THE TRANSFORMATION OF COORDINATES BASED ON DESIGN PATTERNS

Transformação de coordenadas com base em padrão de projeto

NORBERT RÖSCH

Geodetic Institute
Universität Karlsruhe (TH)
roesch@gik.uka.de

ABSTRACT

In this paper a new approach for the transformation of coordinates is suggested. This approach is based on design patterns which simplify the implementation significantly. The whole transformation process is depicted as a directed acyclic orientate graph. There each vertex denotes a class and each edge stands for one or more algorithms. Thus the transformation is nothing but a crossing of vertices and edges of this graph. Besides the rather simple maintenance of the program the complete flow of data concerning the transformation gets quite simple. Furthermore also the extensibility of the program can easily be performed. All aspects are explained by examples and code fragments.
Keywords: Coordinates transformation; Desing patters; A new approach for tansformation coordinates.

RESUMO

Neste artigo uma nova abordagem para a transformação de coordenadas é sugerida. Essa abordagem baseia-se no padrão de projeto que possibilita a simplificação da implementação de forma significativa. A totalidade do processo de transformação é representada como um gráfico orientado acíclico. Cada vértice denota uma classe e cada borda apóia a um ou mais algoritmos. Portanto a transformação consiste em cruzamento de vértices e bordas deste gráfico. Além disto, a manutenção do programa para o completo fluxo dos dados acerca da transformação é realizada de forma muito simples. Ademais o programa pode ser facilmente extendido. Todos os aspectos serão explanados com exemplos e fragmentos de códigos.
Palavras-chave: Transformação de coordenadas; Padrão de projeto; Uma nova abordagem para a trasfomação de coordenadas.

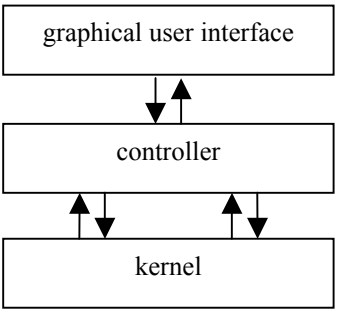
1. INTRODUCTION

The adoption of the object orientated concept led to a paradigm shift in software engineering. The term paradigm shift was first used by Thomas Kuhn in his book “The Structure of Scientific Revolution”(KUHN, 1996). According to the theory outlined in this book a paradigm denotes the sum of all rules used by a specific scientific community. If these rules don’t fit the underlying problems any longer they have to be replaced by new ones.

In the mid sixties the so called software crisis showed all criteria preceding a paradigm shift. As a consequence the concept of software engineering and further on the object orientated programming were introduced. Now these schemes are part of the new paradigm based on new strategies for the implementation of software are developed.

In the mid nineties Erich Gamma coined the term “design patterns” according to an analogy in architecture (GAMMA, 2000). Together with R. Helm, R. Johnson and J. Vlissides he had analysed many software products and found three main patterns which can be subdivided further. These three patterns are: creational patterns, structural patterns and behavioral patterns.

Figure 1. The design



Creational patterns are supposed to produce new instances when they are needed. The factory pattern, the singleton and the builder pattern are examples for members of this category. Behavioral patterns can be classified in observer, visitor, strategy and so on. Structural patterns comprise for example bridge, adapter and composite.

This paper describes the use of design patterns in the field of coordinate transformation which concerns especially geodesy (including GPS) and GIS respectively (see e.g. HECK, 2003). Actually for the transformation of coordinates not all the above mentioned patterns are needed. However the use of some of the most interesting patterns, the factory pattern, the singleton as well as the strategy is illustrated.

