Design Patterns in GIS Development: The Terralib Experience

Gilberto Câmara[1], Lúbia Vinhas[1], Ricardo Cartaxo Modesto de Souza[1],

João Argemiro Paiva[1], Antonio Miguel Vieira Monteiro[1],

Marcelo Tilio de Carvalho[2], Baudouin Raoult[3]

[1]Image Processing Division, National Institute for Space Research (INPE), São José dos Campos, Brazil

{lubia, gilberto, cartaxo, miro, miguel}@dpi.inpe.br

[2]Computer Graphics Group (TECGRAF), Catholic University of Rio de Janeiro, Brazil

tilio@tecgraf.puc-rio.br

[3]European Centre for Medium-Range Weather Forecasts, Shinfield Park, Reading, UK

baudouin.raoult@ecmwf.int

Abstract. This paper discusses the use of "design patterns" in the conception and implementation of an open source GIS library called TerraLib. The work indicates how some well-known design patterns (such as Singleton, Composite and Factory) can be effectively used in the development of GIS applications.

Keywords. GIS, software engineering, design patterns, spatial information engineering.

1 Introduction

This work discusses the use of the concepts of "Design Patterns" in the context of the development of GIS systems. The authors argue that these ideas are well suited to capture the complexity of the components of a GIS and that software developers in the spatial information engineering area can benefit substantially by using patterns in their development.

A *design pattern* can be defined as a common solution to recurring problems in software design. It encapsulates the problem, its context and a proposed solution including classes and their rôles and collaborations. Patterns help create a shared language for communicating insight and experience about these problems and their solutions (Appleton, 2000).

Following the wide acceptance of the book *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma, Helm et al., 1995), the concept of patterns has emerged as one of most important areas of software engineering. The aim of the patterns movement is to create a body of literature to help software developers share experience and code. The basic motivation is that all software projects face a large number of common problems, which are usually solved by the programmer on his own. The programmer may be unaware that an experienced colleague, who has produced a simple and elegant solution, has solved a very similar problem before.

Important books dealing with patterns include Gamma et al (1995), Buschmann et al (1996) and Vlissides (1998).

GIS software development has much to benefit from the use of design patterns. To begin with, the GIS community lacks open source software tools that would allow research results and academic prototypes to be widely shared. Since GIS is an applied science area, academic ideas need to be experienced and tested in real-life applications. Unfortunately, GIS software development in academic groups has been very limited, given the complexities of spatial data structures and algorithms and the fact that most GIS software development has taken place in private companies. By contrast, the computer science community has benefited enormously from the availability of software tools such as the Linux operating system, the GNU compilers and the mySQL and PostgreSQL data base management systems. In other words, the evolution of Geographical Information Science needs a similar evolution in Geographical Information Engineering.

We consider that Design Patterns can provide a common ground for GIS academic software development, promoting reuse and sharing experience. Of particular importance in the case of GIS is the combination of the idea of Design Patterns with the emerging paradigm of Generic Programming (Austern, 1998). This paradigm relies on a programming language's support for parameterised classes (templates), and has been made increasingly popular in the C++ environment since the

adoption of the STL library as part of the ISO C++ standard. By expressing the patterns as templates and deriving specific classes by inheritance, pattern reuse is substantially enhanced.

We will illustrate the use of design patterns in GIS in the context of the development of the **TerraLib** open source library. Earlier works (Gordillo, Balaguer et al., 1997; Lisboa and Iochpe, 1999) have discussed the use of design patterns in the derivation of the architecture of a GIS.

Lisboa and Iochpe (1999) propose **GeoFrame**, a conceptual framework for GIS design, based on a generic data model of spatial data types. The authors propose that specific models for GIS applications can be reduced to a number of typical *analysis patterns*. Their proposal is therefore directed towards conceptual data modelling of a GIS and not towards the actual implementation of a GIS system.

Gordillo et al (1997) go a step further, by suggesting that some new types of design patterns specific to GIS environments. Particularly interesting is their proposals of the *Role* and the *Appearance* pattern. The *Role* pattern allows the same geo-object to have different functions in a GIS application, and to be grouped in different layers according to a query operation. The *Appearance* pattern aim at distinguishing between the object in a spatial database and its visual presentation.

