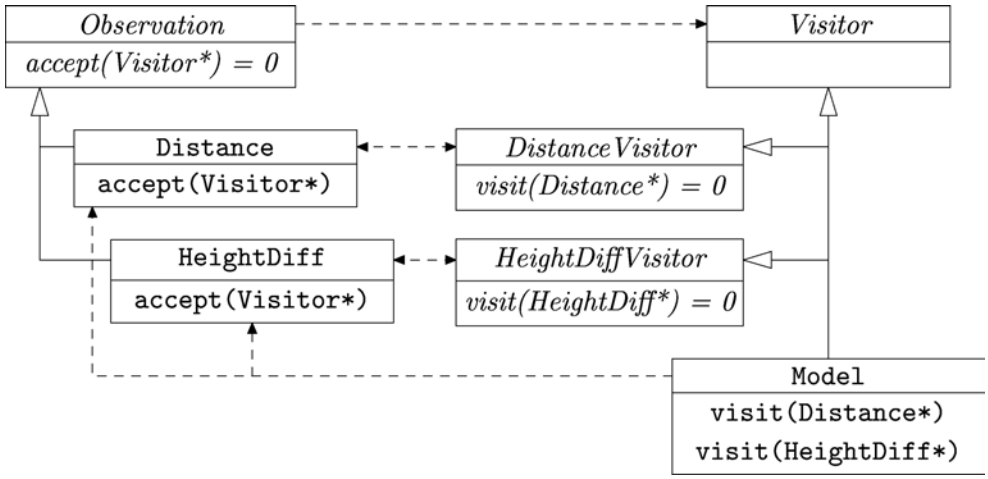
All model classes are derived from the abstract `Visitor` class and they implement their own virtual *visit* functions. In C++ dependency cycles in the source code (dashed arrows) can be eliminated using forward declarations. Abstract `Observation` class is the parent of all implemented observation types. In the Visitor pattern the selection of needed function is implemented as the call of two virtual functions.
We used different names for various visit functions in the abstract class `Visitor`. These functions can be uniquely distinguished by their parameters so the usage of distinct names is

just a stylistic convention and we will use only one overloaded name *visit* in the following text.

The main problem with the visitor patter is that when adding a new observation type the `Visitor` class and all its derived subclasses need to be rewritten. The more, clearly not all geodetic models need to define linearization (or any other similar operation) for all observation types. This disadvantage is eliminated in a variant pattern, known as Acyclic visitor [3].

2.2 Acyclic Visitor

The acyclic visitor pattern uses C++ dynamic cast mechanism to eliminate the need of definition of the *visit* function for a new observation type in all existing models. The basic idea is to remove pure virtual visit functions from the `Visitor` class and put them into individual observation-visitor abstract classes, one for each derived observation type. The `Visitor` class now has no functions (apart from the virtual destructor) and a model class is derived from it and from all observation-visitor classes that make sense for the given model. The model class must then implement all the relevant virtual visit functions, as depicted in the following figure



The role of observation-visitor classes (like *DistanceVisitor* above) is just to introduce a virtual visit function for the given observation type and thus they are ideal candidates for a single template class definition

```
template <class Observation> class Visitor {
public:
   virtual void visit(Observation*) = 0;
};
```

With templated observation-visitor class all *accept* functions are written following a simple scheme, here demonstrated on the example of a distance class

```
void Distance::accept(Visitor* v)
```

```
{
   if (Visitor<Distance>*
       dv = dynamic_cast<Visitor<Distance>*>(v))
      {
         dv -> visit(this);
      }
   else
      // ... error handling
}
```

If the visitor pattern is like a matrix, then acyclic visitor is like a sparse matrix [3], it implements only the relevant operations (in our case, linearization of observation types that are meaningful to the given mathematical model). Acyclic visitor adds only one runtime pointer conversion (dynamic cast) to the two calls to virtual functions from the visitor pattern. Acyclic visitor pattern as described here is based on multiple inheritance, this impose no problem with C++, but is not available in all object oriented languages. In some application areas dynamic cast might be too expensive in the terms of run time efficiency but we believe that this is not the case of geodetic applications.

3. CONCLUSIONS

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. This design enables highly general level of abstraction in a software implementation of the mathematical model, as formulated in [1].

Based on designed patterns published in [2] and [3] the acyclic visitor pattern has been implemented in a new development branch of our free software project for adjustment of geodetic networks http://www.gnu.org/software/gama.

REFERENCES

Petr Vanicek and Edward J. Krakiwski: *Geodesy: the Concepts*, North-Holland, 1986, 2nd ed., ISBN 0444 87775 4

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides: *Design Patterns — Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995, ISBN 0201 63361 2

Robert C. Martin: *Agile Software Development — Principles, Patterns, and Practices,* Prentice Hall, 2002, ISBN 0 1359 7444 5

BIOGRAPHICAL NOTES

Ales Cepek is professor of geodesy from 2003, working at the Department of Mapping and Cartography, Faculty of Civil Engineering, Czech Technical University, Prague, Czech Republic (since 1992). Started his professional career at the Research Institute of Geodesy, Topography and Cartography (VUGTK), Zdiby (from 1980 to 1991) where worked on research projects, programming, analysis and implementation of data structures for cadastral programs, co-author of programs for adjustment and analysis of geodetic networks; later at geodetic observatory observations with circumzenithal (astrolab), project for estimation of parameters of local quasigeoid. At present conducting research in the field of applications of XML and object-oriented processing of geodetic and cartographic data.

Jan Pytel is a PhD student of Ales Cepek. His research is focused on geodetic observations and deformation analysis. Jan Pytel published several papers and intensively works in the field of Free Software development (he is the author of free geodetic adjustment project Rocinante).

CONTACTS

Ales Cepek and Jan Pytel
Dept. of Mapping and Cartography
Fac. of Civil Eng., Czech TU Prague
Thakurova 7
166 29 Prague
Czech Republic
Tel. + 420 2 2435 4647
Fax + 420 2 2435 5419
Email: cepek@fsv.cvut.cz pytel@gama.fsv.cvut.cz
Web site: http://gama.fsv.cvut.cz/~cepek http://gama.fsv.cvut.cz/~pytel