2. GENERAL OVERVIEW

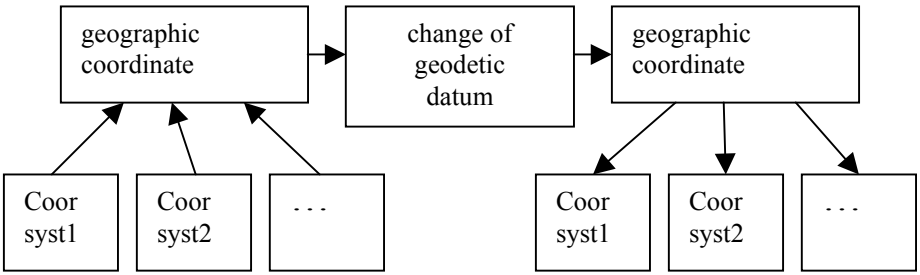
In this section some general aspects concerning the flow of data are discussed. The overall structure of the program is shown in figure 1. On the top level the user interacts with a graphical user interface (GUI). He takes his decision concerning the transformation which simply means he fills in the GUI and then starts the program.

The GUI hands over the parameters to the controller unit (abbr. controller) which is in fact an instance. The controller then forces the computation kernel (abbr. kernel) to perform the required transformation.

This last step is quite difficult, because depending on the users demand, the transformation can be rather complex. Thus the two layers (controller and kernel) have to exchange data for several times. Nonetheless the kernel only performs single point transformations which are treated to be atomic. As a consequence a datum transformation for example has to be subdivided into different steps, whereas the first one is always the transformation of any arbitrary coordinate into a geographic coordinate (see fig. 2). To keep the flow of data symmetric, the next step involves a change of the geodetic datum in any case. If no change of datum is needed, e. g. the datum of the source coordinate and the target coordinate should be the same, a dummy datum transformation is performed.

Due to this procedural method, the flow of data remains strictly symmetric e.g. there are no exceptions needed. Hence this is only possible, if there exists a method to detect the correct transformation method. This is the task of the strategy pattern which is outlined in detail in section 4.

Figure 2. The transformation designed as an directed acyclic graph



The whole transformation process can be depicted as an acyclic graph where the boxes or vertices respectively are symbolising the different stages of the transformation and the edges are standing for the appropriate algorithms (see fig. 2). E. g. to get geographic coordinates starting from UTM-coordinates the algorithm or edge respectively connecting the vertex UTM and *geographic coordinate* has to be passed. Actually the result needn't to be the final target of the transformation process. It is the task of the controller to find the path to the destination. Hence this is rather easy, because the structure of the graph is simple and most of the steps are fixed.

3. THE FACTORY METHOD

This design pattern is supposed to create different instances out of a group of classes during the life cycle of a program. According to the needs not only the instance is created but also the behaviour of the complete program is influenced. In the field of transformation, the creation of a vertex always implies different algorithms. E. g. the creation of an instance of the type "UTM" activates a method which is able to transform such a coordinate to another one of the type *geographic coordinate* (see also listing 1).

In this case also it is the task of the controller to create the vertices according to the user's choice. As a consequence the appropriate algorithms are provided by the specific instance. In practice these behaviour is implemented by the design pattern *factory method*. The *controller* forces the factory to create the vertices needed for the transformation. Hence, the factory only acts on demand, i. e. at the beginning of the program there exists no vertex nor an edge.

Depending on the user's input, the controller creates the appropriate elements of the acyclic graph step by step and passes the specific vertices and edges until the destination is achieved. In most cases this is any kind of a local plane coordinate system (e.g. UTM) in a specific geodetic datum (e.g. WGS84).

To explain the above mentioned we use a part of the implementation as an example. Listing 1 depicts the hierarchy of the coordinate classes. It is important to be aware that the class "GeneralCoordinates" encompasses the two methods "coord2geo" and "geo2coord". The first method stands for all edges connecting the classes on the bottom level of the left branch of the graph in figure 1, whereas the latter represents the edges to the bottom level classes of the left side. Note that all superfluous code is omitted in the listings.