Whilst the emphasis of these authors has been in proposing patterns that could be use in the process of GIS design, our work emphasizes the actual use of patterns in GIS software development. Since our aim is developing a general-purpose GIS component library, our primary aims are code reuse, readability and clarity of expression. Therefore, our examples are concentrated on the use of patterns on a real-life application.

2   General Description of TerraLib

TerraLib is a GIS component library being developed by INPE (National Institute for Space Research), TECGRAF/PUC-RIO (Computer Graphics Group at the Catholic University in Rio de Janeiro) and PRODABEL (Informatics Corporation for the City of Belo Horizonte), available from the Internet as open source. Its main aim is to enable the development of a new generation of GIS applications, based on the technological advances on spatial databases (Câmara, Souza et al., 2000). The basic idea behind TerraLib is that the current and expected advances in database technology will enable, in the next few years, the complete integration of spatial data types in data base management systems (DBMS). This integration is bound to change completely the development of GIS

technology, enabling a transition from the monolithic systems of today (that contain hundreds of functions) to a generation of spatial information appliances. The transition from file-based GIS systems to spatial databases will enable different applications to use the same data, as also is being proposed by the OpenGIS consortium.

On a practical side, TerraLib enables quick development of custom-built applications using spatial databases. Currently, such capabilities are only available by means of proprietary solutions such as COM components available in products such as MapObjects, GeoMedia and ARC/INFO-8. These components are based on transitional technologies that either duplicates in memory the data available in the DBMS, or uses additional access mechanisms such as ArcSDE. TerraLib aims to improve on such capabilities, by providing direct access to a spatial database, without unnecessary middleware.

As a research tool, TerraLib aims to enable the development of GIS prototypes that include new concepts such as spatio-temporal data models (Hornsby and Egenhofer, 2000), geographical ontologies (Fonseca and Egenhofer, 1999) and advanced spatial analysis techniques (Heuvelink, 1998).

The library development has been divided in three components:

- **kernel**: composed of classes for storing geometries and attributes in an object-relational DBMS, such as ORACLE and PostgreSQL, cartographic projections, topological and directional operators and spatio-temporal models. Kernel maintenance and upgrade is the responsibility of the project core team, as typical for other free software projects.

- **functions**: algorithms that use the kernel basic structures, including spatial analysis, query and simulation languages, and data conversion procedures. Again, maintenance and upgrade is the responsibility of the project core team, but it is expected that new functions developed by external collaborators will be incorporated.

- **contrib**: applications built by users of TerraLib, including external authors, who are responsible for their maintenance.

3 Design Patterns in TerraLib

In the next sections, we present code excerpts from actual TerraLib classes, which have been designed using the concepts of design patterns. We have chosen to discuss the real C++ code, rather than generic diagrams, since it provides a better understanding of the rationale behind our choices.

In order to support reuse of patterns in Terralib, one of our basic ideas was to combine the notions of generic programming (templates) with object-oriented programming (inheritance). The idea is to define a design pattern as a template and use this pattern in two steps. First, a base class is defined as an instantiation of this template and then the desired class can be defined as a specialisation of this base class. Consider the following code extract:

```
// generic pattern
template class<T>
class Pattern<T> {
//  ...
}
// pattern applied to a specific class
class Example: public Pattern<Example> {
// ...
}
```

What's happening here? The new class `Example` has been defined as inheriting from an instantiation of a template, where the instantiated type is the class itself. This odd-looking derivation is in fact quite logical, since we want the new class to have the same behaviour as the pattern. This type of combination is discussed in Coplien (1996), who describes different examples of its use and considers this combination of templates with inheritance as a very powerful way of promoting code reuse.

4 Singletons and Co-ordinate Precision

A Singleton is a class that has only one instance, with a global access point (Gamma, Helm et al., 1995). One of the many uses of the singleton pattern in a GIS is the precision associated with geometric operations such as equality of co-ordinates. In this case, we want to use the same value of precision for all operations involving co-ordinates.