Listing 1:

```
abstract class Coordinates {
  String pointid;
} // end Coordinates

abstract class GeneralCoordinates extends Coordinates {
  // attributes and methods
  abstract GeographicCoordinate coord2geo(…);
  abstract void geo2coord(…);
  ...
} // end GeneralCoordinates

final class GeographicCoordinate extends Coordinates {
  // attributs und methods
…
GeographicCoordinate coord2geo(…)
```

```
{// dummy}
void geo2coord(…)
{// dummy }
  ...
} // end GeographicCoordinate
```

As the class “GeneralCoordinates” only comprises the two abstract member functions “coord2geo” and “geo2coord” the specific implementation has to be done on the bottom level of the hierarchy of the coordinate classes. Each coordinate class (e. g. UTM) encompasses at least the above mentioned two member functions which are supposed to implement the specific algorithm as already mentioned in section 2. Listing 2 shows an example for such a bottom level coordinate class. Note that in this listing the class “Gauss” is omitted. This class comprises all the behaviour which is common for all gaussian coordinates.

Listing 2:

```
final class UTM extends Gauss {
  // attributes and methods
  ...
  GeographicCoordinate coord2geo(...)
{ // algorithm }
  void geo2coord(…)
  {// algorithm }
  ...
} // end UTM
```

The factory itself is actually also a class (see listing 3). This class activated by the *controller* “produces” the classes or mathematically expressed the vertices which are necessary to find a path to the target of the transformation. The factory class is depicted in listing 3. It consists only of one member function and has no attributes.

The process of the production of a new instance shall be explained within the next few lines. The argument “System” in line 2 of listing 3 denotes the coordinate system. In one of the next lines the new instance which represents the appropriate coordinate system is created. Together with the data the according member functions are activated too. Thus, the acyclic graph gets his first edge connecting the bottom level class with the vertex “geographic coordinate” (see fig. 1, left side of the graph).

Listing 3:

```
class CoordFactory {
  public static GeneralCoordinates getCoord(String System, …) {
  // the variable System is analysed
  // create a vertex of type COORSYS1 if true
    if (System.equals(COORSYST1))
      return new COORSYST1(control);
  // create a vertex of type UTM
    else if (System.equals(UTM))
      return new UTM(…);
  // there may be more coordinate systems
    else if (System.equals(…))
      return new …(…);
  } // end getCoord
} // end CoordFactory
```

Listing 4 shows, how the factory is implemented. Let us now assume that we want to transform a coordinate of type "UTM" from one zone to another. In this case no change of datum is needed. Thus the transformation gets rather easy.

First of all we will have a look to line 3, where an instance of the type "fromcoord" is created. In terms of graph theory this means that the first vertex of the acyclic graph is built.

Further on we expect a user input in the following line. In this line the user takes his choice concerning the kind of transformation. In our case he simply wants to change the zone number of the given coordinate (see above). Line 6 shows how this coordinate is created by the factory. Actually the information about the kind and extent of the transformation is stored in the *controller* which is denoted by "control" in this line. Actually also the controller is a class. As a result of the invocation of the factory another vertex is creates. This vertex which represents the coordinate in the source system is denoted by "fromcoord" in line 6.

Listing 4:

```
class Transformation {
  public static void main(String[] args) {
    …
  // create an instance of coordinates of the source system
  GeneralCoordinates fromcoord = CoordFactory.getCoord(object,control);
  // line 5
  // compute the according geographic coordinate (i.e. transform it)
  GeographicCoordinate geographic = fromcoord.coord2geo();
  // line 7
```

```
  // create an instance of coordinates of the target system
  GeneralCoordinates tocoord = CoordFactory.getCoord(object,control);
  // line 9
  // transform the coordinate back again
  tocoord.geo2coord(control,geographic);
  // line 11
  } // end main
} // end Transformation
```

The transformation of this coordinate to a geographic coordinate takes place in line 7. The next task in the flow of data is performed in line 9. Here the vertex on the right side of beneath the geographic coordinate, which means the coordinate of the target system (see fig. 2), is created. Although no transformation of the geodetic datum is needed, we can assume a dummy transformation to fit the example in figure 2.

In our case we need a plane coordinate of the type *UTM*. The according transformation with respect to the specific zone is computed by the method *geo2coord* in line 11. This is the last step at least as far as the transformation problem is concerned. The vertex *tocoord* represents the coordinate of the target system.

Actually this was a rather simple transformation problem. In the next section it will get more complex and a transformation which includes a change of the geodetic datum will be conducted. Therefore we introduce another design pattern, which is called *strategy*.

4. THE DESIGN PATTERN STRATEGY

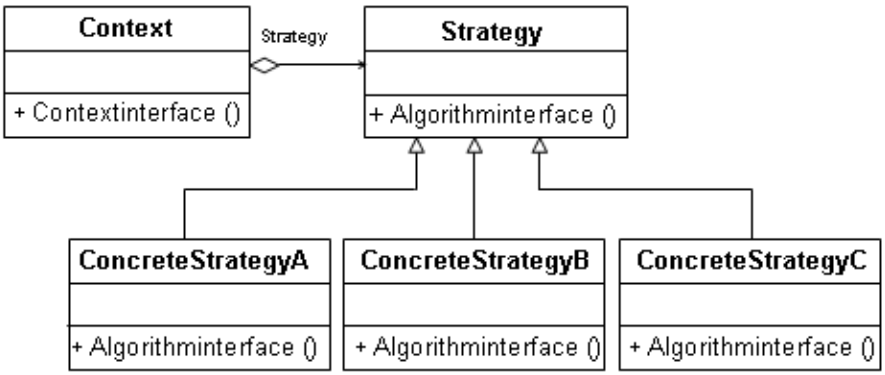
Figure 3 gives a general overview about the principle of the design pattern discussed in this section. The main thing is the *Context*-box, depicted upper left. The strategy only decides on the algorithm which is needed to perform the transformation.

Before we can get more detailed about the algorithms, we discuss the different cases for the geodetic datum transformation.

In practise different kinds of geodetic datum transformations are in use. In Europe most commonly the spatial similarity transformation which sometimes is also called 7-parameter- transformation, is applied. With respect to the according parameters, two different rotation matrices are possible. First we can think of a finite rotation, which leads to a combination of the sine and cosine of the rotation angles. The second solution assumes a infinite rotation. Thus the rotation matrix encompasses only the real angles. The discussion of the advantages and disadvantages of the different approaches is beyond the scope of this paper. So we

simply notice that there are two different rotation matrixes. Nonetheless both possibilities should be implemented in our transformation software.

Figure 3. The strategy pattern



Still also the Molodenskii formulas fit the transformation problem. Also these formulas are known in two variations. There exist a so called Molodenskii standard and an abridged formula. The latter is more compact than the other one. In addition to this for academic purposes there is still another transformation in use, known as 12-parameter transformation.

Further on the implementation of the above mentioned approaches is explained. Actually it is the user who decides about the kind of transformation. As a matter of fact the *controller* has to check this decision and single out the appropriate transformation. In terms of graph theory the controller has to find out the suitable path. Thus all possible transitions have to be implemented as an edge. Therefore we need the design patter *strategy*.

We should be aware, that each transformation is nothing but an algorithm which transforms a specific coordinate in another one of a different geodetic datum. Actually we have to consider different interfaces. In the case of a Molodenskii transformation we need a geographic coordinate whereas is the case of a 7- or 12-parameter transformation we need a 3 dimensional Cartesian coordinate. Hence the *strategy* should be able to handle all cases.

Figure 4 shows, also referring to graph theory, how this interface is implemented. It is nothing but a zoom in of the above depicted box denoted by *change of geodetic datum* in figure 2. It is important to be aware that the graph remains acyclic. Although there are multiple paths which could be passed for a specific coordinate, the *controller* singles out only one. Thus the task of the strategy is to create first the appropriate interface and afterwards activate the specific transformation.

Listing 5:

```
public interface Datum {
  public void datum(ControlParms control, GeographicCoordinate geo);
} // end interface Datum
```

In listing 5 the java interface added to the class definition of the different transformation algorithms is shown. These java interface is different from the interface mentioned in the clause above and shouldn't be confounded with the latter. Because the interface of the strategy is a class to which an algorithm is applied whereas the java interface is used to hand over a method. The term is the same but the sense is quite different.