The Singleton pattern template is defined as having a protected constructor (accessible only by its descendants), and is accessible externally only by means of a static function (`Instance`). The singleton instance is created only once, in the first time this function is called.

```
template class<T>
class TeSingleton<T> {
public:
    static T& instance () {
        static T unique_;
      return unique_; }
protected:
// -- Constructor
    TeSingleton() {}
};
// example of using a singleton
class TePrecision:
public TeSingleton<TePrecision> {
    void setPrecision ( double precision )
        { precision_ = precision; }
    double precision ()
       { return precision_; }
protected:
    double precision_;
}
// Usage of TePrecision
class TeCoord2D {
double x, y_;      // co-ordinates
public:
// ... other operations omitted
    bool operator == (const TeCoord2D& cd)
    { return ((fabs(y_- cd.y)<
TePrecision::Instance().precision() )
       && (fabs(x_- cd.x) <
TePrecision::Instance().precision())))}
}
```

In this case, we consider that the desired precision has been set previously (based on issues such as scale and projection) and that this unique value can be used by all applications. An additional advantage of using a Singleton class is avoiding global variables, always a problem when building a library.

5 Counted Instances and the "Pimpl" Idiom

One of the important issues when dealing with geometric data structures is optimising memory usage. Since many maps have a large number of polygons or lines, copy operations can become very costly. To illustrate the idea, consider the following code fragment:

```
// Creation of a polygon set
TePolygonSet pSet;
//... put data in pset (not shown)
// Copying a polygonSet
TePolygonSet pSet2 = pSet;
// what happens here?
```

In this case, copying all polygons from the first set to the second set can be a very costly operation. In order to avoid such unnecessary memory congestion, we have used the so-called "pointer to implementation" or "pimpl" idiom (Sutter, 2000), which is also called the "Bridge" pattern (Gamma, Helm et al., 1995). This programming idiom proposes a separation between a class and its implementation, by using an opaque pointer to hide the implementation details. De-coupling the abstraction from its implementation allows different instances of a class to share the same implementation. Therefore, no copying will take place, but simply the new object points to the same memory area that contains the data for the first polygon set.

The "pimpl" idiom is usually combined with the idea of a "counted" pattern, a class that keeps track of how many abstract instances are pointing to the same implementation. In Terralib, the "counted" class is implemented as follows:

```
class TeCounted {
public:
// -- Constructor
    TeCounted(): refCount_ ( 0 ){}
// -- Methods
    void attach () { refCount_++; }
    void detach ()
    { if ( --refCount_ == 0 )
        delete this; }
private:
    int refCount_; // members
};
```

The idea here is that all constructors and copy operators for classes which are derived from the "counted" class will increments the number of its virtual instances (refCount_) using the attach method. Conversely, when an instance of a counted object is destroyed the number of virtual references to it is decremented. When the reference count reaches zero, which means that there are no virtual instances of it, the object is effectively destroyed. This is done by the using the detach method.

The use of the "pimpl" and "counted" idioms is best considered in the context of constructing the geometric structures of TerraLib, as explained in the next section.

6 Composite Pattern and Vector Geometries

The *vector data structures* used for representation of spatial information (points, lines, polygons, triangular meshes) can be directly represented by a Composite pattern. This pattern *"defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively."* (Gamma, Helm et al., 1995). As the Composite pattern can be used in many components of a GIS we propose a parameterised implementation of it (shown partially below):

```
template <class T>
class TeComposite: public TeCounted {
public:
        // Add a new component
        void add ( const T& elem )
        { components_.push_back ( elem );

        // Return the i-th element
        T& operator [] ( int i )
        { return components_[ i ]; }


protected:
        vector<T>  components_;
};
```

Based on this pattern, we can now define a "Geometry Composite", a basic template which will be used to derive all vector geometries and that supports the idea of counted instances, for more efficient memory usage.

```
// Geometry composite (simplified version)
template <class T>
class TeGeomComposite {
protected:
    TeComposite<T> * pImpl_;
public:
    // Constructor
    TeGeomComposite() {
        pImpl_ = new TeComposite<T>;
        pImpl_->attach(); }

// Destructor
    virtual ~TeGeomComposite()
    {   pImpl_->detach();  }
// Operator =
    TeGeomComposite& operator=
    ( const TeGeomComposite& other ) {
        if ( this != &other ){
            other.pImpl_->attach();
            pImpl_->detach();
            pImpl_  = other.pImpl_;}
        return *this;
    }
```

```
    // Add a new component
    void add ( T& elem ) {
        pImpl_->add ( elem ); }
    // Return the i-th element
    T& operator[] ( int i )
    {   return pImpl_->operator[] ( i ); }
};
```