Further on we have to expand the class "GeographicCoordinates" by the method *apply_strategy* depicted in listing 6. This method decides on the kind of transformation. In fact it is the *controller* which takes this decision. In listing 6 the specific variable is called *kindoftrafo*. The example in listing 6 encompasses only three out of the six above mentioned datum transformations. The terms describing the transformations are expected to be self- explanatory. The first one called NO_TRAFO is the dummy transformation which is needed for cases explained in section 3 (e. g. the change of a zone number).

Listing 6:

```
public void apply_strategy (ControlParms control) {
  Datum dat;
  if (control.kindoftrafo.equals(NO_TRAFO)) // dummy transformation
    dat = new NoTransformation();
  else if (control.kindoftrafo.equals(MOL_ABRIDGED))
    dat = new Molodenskii_trafo_standard();
  else if (control.kindoftrafo.equals(3D_INFIN))
    dat = new Spatial_similarity_infin();
  else
     … // unexpected error
  dat.datum(control,this);
} // end apply_strategy
```

Listing 7:

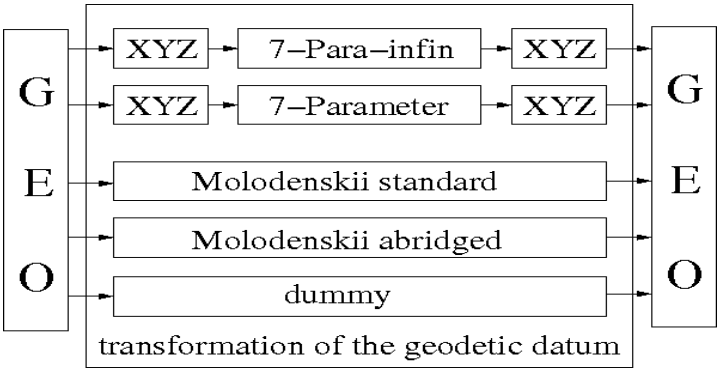
```
class Molodenskii_trafo_abridged implements Datum {
  public void datum (ControlParms control, GeographicCoordinate geo) {
    // algorithm }
} // end Molodenskii_trafo_abridged
```

```
class 3D_similarity implements Datum {
  public void datum (ControlParms control, GeographicCoordinate geo) {
    // algorithm }
} // end 3D_similarity

class No_Transformation implements Datum {
  public void datum(control, GeographicCoordinate geo) {
    //dummy}
} // end No_Transformation
```

Listing 7 shows the formal implementation of the three possible datum transformations listed in listing 6. The implementation of the class named "No_Transformation" seems to be rather curious, because the body of the method *datum* has no code. However these class is necessary to keep the overall design of the transformation symmetric. Due to this approach, exceptions are neither allowed nor are they needed.

Figure 4. The geodetic datum transformation



The adjoining figure shows the implementation details concerning the five transformations mentioned before. Also these terms are expected to be self-explanatory. Note the interface which is denoted by GEO in actually the geographic coordinate. Thus on the left as well as on the right the same interface is used. However each box stands for an algorithm. E. g. the first box named with 7-Para-infin references to the 7-parameter-transformation. Note that any further transformation can be implemented rather easyly. Let us assume we would be interested in a transformation based on a 12-parameter approach. Obviously this transformation is not implemented so far.

However, if we upgrade the program by a class *Spatial_Affinity_Trafo* including a method *datum*, which comprises the appropriate algorithm, we can

expand the functionality by this feature. At least we have to keep in mind that we need to fit the method *apply_strategy* (see listing 6) and the *controller*. All other parts of the program, especially the main program, remain unaffected. This short example makes clear that the strict design facilitates the maintenance and benefits the clearness of the program.

Finally we will modify the example of section three to the extent of a datum transformation. Therefore we assume, that now the user not only wants to change the zone number of a UTM-coordinate but he wants to transform the UTM-coordinate to a local coordinate system which may be the Gauss-Krueger-system. Hence, as already mentioned, the geodetic datum of both systems is different. To keep the example quite simple we omit all the interactions concerning the GUI (see figure 1) and start with the creation of the UTM-coordinate which represents the first vertex of the graph (see line 5 in listing 4).