As can be seen in the above definition, a geometry composite is a generic geometry whose implementation is a pointer to an instance of a Composite template. This idea allows for multiple copies of the same data to point to the same memory area and enables a simple definition for the vector geometries in TerraLib. For example, consider the definitions of TeLine, TeLinearRing, TePolygon, and TePolygonSet:

```
// A class for handling lines
class TeLine2D :
    public TeGeomComposite<TeCoord2D> {};

// A class for handling rings (closed lines)
class TeLinearRing : public TeLine2D {
public:
    // -- Contructor from a line
    TeLinearRing ( TeLine2D& line ); };
// a class for handling polygons
class TePolygon:
  public TeGeomComposite<TeLinearRing> {};
// a class for handling sets of polygons
class TePolygonSet:
  public TeGeomComposite<TePolygon> {};
```

As the code shows, the TePolygonSet class is a composite of TePolygon. A TePolygon is a composition of TeLinearRing. A TeLine is a composite of TeCoord2D. Each new class has been defined in a straightforward fashion, achieving substantial code reuse and economy of expression.

7   Factories and Projections

The "Factory" pattern defines an interface for creating an object and delegating the instantiation to another object. In this pattern *"the creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate concrete product"* (Gamma, Helm et al., 1995). This pattern is used whenever we want to create a specific subclass of an abstract base class based on type information. For example, consider the case of selecting a cartographic projection, given an abstract base class TeProjection. A naïve code would be similar to the following:

```
// selection of a projection
// given a projection name
TeProjection* proj;
if ( name == "UTM" )
    proj = new TeUTM (params);
elseif ( name == "Albers" )
    proj = new TeAlbers (params)
// and so on...
```

Why is this coding style not to be recommended? In an open source library such as TerraLib, where contributions from many partners are expected, each new projection contributed to the code would require changes in all applications that use the projection facilities. In order to avoid this problem, we would like that each new projection class can be automatically included in a projection list, without requiring code recompilation. This can be achieved by a generic factory pattern, as follows:

```
// Factory template (short description)
// uses STL map
template <class T, class Arg>
class TeFactory {
public:
// -- Dictionary of factories
typedef map<string, TeFactory<T,Arg>*>
TeFactoryMap;

// -- Normal Constructor
    TeFactory ( const string& fName)
    { fMap_ [ fName ] = this; }

// -- Builds a new type (should be
implemented by descendants)
    virtual T* build (const Arg& arg) = 0;

// -- Virtual Constructor
    static T* make
    ( const string& name, const Arg& arg );

private:
    // factory list (static)
    static TeFactoryMap fMap_;
};
```

```
// Virtual Constructor implementation
// (called by applications)
template <class T, class Arg>
T*
TeFactory<T,Arg>::make
( const string& name, const Arg& arg )
{
    // try to find the name on the
    // factory dictionary
    TeFactoryMap::iterator i =
            fMap_.find ( name );

    // Not found ?  return zero
    if ( i == fMap_.end() )
        return 0;
    // Create an object,
    // based on the input parameters
    return (*i).second->Build ( arg );
}
```

How does this template code work? The basic idea is to create "families of factories". First, an abstract base class will be created, based on an instantiation of this template. Then, a set of concrete factories will be defined as descendants of this base class. Each new concrete factory, when created, will be placed on the factory map, which will contain a list of factories, indexed by their name (which must be unique).

This pattern relies on the fact that all static objects are created by the compiler before the program starts its execution. If one instance of each concrete factory is defined as a static variable, a pointer to it will be placed on the factory dictionary (the fMap variable in the template). Therefore, when the make method is called, it will search in its list, and will find the appropriate factory.

The advantage of this code is that there is no need for an explicit "if...elseif..." piece of code which would require constant maintenance. Each new subclass will have an associated factory, that, when created, will be automatically "registered" in the factory list. Although the code may look a bit awkward for programmers not at ease with templates, its application is relatively straightforward, as shown is the following example.

In TerraLib, we have used this template for handling multiple cartographic projections. We have defined a base projection class called TeProjection and an auxiliary class TeProjParams that holds all the parameters that define a specific projection. To define a "factory of projections", the first step is to define an abstract factory pattern to build different cartographic projections, and then to define concrete factories for each specific projection.

The following code shown the definition of a specialisation of TeProjection, to handle UTM projections. Note that we create two new classes TeUTM for the projection and a factory called TeUTMFactory. It is also required that a static instance of the factory is defined, which will be registered in the "factory list" before execution takes place.

```
// abstract class for projection factories
class TeProjectionFactory:
public TeFactory<TeProjection,TeProjParams>
{}
// concrete factory for UTM projections
class TeUTMFactory:
public TeProjectionFactory
{
public:
// called by "make" virtual constructor
virtual TeProjection* build
    ( TeProjParams& arg )
    {  return new TeUTM( arg ); }
}
// actual UTM projection class
class TeUTM: public TeProjection {
public:
    // -- constructor
    TeUTM( TeProjParams& arg );
}
// Static instance of the UTM proj factory
// (will be placed in the factory list)
static TeUTMFactory fact("UTM");
```

The inclusion of a new projection and its factory does not require recompilation of the existing classes. The interface for all applications that use cartographic projections will then be very simple, consisting of a single line of code:

```
// Projection parameters for a projection
TeProjParams par;

// The projection factory builds an instance
of  projection
TeProjection* proj =
TeProjectionFactory::make("UTM", par);
```

The factory template is extremely useful for GIS library development. In GIS application, there are a large number of situations where the subclass to be instantiated depends on a type field. Consider the following examples:

- selection of a database driver based on a database name.

- choice of a data conversion function based on file type.

- selection of an appropriate algorithm, given a family of similar techniques (such as the line simplification case).

In all these and similar situations, the "factory" pattern proves to be an appropriate solution. The additional care to be exercised when building the classes will result in much benefit later.

8 In Conclusion: Using Design Patterns for GIS

This paper has given examples of the use of design patterns in the implementation of a particular GIS library (TerraLib), and we conclude with some general remarks. In large-scale programming projects, which involve the collaboration of many users at different places, it is extremely important to propose and enforce coding practices that enhance reusability and reduce the impact on existing code.

We have shown concrete situations where design patterns provide substantial improvements, by reducing memory usage and avoid code duplication (case of the Composite pattern) and by easing code maintenance (as in the case of the Factory pattern). In resume, we hope to have shown that a judicious combination of Design Patterns and Generic Programming is recommended for building reliable and maintainable GIS implementations.

References

Appleton, B. (2000). Patterns and Software: Essential Concepts and Terminology. **2001**. <http://www.enteract.com/~bradapp/docs/>.

Austern, M. (1998). *Generic Programming and the STL : Using and Extending the C++ Standard Template Library*. Reading, MA, Addison-Wesley.

Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad and M. Stal (1996). *Pattern-Oriented Software Architecture*. New York, John Wiley.

Câmara, G., R. Souza, B. Pedrosa, L. Vinhas, A. Monteiro, J. Paiva, M. Carvalho and M. Gattass (2000). *TerraLib: Technology in Support of GIS Innovation*. II Workshop Brasileiro de Geoinformática, GeoInfo2000.

Coplien, J. (1996). Curiously Recurring Template Patterns. In: S. Lippman, *C++ Gems*. New York, SIGS Books: 135-143.

Fonseca, F. and M. Egenhofer (1999). *Ontology-Driven Geographic Information Systems*. 7th ACM GIS Symposium, Kansas City, MO, ACM Press, N.Y.

Gamma, E., R. Helm, R. Johnson and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, Addison-Wesley.

Gordillo, S., F. Balaguer and F. das Neves (1997). *Generating the Architecture of GIS Applications with Design Patterns*. 5th ACM GIS Symposium, Las Vegas, ACM Press.

Heuvelink, G. (1998). *Error Propagation in Environmental Modelling with GIS*. London, Taylor and Francis.

Hornsby, K. and M. Egenhofer (2000). "Identity-Based Change: A Foundation for Spatio-Temporal Knowledge Representation." *International Journal of Geographical Information Science* **14**(3): 207-224.

Lisboa, J. and C. Iochpe (1999). *Specifying Analysis Patterns For Geographic Databases on the basis of a Conceptual Framework*. 7th ACM GIS, Kansas City, ACM Press.

Sutter, H. (2000). *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Exception-Safety Solutions*. Reading, Addison-Wesley.

Vlissides, J. (1998). *Pattern Hatching: Design Patterns Applied*. Reading, MA, Addison Wesley.