Once the class UTM is created there exists also an edge to the vertex *geographic coordinate* as already outlined in section 3. These vertex is at the same time an interface for the strategy pattern which delivers as an output a class of the same type. However, the *geographic coordinate* is now transformed to an new geodetic datum. Actually there are different possibilities to transform the coordinate (see figure 4). Nonetheless, all different transformations can be treated as edges as well. Hence the graph is still directed and remains acyclic.

Referring to figure 2 we have now reached the vertex on the right top level side. We now pass the last edge by creating a plane coordinate of the type Gauss-Krueger and transform the geographic coordinate. The last step is exactly the same as in the example in section 3 (see line 9 in listing 4).

5. THE SINGLETON

A thorough analysis of the acyclic graph reveals that in any case an instance of the type *geographic coordinate* is needed. This instance is a hub for the strategy pattern. Further on we have to keep in mind that each transformation is treated as a datum transformation. So the instance *GeographicCoordinate* is the most important in the whole graph. Thus we try to assure that there exists exactly one instance of such a type at any time.

Therefore we have to expand the class *GeographicCoordinate.* The modified class is shown in listing 8. The major difference to the former definition is the new member function called *getInstance*. This function assures the existence of exactly one instance of the class *GeographicCoordinate* because if there already exists such an instance the method *getInstance* rejects the generation of another one.

Listing 8:

```
final class GeographicCoordinate extends Coordinates {
 ...
 private static GeographicCoordinate geo = null;     // line 3
```

```
  ...
  private GeographicCoordinate() {}                    // line 5
  ...
  public static GeographicCoordinate getInstance() {
    if (geo == null)
      geo = new GeographicCoordinate();
    return geo;
  } // end getInstance
} // end GeographicCoordinate
```

Note that there are some other statements which are especially necessary if the programming language *Java* is used (line 3 and line 5). For other languages (e.g. *Eiffel*) there may be other statements required.

One of the main benefits of the design pattern *singleton* is the improvement of the program performance, because the system calls for the allocation of memory are very time consuming. But this is exactly what the *singleton* avoids. Thus it may be of interest to implement also other instances as a *singleton*. This is useful if we think of the transformation of numerous points. In this case it would make sense to implement the whole path or the elements of this path respectively as *singletons*. Then only the vertices and edges needed by the transformation exist and nothing else. Hence it would be difficult or almost impossible to parallelize the transformation process.

6. SUMMARY

Modern concepts of information science deliver new approaches in software engineering. Among these are the design patterns. In this paper the benefits in the field of coordinate transformation are outlined.

From a purely mathematical point of view the whole transformation process can be compared with a path in a directed acyclic graph. The different stages of a coordinate within this process can be seen as vertices and the according algorithms are the edges. Thus the graph consists as a set of all possible stages of all coordinates. Hence, according to the requirements of the user, a certain coordinate in a given geodetic datum passes different stages until the goal is reached.

The implementation of this theoretical concept can be done in combination with design patterns. Now all vertices and edges are treated as instances and member functions. Thus step by step the program passes the different vertices and edges which are in fact the different stages of a coordinate during the transformation process until the final goal is reached. This is normally a coordinate in a certain local system and geodetic datum respectively.

Due to this approach the maintenance is quite simple. In the developing phase errors can be localised fast and extensions can be implemented easily. Furthermore

the structure of the program is quite strict and thus new developers will be soon familiar with the concept and as a consequence the initial training will be shortened.

REFERENCES

GAMMA, E.: *Design Patterns* – Elements of reusable object-oriented software. Addison-Wesley, 2000.

HECK, B.: *Rechenverfahren und Auswertemodelle der Landesvermessung: klassische und moderne* Verfahren. Wichmann, 3. Aufl., 2003.

KUHN, T.: *The structure of scientific revolutions*. Univ. of Chicago Press, 3rd edition, 1996.

(Invited paper. Recebido em agosto de 2007)